TR-H-171

Refining Hygienic Macros for Modules and Separate Compilation

Matthias Blume

1995. 8. 28
(1995.11.8受付)

ATR人間情報通信研究所

〒619-02　京都府相楽郡精華町光台2-2　☎ 0774-95-1011

ATR Human Information Processing Research Laboratories
2-2, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan
Telephone: +81-774-95-1011
Facsimile: +81-774-95-1008

Refining Hygienic Macros for Modules and Separate Compilation

Matthias Blume
Department of Computer Science, Princeton University
35 Olden Street, Princeton, NJ 08544
e-mail: `blume@cs.princeton.edu`

October 27, 1995

Abstract

Genuine differences in the treatment of identifiers in block-structured languages and those that provide qualified names for accessing components of modules or aggregate data structures invalidate some of the assumptions hygienic macro systems are based on. We will investigate how these assumptions have to be changed, and the consequences for the construction of hygienic macro expanders.

Macro expansion algorithms rely on their ability to rename identifiers throughout the program. This creates difficulties when some identifiers are used to connect individually compiled program units. Therefore, it is necessary to make separate compilation aware of macro expansion and vice versa. We will show how this can be done.

Keywords: hygienic macros, modules, separate compilation, Scheme

1 Introduction

Macro systems are part of many programming languages. The ability to abbreviate common idioms and to extend the syntax of the base language enjoys tremendous popularity. Furthermore, over the last few years most of the technical problems with macro systems have been solved. Kohlbecker's solution for *hygienic* macro expansion, which has been improved several times [BR88, Han91, CR91], provides the mechanism for implementing referentially transparent syntactic extensions in block-structured languages. Today most implementations of Scheme [IEE90] support hygienic macros in one or the other form, ranging from `syntax-rules` [Ce91] to more elaborate solutions like `syntax-case` [Dyb92].

The hygienic macro expander of Clinger and Rees [CR91] combines two different techniques: renaming and syntactic environments. Renaming alone would prevent inadvertent name clashes, but in order to make it work one must be able to distinguish between free and bound occurences of identifiers. This is difficult, because in order to determine the syntactic role (binding or applied occurence) of a name in the macro's output it can be necessary to expand other macro instances that did not yet exist and therefore couldn't have been analyzed at macro definition time. As a result the algorithm renames too many variables, and syntactic environments are used to hide the effects of such renamings.

Existing algorithms take advantage of the fact that identifiers in block-structured languages must always be interpreted with respect to a "current" syntactic environment. Therefore, they will cease to work properly when there are multiple environments, which happens when modules and qualified names are added to the language. Unfortunately, algorithms for macro expansion and interpretation of modules both use syntactic environments, and in different ways. With modules not every name is to be looked up in the "current" environment, because the **x** in **m.x** must be interpreted in the context of module **m**. But during macro expansion **x** could have been subjected to false renaming, so if macro expander and module system don't know about each other, then only the current environment but not the one associated with **m** will have the information necessary to hide "mistakes" of an over-eager renaming scheme.

The following trick might seem to provide a way out of the dilemma: Real implementations of hygienic macros often use *tagging* in order to represent renamed identifiers, and tagging can be undone without consulting any environment. Of course, the whole problem of hygienic macro expansion is to remove just the right tags, and that's where environments play their part. However, if we were able to restrict definitions in modules to untagged names only, then we could simply strip all tags from **x** in **m.x** before looking it up. Unfortunately, despite the fact that some existing systems do just that, it is still not a valid technique. Renaming is necessary to prevent bound identifiers from clashing. Definitions, no matter where they occur, whether in modules or at top level, must be considered binding constructs. Simply prohibiting tagged names here might solve one problem—but it comes at the expense of creating a new one.

It is generally undesirable for two language features to interact in ways such that they are not orthogonal and therefore cannot be used independently. Moreover, the combination of macros and modules is particularly useful for writing modules that can be parameterized by other modules. Consequently, the macro implementation should ensure hygiene throughout the entire language and not exclude the module system.

In this paper we present a rigorous solution to this problem. A refined version of Clinger and Rees' algorithm implements fully hygienic lexically scoped macros in languages that provide block-structure but utilize a non-flat name space.

