Hygienic Macro Expansion

Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, Bruce Duba

Computer Science Department
Lindley Hall 101
Indiana University
Bloomington, Indiana 47405 USA

Abstract. Macro expansion in current Lisp systems is naïve with respect to block structure. Every macro function can cause the capture of free user identifiers and thus corrupt intended bindings. We propose a change to the expansion algorithm so that macros will only violate the binding discipline when it is explicitly intended.

1. Problems with Macro Expansions

Lisp macro functions are powerful tools for the extension of language syntax. They allow programmers to add new syntactic constructs to a programming language. A programmer specifies a macro function which translates actual instances of a syntactic extension to core language expressions. This process can also be pyramided, *i.e.*, macro functions may translate into an already extended language [5]. The defined set of macro functions is coordinated by a preprocessor, usually called a macro expander. The macro expander parses every user input. If the expander finds an instance of a syntactic extension, it applies the appropriate macro function. It repeats this process until an expression of the core language is obtained.

In most current Lisp systems the expander's task is confined to the process of finding syntactic extensions and replacing them by their expansions. This implies, in particular, that each macro function is responsible for the integrity of the program. For Lisp systems (and other languages with similar macro facilities) this means specifically that variable bindings must not be corrupted. This, however, is not as simple a task as it sounds.

Macro functions in Lisp generally act like the context filling operation in the λ-calculus [2], p.29. Given its textual parameters, a macro function places them into the appropriately labeled holes of some expansion pattern. Free identifiers in user code may unintentionally be captured by macro-generated bindings. For example, a macro function for **or**-expressions in Lisp may be understood as a transformation from patterns of the type

$$(\textbf{or}\ \langle exp\rangle_1\ \langle exp\rangle_2)$$

to an expansion pattern like

$$(\textbf{let}\ v\ [\ ]_{\langle exp\rangle_1}\ (\textbf{if}\ v\ v\ [\ ]_{\langle exp\rangle_2})).^1$$

In other words, the **or**-macro fills the hole $[\ ]_{\langle exp\rangle_i}$ with $\langle exp\rangle_i$. An instance like (**or** nil v) is transcribed to (**let** v nil (**if** v v v)). This example reveals that the capturing of free user identifiers is dangerous. The expanded expression will always produce the value nil, independently of the value of the user identifier v, quite contrary to the expectations of a programmer.

The real danger of these erroneous macros is that they are treacherous. They work in all cases but one: when the user—or some other macro writer—inadvertently picks *the* wrong identifier name.

Various techniques have been proposed to circumvent this *capturing problem*, but they rely on the individual macro writer. If even one of the many macro writers is negligent, the macro system becomes unsafe. We claim that the task of safely renaming macro-generated identifiers is mechanical. It is essentially an α-conversion [2], p. 26, which is knowledgeable about the origin of identifiers. For these reasons we propose a change to the naïve macro expansion algorithm which automatically maintains hygienic conditions during expansion time.

The rest of the paper is devoted to the presentation of the problem definition and the new algorithm. The second section describes our target programming language and its macro expander language. In the third section we discuss the naïve macro expansion algorithm. Section 4 contains the hygienic expansion algorithm and a correctness theorem. In Section 5 we show how to extend the

1 This might be expanded to ((**lambda** v (**if** v v $[\ ]_{\langle exp\rangle_2}$)) $[\ ]_{\langle exp\rangle_1}$).

solution to cover important constructs of Lisp. The last section highlights the merits of the new algorithm and its implications for macro writers.

2. Language Considerations

A macro expander maps expressions from an extended programming language to an expression in a core language. Hence, our first consideration must be the source and target language of the expander.

The most important aspect of the target language with respect to the capturing problem is its lexical scoping mechanism. We have chosen to use the λ-calculus as it is the prototype of block structured programming languages. It is syntactically simple, yet contains all the required elements to make the case interesting, and has the right level of complexity. Furthermore, it is a fairly trivial task to generalize an algorithm for the λ-calculus to Common Lisp [6], Scheme [3], or Algol-like languages.

The variant of the λ-calculus we use as our target programming language is defined by the grammar:

$$\begin{aligned} \lambda term ::= \; & var \\ & \mid const \\ & \mid (\textbf{lambda}\ var\ \lambda term) \\ & \mid (\lambda term\ \lambda term). \end{aligned}$$

The characters "(", ")", and "**lambda**" are terminal symbols and are collectively referred to as the set of core tokens: $coretok = \{(,), \textbf{lambda}\}$. The set *const* includes all constants commonly found in Lisp such as strings, vectors, numbers, closures, *etc.* The set *var* is composed of Lisp symbols that are used as identifier names; it is disjoint from the set of core tokens.

Variable and constant expressions stand for values as usual. Abstractions, *i.e.*, **lambda**-expressions, represent procedures of one variable. The **lambda**-bound identifier—the parameter—can only be referred to within the abstraction body, *i.e.* identifiers are lexically scoped. We call the occurrence of a variable in the parameter part of a **lambda**-expression its *binding instance*. Applications correspond to function invocations.