Another problem, orthogonal to modules and qualified names, arises if we wish to provide means of separate compilation. In order to preserve hygiene the macro expander chooses fresh names for bound variables throughout the program. Furthermore, upon macro invocation it renames identifiers captured by that macro. This applies to both local and "external" bindings. Actual names chosen by the macro expander are arbitrary and will depend on its internal state. This is unfortunate, because we rely on externally bound names in order to perform program linking.

The second part of the paper shows a way of implementing a naming scheme for a compilation unit's imports and exports that is independent of arbitrary choices of the macro expander.

$I \in$ Ide	identifiers
$Q \in$ QIde	qualified identifiers
$T \in$ Tf	macro transformers
$M \in$ Mod	modules
$E \in$ Exp	expressions

Figure 1: Syntactic entities

```
Exp ⟶  Q | (E E*) | (lambda (I) E)
       | (let ((I E)) E)
       | (let-syntax ((I T)) E)
       | (let-module ((I M)) E)
```

Figure 2: Syntax of expressions

```
Mod ⟶  ()
       | (let ((I E)) M)
       | (let-syntax ((I T)) M)
       | (let-module ((I M)) M)
```

Figure 3: Syntax of modules

$i, j \in$	Ide	identifiers
q	$\mathit{QIde} = \mathit{Ide} + \mathit{QIde} \times \mathit{QIde}$	qualified identifiers
$t \in$	Tag	tags
$e \in$	Env	environments
$d \in$	$\mathit{Den} = \mathit{Var} + \mathit{Spec} + \mathit{Mac} + \mathit{Mod} + U$	
	$U = \{\mathit{unbound}\}$	error
		denotations
$v \in$	$\mathit{Var} = \mathit{QIde}$	variables
$s \in$	$\mathit{Spec} = \{\mathit{lambda},\ \mathit{let},\ \mathit{let\text{-}syntax},\ \mathit{let\text{-}module}\}$	keywords
	$\mathit{Mac} = R \times \mathit{Ide}^* \times \mathit{Env}$	macros
$r \in$	R	macro transform
	$\mathit{Mod} = \mathit{Env}$	modules

Figure 4: Domains

2 Macros and modules

2.1 The language

In this presentation we will consider a language that in addition to λ-abstractions offers block-structured definition of variables, macros, and modules.

Qualified identifiers are treated as if they were lexical units. This is unnecessary but makes it somewhat easier to present the algorithm. However, macro transformers can assemble qualified identifiers from other (simple or qualified) identifiers. Informally we use the notation m.x to denote a name x qualified by module name m.

For brevity we do not talk about a specific syntax for macro transformers, but only require that it must be possible to implement the macro compiler *compile* (figure 6). Our discussion applies equally well to both high-level macro languages based on pattern matching and low-level systems. In examples we use syntax-rules [Ce91].

Macro calls of the form (Q ...) can be used wherever Exp or Mod is required—as long as they expand into another Exp- or Mod-form, respectively.

2.2 The algorithm

We follow ideas in Rees [Ree93] by using a domain *Ide* (figure 4) that consists of symbols and tagged names. Only symbols can appear in the original program text. Tagged names are introduced by the macro expander. Tagging is simply a way of uniformly renaming all identifiers inserted by the same instance of a macro.

The environment constructors (figure 5) *ident* and *bind* are taken directly from Clinger and Rees' paper [CR91]. Their presentation avoids going into details of how identifiers are renamed. Consequently, they don't mention tags at all. Rees [Ree93] later described that idea separataly, and we make it explicit by introducing *tag* and *fork*. In addition, *qualify* and *empty* were added for the module system.

Every macro expansion produces a new expression and the corresponding environment for expansion. This environment combines the macro's environment of definition with the current environment. The former must be used for names introduced by the macro, while the latter takes care of all others. Since the same tag value is used in the *fork* node and in names inserted by the macro it is possible for *lookup* to properly distinguish between the two cases (figure 7).

If a macro is defined inside a module and refers to something in that module by an unqualified name, then upon expansion of the macro outside the module that name must be qualified. In the following example the program in the left should expand into that on the right:

```
(let-module
  ((mod (let ((x 1))
          (let-syntax                        (let-module
              ((mac (syntax-rules ()    ==>     ((mod (let ((x 1)) ())))
                      ((_) x))))                m.x)
            ()))))
  (m.mac))
```

The transcription of (m.mac) is an unqualified, but tagged version of x. Tagging correctly relates it to the environment where mac was defined, which was in module mod. Our algorithm uses *qualify* constructors to annotate such environments with the name of the module they belong to. Every time *lookup* (figure 7) encounters an environment constructed with *qualify* it must coerce the denotation obtained from the recursive invocation of lookup accordingly (figure 8). In our example it would prefix x with the module name m.

Let's assume y is bound in module m, while at the same time x, which is of the form $tag(t, \mathrm{y})$, is not bound there. By definition, x must have been renamed from y by some macro. Figure 11 shows an example of how this situation can arise. Since that macro didn't also insert a binding for m.x into its expansion—because contrary to our assumption it would mean x *is* bound in m—we must conclude that m.x should really be interpreted as m.y. This argument easily extends to more than one renaming step.

Fortunately, in the case of modules it is not difficult to undo such false renamings. But this must be done without help from the environment structure, because only the current environment has been augmented with *fork* nodes, which means unlike in the case of unqualified names we cannot depend on ordinary hygiene procedures.

$$
\begin{array}{rcl}
\mathit{tag} & \in & \mathit{Tag} \times \mathit{Ide} \to \mathit{Ide} \\
\mathit{empty}, \mathit{ident} & \in & \mathit{Env} \\
\mathit{bind} & \in & \mathit{Env} \times \mathit{Ide} \times \mathit{Den} \to \mathit{Env} \\
\mathit{fork} & \in & \mathit{Tag} \times \mathit{Env} \times \mathit{Env} \to \mathit{Env} \\
\mathit{qualify} & \in & \mathit{Env} \times \mathit{Var} \to \mathit{Env}
\end{array}
$$

Figure 5: Environment constructors

$$
\begin{array}{rcl}
\mathit{compile} & \in & \mathrm{Tf} \times \mathit{Env} \to \mathit{R} \times \mathit{Ide}^* \\
\mathit{coerce} & \in & \mathit{Den} \times \mathit{Var} \to \mathit{Den} \\
\mathit{lookup} & \in & \mathit{Env} \times \mathit{Ide} \to \mathit{Den} \\
\mathit{try} & \in & \mathit{Env} \times \mathit{Ide} \to \mathit{Den} \\
\mathit{qlookup} & \in & \mathit{Env} \times \mathit{QIde} \to \mathit{Den} \\
\mathit{transcribe}_E & \in & \mathrm{Exp} \times \mathit{R} \times \mathit{Ide}^* \times \mathit{Env} \to \mathrm{Exp} \\
\mathit{transcribe}_M & \in & \mathrm{Mod} \times \mathit{R} \times \mathit{Ide}^* \times \mathit{Env} \to \mathrm{Mod}
\end{array}
$$

Figure 6: Function signatures

$$
\begin{array}{rcl}
\mathit{lookup}(\mathit{ident}, i) & = & i \\
\mathit{lookup}(\mathit{empty}, i) & = & \mathit{unbound} \\
\mathit{lookup}(\mathit{bind}(e, i', d), i) & = & (i = i') \to d, \mathit{lookup}(e, i) \\
\mathit{lookup}(\mathit{fork}(r, e_d, e_u), i) & = & (i = \mathit{tag}(t, i')) \to \mathit{lookup}(e_d, i'), \mathit{lookup}(e_u, i) \\
\mathit{lookup}(\mathit{qualify}(e, v), i) & = & \mathit{coerce}(\mathit{lookup}(e, i), v)
\end{array}
$$

Figure 7: Equations for *lookup*

$$
\begin{array}{rcl}
\mathit{coerce}(v, v') & = & (v', v) \\
\mathit{coerce}(s, v) & = & s \\
\mathit{coerce}((r, i^*, e), v) & = & (r, i^*, \mathit{qualify}(e, v)) \\
\mathit{coerce}(e, v) & = & \mathit{qualify}(e, v) \\
\mathit{coerce}(\mathit{unbound}, v) & = & \mathit{unbound}
\end{array}
$$