The source language needs to be an extension of the target language. It must allow for one kind of an expression which is only specified in a rather general way. The concrete extension of $\lambda term$ as defined by the indivdual macros will be specializations of this language. We refer to the source language as the language of syntax trees and define it inductively by:

$$\begin{aligned} stree ::= \; & var \\ & \mid const \\ & \mid mstree \\ & \mid (\textbf{lambda}\ var\ stree) \\ & \mid (stree\ stree). \end{aligned}$$

The set *mstree* is the sublanguage of *syntactic extensions*.

We assume that instances of macro expressions are recognized by the presence of macro tokens, *i.e.* elements of a distinguished set *mactok*. Macro tokens can either be syntactic extensions by themselves or are the first component of an arbitrarily long syntax tree:

$$mstree ::= mactok \mid (mactok\ stree_1 \ldots stree_n)$$
$$\text{for all } n \geq 0.$$

Since this syntax is ambiguous, we add the provision that an expression of the form $(m\ s)$ with $m \in mactok$ and $s \in stree$ is a syntactic extension; it is not an application. See Figure 1 for a summary of the definitions.

3. The Naïve Approach to Macro Expansion

Before we can describe the expansion algorithm which is currently employed in Lisp systems, we need to define some terminology. Recall that a possible form for a syntactic extension is:

$$(mactok\ stree_1 \ldots stree_n).$$

The trees $stree_1$ through $stree_n$ are called the *syntactic scope* of the extension.

We say a syntactic extension or any $\lambda term$-expression *occurs* in a syntax tree if it is a subtree that is not nested within the syntactic scope of another syntactic extension. For example, the syntactic extension (**or** x y) occurs within the expression

$$(\textbf{lambda}\ x\ (\textbf{or}\ x\ y)),$$

but it does not occur within

$$(\textbf{case}\ tag\ (name\ (\textbf{or}\ x\ y)))$$

nor within

$$(\textbf{case}\ tag\ (\textbf{or}\ x\ y))$$

because it is in the syntactic scope of the **case**-extension.

The notion of occurrence reflects the fact that every sentence in the language of syntax trees has two interpretations. It may be considered as an element of the $\lambda term$-language or as a proper syntactic extension. Since the expansion of a syntactic extension involves a rearrangement of (parts of) syntax trees in its syntactic scope, we can only be sure about the interpretation of an expression when it is not embedded in a syntactic extension. In the above example, the list (**or** x y) is in the first two cases a syntactic extension; in the third one, however, it only stands for a list with **or** in the first position and x and y in the rest of the list. The same is true if we replace the **or** by **lambda**.

A *syntactic transform function* ($\in STF$) is a macro function which is defined by the macro writer and which expands a particular class of syntactic extensions, *e.g.*,

the **or**-macro of Section 1. The result of applying a transform function to an occurrence of a syntactic extension is called a *transcription*. A *transcription step* is the one-step expansion of a syntactic extension. Symbols which are introduced during a transcription step are referred to as *generated symbols*.

The set of macros used during an expansion is a *syntax table* ($\in ST$). Applied to an occurrence of a syntactic extension it produces its transcription. It serves as a dispatcher and applies the appropriate transform function to its argument.

Equipped with these definitions, we can define the *expansion* of a syntax tree with respect to a syntax table as the tree in which all occurrences of syntactic extensions are replaced by the expansions of their transcriptions. If the expansion process halts, the result of an expansion is a $\lambda term$, *i.e.*, an expression of the core language. We have formalized these definitions in Figure 1.

The major problem with naïve expansion is that it does not enforce the integrity of lexical bindings. This point was illustrated in Section 1 with the incorrect expansion of an **or**-expression. The example may suggest that one can simply rename all generated identifiers after each transcription step. But this impression is too simplistic. Generated identifiers may act as free variables which are to be captured by the user-supplied program context. They must not be renamed. On the other hand, since macro expansion is possibly pyramided it may also not be quite obvious after a transcription step which identifier is to be free and which one is to be bound. A final difficulty is that sometimes capturing is desired. Consider a loop-macro which transforms patterns of the form

(loop (block))

by filling the expansion pattern

$$((\mathbf{Y}\ (\mathbf{lambda}\ f\ (\mathbf{lambda}\ c\ (f\ |\ |_{(\mathbf{block})})))))\ 1).$$

This fill-operation of contexts captures the free identifiers f and c in the expression (block). The capturing by f is almost certainly undesired, but the one by c may be quite useful. The identifier c is always bound to the result of the last iteration step, initially it is 1, and it might be necessary to have this value around. Given a protocol, which indicates this binding of c within the syntactic scope of **loop** for the macro user, one can imagine that the results of applying the macro function must capture free c's in

Figure 1: Naïve macro expansion

Syntactic Domains:

$c \in const$	constant names,
$v \in var$	identifier names,
$m \in mactok$	macro tokens,
$e \in mstree$	macro expressions,
$s \in stree$	expressions.

Syntax:

$$s ::= c \mid v \mid e \mid (\mathbf{lambda}\ v\ s) \mid (s_1\ s_2),\ s_1 \notin mactok,$$
$$e ::= m \mid (m\ s_1 \ldots s_n) \text{ for } n \geq 0$$
with the above restriction.