Figure 8: Equations for *coerce*

$$
\frac{\mathit{lookup}(e, i) \neq \mathit{unbound}}{\mathit{try}(e, i) = \mathit{lookup}(e, i)} \quad [\mathit{success}]
$$

$$
\frac{\mathit{lookup}(e, \mathit{tag}(t, i)) = \mathit{unbound}}{\mathit{try}(e, \mathit{tag}(t, i)) = \mathit{try}(e, i)} \quad [\mathit{again}]
$$

Figure 9: Rules for *try*

$$
\begin{array}{rcl}
\mathit{qlookup}(e, i) & = & \mathit{lookup}(e, i) \\
\mathit{qlookup}(e, (q_1, (q_2, q_3))) & = & \mathit{qlookup}(e, ((q_1, q_2), q_3)) \\
\mathit{qlookup}(e, (q, i)) & = & \mathit{try}(\mathit{qlookup}(e, q), i)
\end{array}
$$

Figure 10: Equations for *qlookup*

```
(let-syntax
    ((mac (syntax-rules () ((_ m) m.x))))
  (let-module
      ((mod '(let ((x 1)) ())))
    (mac mod)))
```

Figure 11: Tagged qualified names

When $tag(t, \mathbf{y})$ is not bound in x's environment we make the guess that this is so because of false renaming and start another attempt—this time without the tag t. If it fails again we keep stripping tags from the identifier until we either find a binding or reach an untagged name (figure 9). Our discussion shows that upon success we have found the binding we are looking for, otherwise there simply isn't any, and `m.x` was not a legal construction.

The system of rewrite rules in figure 12 specifies the conditions under which an environment e justifies the "expands to" relation $\xrightarrow{E}$ for expressions. If E expands to E′ in e, then we write $e \vdash \mathrm{E} \xrightarrow{E} \mathrm{E}'$. Function *compile* implements the macro language. It returns the list of identifiers captured by the macro and a transformer program r, which then can be "executed" by either $transcribe_E$ or $transcribe_M$. For simplicity we have assumed that macros take exactly one argument, which can be any symbolic expression.

Rule $[mod_E]$ refers to another kind of "expands to" relation. The notation $e \vdash (\mathrm{M}, e_m) \xrightarrow{M} (\mathrm{M}', e'_m)$ not only tells us what a module M expands into, but at the same time also calculates the module's environment. Figure 13 shows the set of rules for relation $\xrightarrow{M}$. Not surprisingly, it is very similar to the one defining $\xrightarrow{E}$ with the main difference being some extra effort spent constructing the module's environment.

3 Separate compilation

In our model a program is nothing more than a big unnamed module, which simply means it consists of a number of nested definitions. Therefore, we will use Mod as our syntax for programs. A program can be broken into an ordered sequence of smaller programs—compilation units.

The objective of compilation is to produce some form of directly executable code. Following Appel and MacQueen [AM94] we translate each compilation unit into a closed λ-expression. All free variables of the original module appear as explicit arguments, and the return value is a data structure containing the values for the unit's variable definitions. Since the expression has no free variables it can then be compiled without need to reference any outside environment.

It is the linker's responsibility to fetch correct values for the arguments of the expression from the "global" environment and to later augment it with new bindings. In order to be able to do so the linker must be provided with information about imports and exports.

$$\frac{\mathit{qlookup}(e, \mathbf{q}) = \mathbf{v}}{e \vdash \mathbf{q} \xrightarrow{E} \mathbf{v}} \qquad [\mathit{var}_R]$$

$$\frac{\begin{array}{c} e \vdash \mathbf{E}_0 \xrightarrow{E} \mathbf{E}_0' \\ \forall i \in \{1, \ldots, k\}, k \geq 0 : e \vdash \mathbf{E}_i \xrightarrow{E} \mathbf{E}_i' \end{array}}{e \vdash \texttt{(}\mathbf{E}_0\ \mathbf{E}_1\ \ldots\ \mathbf{E}_k\texttt{)} \xrightarrow{E} \texttt{(}\mathbf{E}_0'\ \mathbf{E}_1'\ \ldots\ \mathbf{E}_k'\texttt{)}} \qquad [\mathit{app}_E]$$

$$\frac{\begin{array}{c} \mathit{qlookup}(e, \mathbf{k}_0) = \mathit{lambda} \\ \mathit{bind}(e, \mathbf{x}, \mathbf{x}') \vdash E \xrightarrow{E} E' \\ \mathbf{x}' \text{ is a fresh identifier} \end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{(}\mathbf{x}\texttt{)}\ \mathbf{E}\texttt{)} \xrightarrow{E} \texttt{(lambda (}\mathbf{x}'\texttt{)}\ \mathbf{E}'\texttt{)}} \qquad [\lambda_E]$$

$$\frac{\begin{array}{c} \mathit{qlookup}(e, \mathbf{k}) = (r, \langle i, \ldots \rangle, e_d) \\ t \text{ is a fresh tag} \\ \mathit{transcribe}_E(\texttt{(}\mathbf{k}\ \mathbf{B}\texttt{)}, r, \langle \mathit{tag}(t, i), \ldots \rangle, \mathit{fork}(t, e_d, e)) = \mathbf{E} \\ \mathit{fork}(t, e_d, e) \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}' \end{array}}{e \vdash \texttt{(}\mathbf{k}\ \mathbf{B}\texttt{)} \xrightarrow{E} \mathbf{E}'} \qquad [\mathit{mac}_E]$$

$$\frac{\begin{array}{c} \mathit{qlookup}(e, \mathbf{k}_0) = \mathit{let} \\ e \vdash \mathbf{E}_1 \xrightarrow{E} \mathbf{E}_1' \\ \mathbf{x}' \text{ is a fresh identifier} \\ \mathit{bind}(e, \mathbf{x}, \mathbf{x}') \vdash \mathbf{E}_2 \xrightarrow{E} \mathbf{E}_2' \end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{((}\mathbf{x}\ \mathbf{E}_1\texttt{))}\ \mathbf{E}_2\texttt{)} \xrightarrow{E} \texttt{(let ((}\mathbf{x}'\ \mathbf{E}_1'\texttt{))}\ \mathbf{E}_2'\texttt{)}} \qquad [\mathit{let}_E]$$

$$\frac{\begin{array}{c} \mathit{qlookup}(e, \mathbf{k}_0) = \mathit{let\text{-}syntax} \\ \mathit{compile}(\mathbf{T}, e) = (r, i^*) \\ \mathit{bind}(e, \mathbf{x}, (r, i^*, e)) \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}' \end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{((}\mathbf{x}\ \mathbf{T}\texttt{))}\ \mathbf{E}\texttt{)} \xrightarrow{E} \mathbf{E}'} \qquad [\mathit{syn}_E]$$

$$\frac{\begin{array}{c} \mathit{qlookup}(e, \mathbf{k}_0) = \mathit{let\text{-}module} \\ e \vdash (\mathbf{M}, \mathit{empty}) \xrightarrow{M} (\mathbf{M}', e_m) \\ \mathbf{x}' \text{ is a fresh identifier} \\ \mathit{bind}(e, \mathbf{x}, \mathit{qualify}(e_m, \mathbf{x}')) \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}' \end{array}}{e \vdash \texttt{(}\mathbf{k}_0\ \texttt{((}\mathbf{x}\ \mathbf{M}\texttt{))}\ \mathbf{E}\texttt{)} \xrightarrow{E} \texttt{(let-module ((}\mathbf{x}'\ \mathbf{M}'\texttt{))}\ \mathbf{E}'\texttt{)}} \qquad [\mathit{mod}_E]$$

Figure 12: Rewrite rules for Exp

$$e \vdash (\texttt{()}, e_m) \xrightarrow{M} (\texttt{()}, e_m) \qquad [nul_M]$$

$$\frac{\begin{gathered} glookup(e, \mathrm{k}) = (r, \langle i, \ldots \rangle, e_d) \\ t \text{ is a fresh tag} \\ transcribe_M(\texttt{(k B)}, r, \langle tag(t, i), \ldots \rangle, fork(t, e_d, e)) = \mathrm{M} \\ fork(t, e_d, e) \vdash (\mathrm{M}, e_m) \xrightarrow{M} (\mathrm{M}', e'_m) \end{gathered}}{e \vdash (\texttt{(k B)}, e_m) \xrightarrow{M} (\mathrm{M}', e'_m)} \qquad [mac_M]$$