Semantic Domains:

$$STF = mstree \rightarrow stree,$$
$$\vartheta \in ST = mstree \rightarrow stree,$$
$$\lambda term.$$

Semantic Functions:

$\mathcal{E}_{naive} : stree \rightarrow ST \rightarrow \lambda term;$

$$\mathcal{E}_{naive}[\![c]\!] = \lambda\vartheta.c,$$
$$\mathcal{E}_{naive}[\![v]\!] = \lambda\vartheta.v,$$
$$\mathcal{E}_{naive}[\![e]\!] = \lambda\vartheta.\mathcal{E}_{naive}[\![(\vartheta e)]\!]\vartheta,$$
$$\mathcal{E}_{naive}[\![(\mathbf{lambda}\ v\ s)]\!] = \lambda\vartheta.(\mathbf{lambda}\ v\ \mathcal{E}_{naive}[\![s]\!]\vartheta),$$
$$\mathcal{E}_{naive}[\![(s_1\ s_2)]\!] = \lambda\vartheta.(\mathcal{E}_{naive}[\![s_1]\!]\vartheta\ \mathcal{E}_{naive}[\![s_2]\!]\vartheta).$$

(block) but must avoid the capturing of f's. The situation looks hopeless for a mechanized solution. Therefore, people have invented a variety of techniques which give the designer of the macro functions a mechanism for avoiding capture when required.

One of the common solutions to the capturing problem uses bizarre or freshly created identifier names for macro-generated bindings. Another solution involves the freezing—closing—of user-code at the right time in the correct environment [7]. It is clear that bizarre names only lower the probability of the problem occurrence, but do not eliminate it. The freshly created identifier approach works if the macro writer always specifies which identifiers are to be so considered. Freezing and thawing user-code is even more complicated since it has to be done in the right environment. All of these solutions suffer from the same drawback; the macro writer is responsible for their realization. If he is negligent, the macro is insidious.

In the next section we present an expansion algorithm which automatically resolves the problem and requires little modification to the conventional macro writing style. It relies on the fact that almost all generated identifiers are to be freshly created, and it allows exceptions from this default assumption when necessary.

4. Hygienic Macro Expansion

The capturing problem of the naïve expansion algorithm is analogous to the substitution problem in the λ-calculus. When an expression M with free variables is to be substituted into an expression N, the binding variables of N must be different from the free ones in M. Put differently, bindings in N must not capture free variables in M. Kleene calls this condition "being-free-for"[4]; the term "hygiene condition" is a more informal but rather descriptive name for it [1].

So, what we want to impose on macro expansion is something like a hygiene condition. With a few exceptions, we do not want generated binding instances created by one transcription step to capture user-supplied variables or variables from some other transcription step. Thus, not taking intended capturings into account, we formulate the

Hygiene Condition for Macro Expansion.
Generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step. (HC/ME)

From the λ-calculus, one knows that if the hygiene condition does not hold, it can be established by an appropriate number of α-conversions. That is also the basis of our solution. Ideally, α-conversions should be applied with every transformation step, but as we have discussed in the previous section, that is impossible. One cannot know in advance which macro-generated identifier will end up in a binding position. Hence, it is a quite natural requirement that one retains the information about the origin of an identifier. To this end, we combine the expansion algorithm with a tracking mechanism.

Tracking is accomplished with a time-stamping scheme. Time-stamps, sometimes called clock values, are simply non-negative integers. The domain of time-stamped variables (*tsvar*) is isomorphic to the product of identifiers and non-negative integers. We sometimes refer to elements of *tsvar* as tokens. The source and target language of the individual macros is extended to the language of time-stamped syntax trees. Time-stamped syntax trees are defined like syntax trees but instead of identifiers they include elements from the union of identifiers and tokens. The formal definition is shown in Figure 2.

Figure 3 contains functions which connect time-stamped domains with pure ones. The function $\$$ takes a time-stamp as an argument and returns a function which injects identifiers into *tsvar* with the given time-stamp. $\$_0$ is the function which stamps an identifier with a 0. It will play a role in the treatment of intended capturings. The function $[v/w]t$ acts like a substitution: given a time-stamped w, a v, and a time-stamped $\lambda term$, it substitutes all free occurrences of w by v. We have omitted the formal definition of the domain of time-stamped $\lambda terms$ but it is the subset of *tstrees* which do not contain syntactic extensions.

The new algorithm consists of four major phases; see Figure 4. It starts out by transforming the user-supplied *stree* into a time-stamped syntax tree. This is accomplished by the function $\mathcal{T}$ which parses the *stree* and stamps all identifier leaves with the function τ. For the initial pass, τ is the function $\$_0 = (\$0)$. Then the real expansion process begins and the clock value is increased to 1.

As before, in the naïve algorithm, the function $\mathcal{E}$ parses through *tstree*-expressions that are also $\lambda term$-expressions. When it discovers a syntactic extension, it generates the appropriate transcription. But instead of immediately continuing, the algorithm first time-stamps all the macro-generated identifiers. Again, this time-stamping process is performed by the function $\mathcal{T}$ in cooperation with the function $(\$j)$, where j is the current clock value. Afterwards, the clock value is increased and the expansion continues.