$$\frac{\begin{gathered} glookup(e, \mathrm{k}_0) = let \\ e \vdash \mathbf{E} \xrightarrow{E} \mathbf{E}' \\ \mathrm{x}' \text{ is a fresh identifier} \\ bind(e, \mathrm{x}, \mathrm{x}') \vdash (\mathrm{M}, bind(e_m, \mathrm{x}, \mathrm{x}')) \xrightarrow{M} (\mathrm{M}', e'_m) \end{gathered}}{e \vdash (\texttt{(k}_0\ \texttt{((x E)) M)}, e_m) \xrightarrow{M} (\texttt{((let ((x}'\ \texttt{E}'\texttt{)) M}'\texttt{)}, e'_m)} \qquad [let_M]$$

$$\frac{\begin{gathered} glookup(e, \mathrm{k}_0) = let\text{-}syntax \\ compile(\mathrm{T}, e) = (r, i^*) \\ bind(e, \mathrm{x}, (r, i^*, e)) \vdash (\mathrm{M}, bind(e_m, \mathrm{x}, (r, i^*, e))) \xrightarrow{M} (\mathrm{M}', e'_m) \end{gathered}}{e \vdash (\texttt{(k}_0\ \texttt{((x T)) M)}, e_m) \xrightarrow{E} (\mathrm{M}', e'_m)} \qquad [syn_M]$$

$$\frac{\begin{gathered} glookup(e, \mathrm{k}_0) = let\text{-}module \\ e \vdash (\mathrm{M}_1, empty) \xrightarrow{M} (\mathrm{M}'_1, e_{m,1}) \\ \mathrm{x}' \text{ is a fresh identifier} \\ qualify(e_{m,1}, \mathrm{x}') = e^q_{m,1} \\ bind(e, \mathrm{x}, e^q_{m,1}) \vdash (\mathrm{M}_2, bind(e_m, \mathrm{x}, e^q_{m,1})) \xrightarrow{M} (\mathrm{M}'_2, e'_m) \end{gathered}}{e \vdash (\texttt{(k}_0\ \texttt{((x M}_1\texttt{)) M}_2\texttt{)}, e_m) \xrightarrow{M} (\texttt{((let-module ((x}'\ \texttt{M}'_1\texttt{)) M}'_2\texttt{)}, e'_m)} \qquad [mod_M]$$

Figure 13: Rewrite rules for Mod

3.1 Augmenting the global environment

Every compilation unit must be compiled with respect to a current global environment. The domain of global environments is different from *Env*, because it binds names to "global" denotation. Variable bindings, for example, are no longer represented by qualified names but directly associate identifiers with values.

Nevertheless, the overall structure of the global environment is very similar to *Env*. Therefore, there is a relatively straightforward way of producing export information from a compilation unit:

1. Expand unit **U** in the current global environment: $e_g \vdash (\mathbf{U}, \mathit{empty}) \xrightarrow{M} (\mathbf{U}', e_u)$.

2. Traverse e_u and assign numbers to bindings of variables. This has to be done recursively, so that variables in modules are numbered as well.

3. Construct an expression that at runtime will evaluate into a vector v of the so-numbered values. Value i must be stored at $v[i]$. The expression itself is then to replace the innermost () of the compilation unit.

4. Convert e_u to a Δ-environment e_u^Δ. Δ-environments encode incremental changes to the global environment. They only use persistent data and avoid cycles as well as sharing. Δ-environments are suitable for storage on stable media such as files in the ambient file system.

5. When **U**'s code has successfully executed we can instantiate e_u^Δ using the current global environment and the vector of export values obtained from running the code.

The following problems must be addressed when creating e_u^Δ from e_u:

- Cycles and sharing in e_u are eliminated by representing back- and cross edges using some marking technique. (In our case there can never be cycles, but introducing recursive definitions would change this situation.)

- Variable bindings are represented by numbers referring to slots of the export vector.

- Newly introduced tags and labels must be encoded in such a way that the instantiation of e_u^Δ can later be labeled and tagged equivalently. Within Δ-environments we rely on small numbers instead of tags and labels using some isomorphic mapping from tags and labels to an initial segment of the natural numbers.

- There are names in the environment structure that have been imported from the global environment while **U** was expanded. In order to be able to re-establish this situation when instantiating e_u^Δ we use *recipes*, which are the same kind of persistent identifiers that are also used as part of the import mechanism. This technique will be described in section 3.2.

- All remaining names have been created by tagging other, existing identifiers. The tags correspond to tags at *fork* nodes in e_u. Therefore, they are also encoded by mapping them to small integers.

The conversion can be implemented by a conventional recursive traversal of the data structure that implements e_u. In our case it is directed and acyclic, but in general—with recursion—it can be an arbitrary directed graph.

The inverse process of instantiating a Δ-environment with respect to some vector of values and a current global environment is then straightforward.

- Cycles and sharing are re-created according to the marking information in e_u^Δ.
- Variables are bound to values. Those values are fetched from the result vector using the slot numbers that are stored in the Δ-environment.
- For each integer in e_u^Δ that stands for some tag or label we create a fresh tag or label.
- Recipes are instantiated in the current global environment. This operation is the same that is also used for the import mechanism (section 3.2).
- All other names are created by tagging existing ones according to the integer-to-label map mentioned above.

3.2 Imports

Hygienic macros complicate the problem of specifying how an import value is to be located, because identifiers can be names generated by the macro system. Furthermore, since environments contain fork points it may happen that a name must be looked up in some macro's environment of definition and not in the current global one. To deal with this we need to be able to specify generated names without mentioning actual tags and to redirect the lookup operation to other environments.

Let's assume we had a persistent (tag-free) naming scheme for identifiers. If a name is not to be looked up in the current global environment, then it must have been inserted by some global macro. In that case it is necessary to find the macro's environment of definition, which, of course, is nothing less than another instance of the import problem itself.

Fortunately, there is always an end to this process, because otherwise there would have been infinitely many expansion steps in the original source program. Therefore, we can refer to any global environment using a sequence of macro names, which have to be looked up in order, starting in the global environment. This sequence is called a path, and the global environment itself is represented by the empty path. Imports can now be specified using a path for the environment and another persistent name for the identifer to be looked up.

The only missing piece is a method for specifying identifers without mentioning actual tags. There are three kinds of names:

1. Plain symbols, which are not tagged at all.
2. Names tagged while expanding U.
3. Names that have been tagged before U was expanded.

Rec	$= \mathit{Symbol} + \mathit{Mc} + \mathit{Untag}$	recipes
Mc	$= \mathit{Imp} \times \mathit{Integer}$	name capture
Untag	$= \mathit{Rec}$	untag
Path	$= \mathit{Rec}^*$	paths
Imp	$= \mathit{Path} \times \mathit{Rec}$	imports

Figure 14: Recipes, paths, and imports

$$\begin{array}{lll} \mathit{extern} & \in & \mathit{GEnv} \times \mathit{Path} \to \mathit{Env} \\ \mathit{gcoerce} & \in & \mathit{GDen} \times \mathit{Path} \times \mathit{Rec} \to \mathit{Den} \end{array}$$

Figure 15: *extern* and *gcoerce*

Plain names are easy to deal with, because they can represent themselves. The second kind of name can never occur in a global lookup operation. First of all, tags created while expanding **U** do not occur in the global environment, so every global lookup operation of such an identifier is bound to fail. Furthermore, a name tagged in such a way will always be interpreted in an environment that contains the corresponding *fork* nodes, and those cancel new tags before the lookup operation "reaches" the external environment.

Thus, only the third kind of name is of interest to us. How can those names find their way into compilation unit **U**? Without non-hygienic macros there is only one possibility: They must have been inserted by a global macro. Luckily, we know exactly which names a macro can insert, because function *compile* computes this information from a transformer. Therefore, it is possible to "name" such an identifier by specifying the macro it was inserted by together with the position number in the macro's list of captured names. Naming the macro is yet another instance of the import problem, which we have already discussed above.

The module system raises a few more minor issues. First of all, a global lookup can take place with respect to a module's environment. Also, the *try* function might eliminate tags from names before they are fed into the lookup procedure. Therefore, our persistent naming scheme must offer a way of expressing this. Finally, if we allowed low-level non-hygenic macros, then we would also need a way of specifying identifiers that were created non-hygienically.