The result of the function $\mathcal{E}$ is a time-stamped $\lambda term$. It differs from the result of the naïve algorithm. Wherever the naïve result contained a variable, the modified result contains a corresponding time-stamped variable. For example, when the naïve algorithm returned the tree

(lambda x (lambda x ((f x) x)))

the modified result may appear as

(lambda x:0 **(lambda** x:1 ((f:1 x:0) x:1))).

This indicates that according to the HC/ME the naïve algorithm would have gotten the bindings wrong.

The third phase of the modified algorithm replaces all bound, time-stamped identifiers by unstamped identifiers. It is important that we can tell when a token was generated. Tokens with different time-stamps came from different transcription steps, and this difference must be preserved. Since the expression is now a time-stamped *λterm*, α-conversions easily achieve the effect. The function $\mathcal{A}$ parses the term and applies the appropriate substitution function to λ-abstractions. The above example would become something similar to

(lambda a **(lambda** b ((f:1 a) b))).

The bindings are now as intended. The only remaining time-stamped identifiers correspond to free identifiers. Their meanings are determined by the identifier components and, hence, they must be unstamped. This is the task of the function $\mathcal{U}$. It parses the tree and removes all time-stamps. The result of this fourth and last phase is a pure *λterm*:

(lambda a **(lambda** b ((f a) b))).

Before we can examine the similarities and differences between the results of the naïve and hygienic expansion algorithm, we need to discuss the implications of the modified expander on the transform functions. Since we have changed the source and target languages of STFs, we should expect that transform functions must employ different functions. However, the change of languages is really a minor one. Indeed, if we consider time-stamped identifiers simply as a special kind of variable, the transform functions are not changed except that functions that need to know or compare identifier names must unstamp the appropriate token with the function $\mathcal{U}$ (or a respective restriction thereof). Thus, a syntax table ϑ for the naïve expander *induces* a syntax table ϑ' for the hygienic one in such a way that

$$\text{for all } f \in tsmstree, \vartheta(\mathcal{U}[\![f]\!]) = \mathcal{U}[\![\vartheta'(f)]\!].$$

Figure 2: Hygienic macro expansion (1)

Syntactic Domains:

$c \in const$	constant names,
$v \in var$, $w \in tsvar$	(time-stamped) identifier names,
$m \in mactok$,	macro tokens,
$e \in mstree$, $f \in tsmstree$	(time-stamped) macro expressions,
$s \in stree$, $t \in tstree$	(time-stamped) expressions.

We also refer to

$x \in const \cup var \cup mactok \cup coretok$,
$y \in const \cup tsvar \cup mactok \cup coretok$,
$z \in coretok \cup tstree$.

Syntax:

$$s ::= c \mid v \mid e \mid (\textbf{lambda}\ v\ s) \mid (s_1\ s_2),\ s_1 \notin mactok,$$
$$e ::= m \mid (m\ s_1 \ldots s_n) \text{ for } n \geq 0;$$
$$t ::= c \mid v \mid w \mid f \mid (\textbf{lambda}\ v\ t) \mid (t_1\ t_2),\ t_1 \notin mactok,$$
$$f ::= m \mid (m\ t_1 \ldots t_n) \text{ for } n \geq 0$$

with the above restriction.

Semantic Domains:

$$STF = tsmstree \rightarrow tstree,$$
$$\vartheta \in ST = tsmstree \rightarrow tstree,$$
$$\langle \lambda term \rangle.$$

Figure 3: Hygienic macro expansion (2)

Auxiliary Functions:

$\$: N \to var \to tsvar$; $\$\, i\, v = v{:}i$.

$\$_0: var \to tsvar$; $\$_0 = \$\, 0$.

$[\ /\ ]: tsvar \times var \to tstree \to tstree$

where the *tstrees* are restricted to time-stamped λ*terms*;

$$
\begin{aligned}
[v/w_1]w_2 &= w_1 \stackrel{?}{=} w_2 \to v, w_2,\\
[v/w]x &= x,\\
[v/w_1](\textbf{lambda}\ w_2\ t) &= w_1 \stackrel{?}{=} w_2 \to\\
&\qquad (\textbf{lambda}\ w_2\ t),\\
&\qquad (\textbf{lambda}\ w_2\ [v/w_1]t),\\
[v/w](t_1\ t_2) &= ([v/w]t_1\ [v/w]t_2).
\end{aligned}
$$

Figure 4: Hygienic macro expansion (3)

Semantic Functions:

$\mathcal{E}_{hyg}: stree \to ST \to \lambda term$,

$\mathcal{T}: tstree \to (var \to tsvar) \to tstree$,

$\mathcal{E}: tstree \to ST \to N \to \lambda term$,

$\mathcal{A}: tstree \to tstree$,

with the domain restricted to time-stamped λ*terms*,

$\mathcal{U}: tstree \to stree$;

$$\mathcal{E}_{hyg}[\![s]\!] = \lambda\vartheta.\mathcal{U}[\![\mathcal{A}[\![\mathcal{E}[\![\mathcal{T}[\![s]\!]\$_0]\!]\vartheta j_0]\!]\!] \text{ where } j_0 = 1;$$

$$
\begin{aligned}
\mathcal{T}[\![y]\!] &= \lambda\tau.y,\\
\mathcal{T}[\![v]\!] &= \lambda\tau.\tau v,\\
\mathcal{T}[\![(z_1 \ldots z_n)]\!] &= \lambda\tau.(\mathcal{T}[\![z_1]\!]\tau \ldots \mathcal{T}[\![z_n]\!]\tau);
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![c]\!] &= \lambda\vartheta j.c,\\
\mathcal{E}[\![w]\!] &= \lambda\vartheta j.w,\\
\mathcal{E}[\![f]\!] &= \lambda\vartheta j.\mathcal{E}[\![\mathcal{T}[\![(\vartheta f)]\!](\$\, j)]\!]\vartheta(j+1),\\
\mathcal{E}[\![(\textbf{lambda}\ w\ t)]\!] &= \lambda\vartheta j.(\textbf{lambda}\ w\ (\mathcal{E}[\![t]\!]\vartheta j))\\
\mathcal{E}[\![(t_1\, t_2)]\!] &= \lambda\vartheta j.(\mathcal{E}[\![t_1]\!]\vartheta j\ \mathcal{E}[\![t_2]\!]\vartheta j);
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{A}[\![v]\!] &= v,\\
\mathcal{A}[\![y]\!] &= y,\\
\mathcal{A}[\![(\textbf{lambda}\ w\ t)]\!] &= (\textbf{lambda}\ v\ \mathcal{A}[\![[v/w]t]\!])\\
&\qquad \text{where } v \text{ is a fresh variable,}\\
\mathcal{A}[\![(t_1\ t_2)]\!] &= (\mathcal{A}[\![t_1]\!]\ \mathcal{A}[\![t_2]\!]);
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{U}[\![x]\!] &= x,\\
\mathcal{U}[\![v{:}i]\!] &= v,\\
\mathcal{U}[\![(z_1 \ldots z_n)]\!] &= (\mathcal{U}[\![z_1]\!] \ldots \mathcal{U}[\![z_n]\!]).
\end{aligned}
$$

In other words, if we disregard the time-stamps, ϑ and the induced ϑ' generate the same results.

Another relationship that we have to consider is the one between *λterms* as generated by the naïve and hygienic expander. It is clear that the hygienic expansion should work so that the resulting terms are the same except for the bindings. These must respect the HC/ME. We call this relation *structural equivalence* and define it in the following way: Two *λterms* are structurally equivalent if they are equal after replacing all bound variables by the symbol X. Given the notions of induced syntax tables and structurally equivalent terms, we can formalize the difference between hygienic and naïve macro expansion with:

Theorem. *Let ϑ be a syntax table and let ϑ' be the induced syntax table. Then, for all strees P, if $\mathcal{E}_{naive}[\![P]\!]\vartheta$ expands into a λterm, then $\mathcal{E}_{hyg}[\![P]\!]\vartheta'$ expands into a structurally equivalent term which satisfies the hygiene condition for macro expansion.*

Proof. The proof is structured according to the four phases of the function $\mathcal{E}_{hyg}$.

Step 1.—The result of $\mathcal{T}[\![P]\!]\mathcal{S}_0$ is structurally equivalent to P; all identifiers have the 0 time-stamp. This claim can be verified by an induction on the structure of P.

Step 2.—Call the output of *Step 1* P_0. Then we can prove two statements about the relationship of $\mathcal{E}_{naive}[\![P]\!]\vartheta$ to $\mathcal{E}[\![P_0]\!]\vartheta'$.

1) The two results are equal modulo the time-stamps, *i.e.*, $\mathcal{E}_{naive}[\![P]\!]\vartheta = \mathcal{U}[\![\mathcal{E}[\![P_0]\!]\vartheta']\!]$. This proposition depends on the fact that ϑ' is induced by ϑ. It implies that the two results are structurally equivalent.

2) Moreover, all variables of a transcription step receive the same time-stamp which is unique with respect to the path from the root of the term to the occurrence of the respective syntactic extension). This follows from the fact that all transcriptions previous to the current one were time-stamped with a clock value of less than j—the current clock value. This statement is true for all syntactic extensions occurring in P_0. It is re-established by the time-stamping that immediately follows a transcription step. The variables in $(\vartheta' f)$ are either pure identifiers or tokens that already occurred in f. The pure ones are stamped with j and are thus distinguishable from all the previously generated identifiers. Afterwards the clock is advanced and all following expansions receive time-stamps at a higher level. As for applications, we know that syntactic extensions in the function and argument part cannot overlap. Hence, it is justified to continue the expansion process on both paths with the same clock value.

Step 3.—We know from the previous step that the result of $\mathcal{E}$ is structurally equivalent to the result of $\mathcal{E}_{naive}$ and that all identifiers of the hygienic result have a unique time-stamp reflecting their origin. Hence, if we α-convert all λ-expressions such that each time-stamped parameter is replaced by a fresh variable, the result satisfies the HC/ME and is also structurally equivalent to the input of $\mathcal{A}$. This can easily be verified by showing that $[v/w]z$ is a substitution function and that $\mathcal{A}$ otherwise preserves the structure.