Together this results in mutually recursive definitions for imports, paths, and recipes. Recipes, which are used to name identifiers, can be one of the following:

- a symbol
- an import specifying some macro together with a position number adressing one of the names captured by that macro
- a recipe plus an "untag" directive, which indicates that the last tag is to be removed

Figure 14 shows the relevant domain equations. Recipes and imports are prescriptions for re-enacting some of the same operations the macro expander would perform if we would let it expand the original source of **U** again. However, these operations are "distilled" to focus efforts on just those names that are important for external linkage.

3.3 Compiling one unit

Global environments are similar, but not identical to ordinary environments. To capture this idea we will use a new domain *GEnv*. Obviously, the rules in figure 12 and 13 cannot be used directly with elements of *GEnv*. Therefore, we introduce *extern* as a constructor that takes elements of *GEnv* to element of *Env*.

Besides the technical issue of dealing with two different domains this also has a practical advantage by providing a way of detecting free variables in the compilation unit. It is convenient to annotate *extern* nodes with the path that is the persistent name for the global environment in question. The coercion function *gcoerce*, which is analogous to *coerce* in the case of module environments, takes global denotations (*GDen*) to elements of *Den*(figure 15).

One important part of *gcoerce*'s task is to detect free variables of the program. For each free variable it calculates the import specification, invents a new identifier, and remember the relationship between the two. The fresh name is returned from *lookup* in form of a *Var* denotation. In the end we can close over the fully expanded program by wrapping it in *lambda* abstractions. The list of imports tells the linker, which values have to be fetched from the global environment, so they can be passed to the executable code.

How do we calculate recipes? The trick is to do this on the fly, during macro expansion itself. Elements of domain *Ide* are treated as an abstract datatype. Internally it consists of two parts: the actual name and the current recipe. Every time the macro expander constructs a new identifier it also remembers how this was done. To make that work we must adjust the definition of recipes, so we can express the fact that a name carries a newly generated tag. Such new tags will ultimately go away before the name becomes subject to global lookup, but in the meantime it is necessary to keep track of them. All operations that create names must also compute the corresponding recipe. Thus, we pay a constant penalty per such operation. Still, the asymptotic complexity of the algorithm stays the same.

4 Prototype implementation

We have implemented an experimental macro expander for a language similar to Scheme. The expander is based on techniques explained in this paper and was written in Standard ML [MTH90]. Experiments were conducted using SML/NJ [AM91].

Our prototype language provides low-level macro transformers with an escape from hygiene, recursive definitions, and modules. Furthermore, it also supports Scheme's mutable variables. The implementation not only performs macro expansion, but also eliminates mutable variables by introducing explicit reference cells where necessary.

The program can be considered the front end of a compiler. It translates source-language expressions into a macro-free intermediate representation similar to the λ-language that is used in various other compilers for Scheme or Standard ML. A simple interpreter executes the expanded code. In a realistic system it would be replaced by an optimizing back end.

The algorithms are asymptotically as fast as their precursors, and experiments confirm that they perform well in practice. We expect to be able to combine our solution with existing compilers for Scheme and Scheme-like languages. So far we have not investigated the problem of integrating macros with parametric modules or strong type checking, both of which are useful features found in languages such as Standard ML.

References

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[AM94] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Proc. SIGPLAN '94 Symp. on Prog. Language Design and Implementation*, pages 13–23. ACM Press, June 1994.

[BR88] Alan Bawden and Jonathan Rees. Syntactic Closures. In *1988 ACM Conference on Lisp and Functional Programming*, pages 86–95, 1988.

[Ce91] William Clinger and Jonathan Rees (editors). Revised^4 Report on the Algorithmic Language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[CR91] William Clinger and Jonathan Rees. Macros That Work. In *Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 155–162, 1991.

[Dyb92] R. Kent Dybvig. Writing Hygienic Macros in Scheme with Syntax-Case. Technical Report 356, Indiana University Computer Science Department, June 1992.

[Han91] Chris Hanson. A Syntactic Closures Macro Facility. *LISP Pointers*, IV(4):9–16, December 1991.

[IEE90] IEEE Standard 1178-1990: IEEE Standard for the Scheme Programming Language, 1990.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Ree93] Jonathan Rees. The Scheme of Things: Implementing Lexically Scoped Macros. *LISP Pointers*, VI(1):33–37, January-March 1993.