Step 4.—The input to the last step is a term which satisfies the HC/ME and is structurally equivalent to the naïvely expanded program modulo time-stamps of free variables. It is a routine matter to prove by induction that the function $\mathcal{U}$ removes these time-stamps and leaves all other properties intact.

This concludes the proof. □

Implementation Note. From the above discussion and proof one can deduce an important fact about the implementation of the time-stamping scheme. Time-stamped variables have two essential properties. First, they are unique with respect to the rest of the program. Second, they must contain a component which indicates the original name. Hence, one can use gensym'd atoms with a property "original-name". When they turn out to be bound variables, they can simply stay in place. If they are free, they are replaced by the original name. The functions $\mathcal{A}$ and $\mathcal{U}$ have to be changed accordingly. **End of Note**

Now that we have a hygienic expansion algorithm, we can think about the implementation of exceptions to the HC/ME-rule. The exceptions which we have in mind should specify that certain "free identifiers" in a parameter to a transform function are captured by generated binding instances. The meaning of "free identifier," however, is not quite clear. To begin with, identifiers may occur in syntactic trees which are not expanded yet. Second, identifiers have time-stamps in the modified expander. We must ask whether we only want to consider identifiers with time-stamp 0—they are the user-supplied ones—or identifiers with all kinds of time-stamps.

The response to the first point is clear. If some identifier is to be captured, then it must be the one which survives as a free identifier until the input is completely expanded, no matter whether we can predict it or not. On the other hand, the second point cannot be resolved so easily. If we allowed capturing at all time-stamp levels, it would mean that there could be interaction of various syntactic extensions which are unpredictable. Since transform functions are all declared at the same level, *i.e.*, there is no scoping as among lexically scoped procedures,

the interactions cannot be deduced from static expressions. This would render the situation worse than before. We have therefore decided that macros may only capture user-supplied identifiers. The decision should be reconsidered when a macro system is being designed which allows for the modularization of syntactic extensions, for example, by blocks in a lexically scoped language.

We modify the hygiene condition to reflect our decision:

Modified Hygiene Condition for Macro Expansion. *Generated identifiers that become binding instances in the completely expanded program must only bind identifiers that are generated at the same transcription step or identifiers of the original user-input. (mHC/ME)*

The realization of this modified rule is simple. We provide the macro writer the function $\mathcal{S}_0$ which generates identifiers with a 0 time-stamp. If the transform function places these tokens in a binding position, they capture all the corresponding user-supplied identifiers. It is easy to see that $\mathcal{E}_{hyg}$ together with this variation satisfies the above theorem for the mHC/ME.

5. Adding More Lisp Constructs

Although the λ-calculus is a prototypical example of a programming language, it is by no means a real-world language. Compared to Lisp it is rather sparse. It lacks assignment statements, conditional expressions, and quoted structures. When we wish to add to these core forms, we extend the set *coretok* to include whatever symbols we choose to designate them. For example, *coretok* might become

$$\{(,), \textbf{lambda}, \textbf{set!}, \textbf{if}, \textbf{quote}\}.$$

Assignments and conditionals cause no problems at all because they are not binding constructs requiring special treatment by $\mathcal{E}$. Hence, they are treated like Lisp applications with a slightly more elaborate syntactic structure.

Quoted atoms or lists need special treatment. The expander can only recognize syntactic forms not occurring in the syntactic scope of any macro expression. Structures which seem to occur inside the syntactic scope of an extension may get rearranged during the expansion process, *e.g.*, the or-expression in Section 3. What appears to be a quoted structure because of the presence of the symbol 'quote' may not be an actual quoted structure. Thus its components must be time-stamped. However, when $\mathcal{E}$, $\mathcal{A}$, and $[\ /\ ]$ encounter an expression of the type **(quote** β**)**, they must inhibit the parsing process. The expression is a sentence of the target language and it is a constant expression. The respective additional lines in these functions are:

$$\mathcal{A}[\textbf{(quote } \beta\textbf{)}] = \textbf{(quote } \beta\textbf{)}$$
$$\mathcal{E}[\textbf{(quote } \beta\textbf{)}] = \textbf{(quote } \beta\textbf{)}$$
$$[v/w]\textbf{(quote } \beta\textbf{)} = \textbf{(quote } \beta\textbf{)}.$$

The time-stamps in β are ultimately removed when the unstamp function $\mathcal{U}$ is applied to the entire program.

6. Conclusion

The gains of the hygienic macro expander are clear. Macro writers can concentrate on the functional aspects of transform functions and need not worry about scope issues. Prohibiting the inadvertent capture of lexical identifiers has been an additional detail that the careful macro writer has had to remember. Furthermore, users find hygienic expansion more trustworthy. Careless macro writers will no longer surprise a user with unexpected bindings. While the user needs to know a semantics for the macro expressions, he should not need to know that a particular macro accomplishes its goal by binding certain local, temporary identifiers.

For those bindings the macro writer wishes to make public, the algorithm requires a change in conventional macro writing style. We expect the writer to inform the macro system of his decision as well as to document it for the user. In the past, the writer has been able to rely on the expansion algorithm to effect his desired bindings.

We have found that macros requiring capturing identifiers are rarer than those that introduce local binding identifiers. Thus we have shifted the expander's default behavior from possibly capturing all identifiers to only capturing those explicitly designated. In summary, we have given the macro writer less to worry about. And we have assured the macro user that any identifiers he puts in a macro expression will have the bindings he expects.

A transform function must satisfy two new conditions:

(1) time-stamped identifiers must be mapped to identifier names with the function $\mathcal{U}$ in a situation where the name of an identifier is needed by a transform function;

(2) identifiers in the output of a transform function must be unstamped, generated by $\mathcal{S}_0$, or time-stamp-equal to input identifiers.

The first refers to the situation in which a transform function takes some action that involves an actual identifier from the input. Such cases occur, for example, when different expansions are triggered by the presence of different identifiers in the user expression or when the transform function saves a piece of the input expression for some

purpose other than actual expansion. The second means that we cannot allow spurious identifiers generated by the transform function which appear stamped but were not produced by $\mathcal{S}_0$ or contained in the input expression. We must be able to recognize our own time-stamps. We have found that these restrictions on transform functions are usually satisfied or that it requires little effort to adapt existing transform functions.

In conclusion, we feel that hygienic expansion makes the writing and use of macros easier. It is safer than naïve expansion since the accidental capturing of identifiers that appear in user code cannot occur.

Acknowledgements

Mitch Wand provided invaluable help in the presentation of these ideas. Guy Steele pointed out a slight generalizaion to our original solution. Eugene Kohlbecker is an IBM Graduate Fellow. This material is based on work supported by the National Science Foundation under grants DCR 85-01277 and MCS 83-03325.

References

1. BARENDREGT, H. P. Introduction to the lambda calculus. *Nieuw Archief voor Wisenkunde 2* 4 (1984), 337–372.
2. BARENDREGT, H. P. *The Lambda Calculus: Its Syntax and Semantics Revised Edition.* North-Holland, Amsterdam, 1984.
3. CLINGER, W. D., (ED.). The revised revised report on Scheme. Joint Technical Report Indiana University and MIT Laboratory for Computer Science, 1985.
4. KLEENE, STEPHEN COLE. *Introduction to Metamathematics*, Van Nostrand, New York, 1952.
5. MCILROY, M. DOUGLAS. Macro instruction extensions of compiler languages. *CACM 3*, 4 (1960), 214–220.
6. STEELE, GUY L., JR. *Common Lisp: the Language.* Digital Press, 1984.
7. STEELE, GUY L., JR. AND GERALD J. SUSSMAN. The revised report on Scheme, a dialect of Lisp. Memo 452, MIT AI-Lab, 1978.

(Appendix begins on the next page.)

Appendix
An Implementation in Scheme

```
(define Ehyg
   (lambda (e)
      (lambda (theta)
         (U (A (((E ((T e) S-naught)) theta) 1))))))

(define T
   (lambda (t)
      (lambda (tau)
         (cond
            [(atomic-non-var? t) t]
            [(var? t) (tau t)]
            [else (map (lambda (t) ((T t) tau)) t)]))))

(define E
   (lambda (t)
      (lambda (theta)
         (lambda (j)
            (cond
               [(const? t) t]
               [(stamped? t) t]
               [(quote? t) t]
               [(macro? t)
                (((E ((T (theta t)) (S j))) theta)
                 (add1 j))]
               [(lambda? t)
                `(LAMBDA ,(var t)
                    ,(((E (body t)) theta) j))]
               [(app? t)
                `(,(((E (fun t)) theta) j)
                   ,(((E (arg t)) theta) j))])))))


(define A
   (lambda (t)
      (cond
         [(var? t) t]
         [(atomic-non-var? t) t]
         [(quote? t) t]
         [(lambda? t)
          (let ([v (gensym (U (var t))
                     ":" "new")])
             `(LAMBDA ,v
                 ,(A ((*/* v (var t))
                      (body t)))))]
         [(app? t)
          `(,(A (fun t))
             ,(A (arg t)))])))

(define U
   (lambda (t)
      (cond
         [(atomic-not-stamped? t) t]
         [(stamped? t)
          (get t 'original-name)]
         [else (map U t)])))
```

```
(define S
   (lambda (n)
      (let ([seen '()])
         (lambda (v)
            (let ([info (assq v seen)])
               (if info
                  (cdr info)
                  (let ([new (gensym v ":" n)])
                     (put new 'original-name v)
                     (set! seen
                        (cons
                           (cons v new) seen))
                     new)))))))


(define S-naught (S 0))

(define */*
   (lambda (v w)
      (lambda (t)
         (cond
            [(stamped? t) (if (eq? t w) v t)]
            [(atomic-not-stamped? t) t]
            [(quote? t) t]
            [(lambda? t)
             (if (eq? w (var t))
                 `(LAMBDA ,w ,(body t))
                 `(LAMBDA ,(var t)
                     ,((*/* v w)
                       (body t))))]
            [(app? t)
             `(,((*/* v w) (fun t))
                ,((*/* v w) (arg t)))]))))


(define stamped?
   (lambda (w)
      (and (symbol? w)
           (get w 'original-name))))

(define mactok?
   (lambda (m)
      (and (symbol? m)
           (get m 'mactok))))

(define coretok?
   (lambda (c)
      (and (symbol? c)
           (get c 'coretok))))

(define quote?
   (lambda (t)
      (and (pair? t)
           (eq? (car t) 'QUOTE)
           (pair? (cdr t))
           (null? (cddr t)))))
```

```
(define lambda?
   (lambda (t)
      (and (pair? t)
           (eq? 'LAMBDA (car t))
           (pair? (cdr t))
           (var? (cadr t))
           (pair? (cddr t))
           (null? (cdddr t)))))

(define app?
   (lambda (t)
      (and (pair? t)
           (pair? (cdr t))
           (null? (cddr t)))))

(define atomic-non-var?
   (lambda (y)
      (or (const? y)
          (stamped? y)
          (mactok? y)
          (coretok? y))))

(define atomic-not-stamped?
   (lambda (x)
      (or (const? x)
          (and (var? x)
               (not (stamped? x)))
          (mactok? x)
          (coretok? x))))

(define var? symbol?)
(define const? number?)
(define var cadr)
(define body caddr)
(define fun car)
(define arg cadr)


(put 'LAMBDA 'coretok 'true)
(put 'QUOTE 'coretok 'true)
(put 'LET 'mactok 'true)
(put 'IF 'mactok 'true)
(put 'OR 'mactok 'true)
(put 'NAIVE-OR 'mactok 'true)
(put 'FAKE 'mactok 'true)
(put 'CASE 'mactok 'true)

(define macro?
   (lambda (m)
      (record-case m
         [LET (var val body) true]
         [IF (a b c) true]
         [OR (a b) true]
         [NAIVE-OR (a b) true]
         [FAKE (x) true]
         [CASE (a b) true]
         [else false])))
```

```
(define ST
   (lambda (m)
      (record-case m
         [LET (i e b) `((LAMBDA ,i ,b) ,e)]
         [IF (a b c) `(((ef ,a) ,b) ,c)]
         [OR (a b) `(LET v ,a (IF v v ,b))]
         [NAIVE-OR (a b)
            (let ([v (S-naught 'v)])
               `(LET ,v ,a (IF ,v ,v ,b)))]
         [FAKE (x) `(QUOTE ,x)]
         [CASE (exp pair)
               `(LET v ,exp
                  (IF ((eq? v) (QUOTE ,(car pair)))
                      ,(cadr pair)
                      false))]
         [else (error "syntax table: no match" m)])))
```

——————— demonstration ———————

```
((Ehyg '(LET x (OR a v) (NAIVE-OR x v))) ST)

1 (LET x:0 (OR a:0 v:0) (NAIVE-OR x:0 v:0))
2 (NAIVE-OR x:0 v:0)
3 (LET v:0 x:0 (IF v:0 v:0 v:0))
4 (IF v:0 v:0 v:0)
4 (((ef:4 v:0) v:0) v:0)
3 ((LAMBDA v:0 (((ef:4 v:0) v:0) v:0)) x:0)
2 ((LAMBDA v:0 (((ef:4 v:0) v:0) v:0)) x:0)
2 (OR a:0 v:0)
3 (LET v:2 a:0 (IF v:2 v:2 v:0))
4 (IF v:2 v:2 v:0)
4 (((ef:4 v:2) v:2) v:0)
3 ((LAMBDA v:2 (((ef:4 v:2) v:2) v:0)) a:0)
2 ((LAMBDA v:2 (((ef:4 v:2) v:2) v:0)) a:0)
1 ((LAMBDA x:0
     ((LAMBDA v:0 (((ef:4 v:0) v:0) v:0)) x:0))
   ((LAMBDA v:2 (((ef:4 v:2) v:2) v:0)) a:0))

((LAMBDA x:new
    ((LAMBDA v:new (((ef v:new) v:new) v:new)) x:new))
 ((LAMBDA v:new (((ef v:new) v:new) v)) a))
```

```
((Ehyg '(LAMBDA a (CASE (FAKE a) (QUOTE a)))) ST)

1 (CASE (FAKE a:0) (QUOTE a:0))
2 (LET v:1 (FAKE a:0)
     (IF ((eq?:1 v:1) (QUOTE QUOTE)) a:0 false:1))
3 (IF ((eq?:1 v:1) (QUOTE QUOTE)) a:0 false:1)
3 (((ef:3 ((eq?:1 v:1) (QUOTE QUOTE))) a:0) false:1)
3 (FAKE a:0)
3 (QUOTE a:0)
2 ((LAMBDA v:1 (((ef:3 ((eq?:1 v:1) (QUOTE QUOTE))) a:0)
                false:1))
   (QUOTE a:0))
1 ((LAMBDA v:1 (((ef:3 ((eq?:1 v:1) (QUOTE QUOTE))) a:0)
                false:1))
   (QUOTE a:0))

(LAMBDA a:new
   ((LAMBDA v:new
       (((ef ((eq? v:new) (QUOTE QUOTE))) a:new) false))
    (QUOTE a)))
```