A Theory of Hygienic Macros

David Herman and Mitchell Wand

College of Computer and Information Science
Northeastern University
Boston, MA 02115
{dherman,wand}@ccs.neu.edu

Abstract. Hygienic macro systems automatically rename variables to prevent unintentional variable capture—in short, they "just work." But hygiene has never been presented in a formal way, as a *specification* rather than an algorithm. According to folklore, the definition of hygienic macro expansion hinges on the preservation of alpha-equivalence. But the only known definition of alpha-equivalence for Scheme depends on the results of macro expansion! We break this circularity by introducing binding specifications for macros, permitting a definition of alpha-equivalence independent of expansion. We define a semantics for a first-order subset of Scheme macros and prove hygiene as a consequence of confluence.

The subject of macro hygiene is not at all decided, and more research is needed to precisely state what hygiene formally means and [precisely which] assurances it provides.

—Oleg Kiselyov [1]

1 What are Hygienic Macros?

Programming languages with hygienic macros automatically rename variables to prevent subtle but common bugs arising from unintentional variable capture—the experience of the practical programmer is that hygienic macros "just work." Numerous macro expansion algorithms for Scheme have been developed over many years [2–6], and the Scheme standard has included hygienic macros since R^4RS [7].

Yet to date, a formal specification for hygiene has been an elusive goal. Intuitively, macro researchers have always understood hygiene to mean *preserving α-equivalence*. In particular, performing an α-conversion of a bound variable should not result in a macro expansion that accidentally captures the renamed variable. But this idea has never been made precise.

Why should such a simple idea be so hard to formalize? The problem is this: since the only known binding forms in Scheme are the core forms, the binding structure of a Scheme expression does not become apparent until after it has been fully expanded to core Scheme. Thus α-equivalence is only well-defined for Scheme programs that have been fully expanded, with no remaining instances of macros. So if the conventional wisdom is correct, the definition of hygienic macro

expansion relies on α-equivalence, but the definition of α-equivalence relies on the results of macro expansion! This circularity is clearly paradoxical, and the definition of hygiene has consequently remained a mystery.

But observe that in practice, well-behaved macros follow regular binding disciplines consistently, independent of their particular expansion. For example, despite the fact that the `let` form can be defined in Scheme as a macro, programmers rely on knowing the binding structure of `let` without actually thinking about its expansion into a primitive `lambda` form. If the semantics of macros only had access to this binding structure in such a way that we could reason formally about the scope of Scheme programs without resorting to operational reasoning about their expansion, we could cut the Gordian knot and specify both α-equivalence and hygiene in an intuitive and precise way.

To put it more succinctly, we argue that *the binding structure of a macro is a part of its interface*. In this paper, we make that interface explicit as a type annotation. Our type system is novel but incorporates ideas both from the shape types of Culpepper and Felleisen [8] and nominal datatypes of Gabbay and Pitts [9]. With the aid of these type annotations, we generalize Scheme's α-equivalence relation to include programs with macros and prove hygiene as a consequence of confluence.

The organization of this paper is as follows. The next section introduces λ_m, a Scheme-like language with typed macros. Section 3 defines the α-equivalence relation for λ_m, and Section 4 introduces the macro type system. Section 5 defines the macro expansion semantics. The next two sections present the key correctness theorems: type soundness in Section 6 and hygiene in Section 7. In Section 8 we present a front end for parsing S-expressions as λ_m expressions. Section 9 concludes with a discussion of related and future work.

2 λ_m: an Intermediate Language for Modeling Macros

Macro expansion transforms Scheme S-expressions into a small, fixed set of core forms which the underlying compiler or interpreter is designed to recognize. Expansion eliminates uses of macros by translating them according to their definitions, repeating this process recursively until there are no derived forms left to translate. Thus the macro expansion process consumes programs in surface syntax:

```
(let ((x (sqrt 2)))
  (let ((y (exp x)))
    (lambda (f)
      (f y))))
```

and produces programs containing only the internal forms recognized by the compiler:

$$((\lambda x.\,((\lambda y.\,\lambda f.\,f\ y)\ (exp\ x)))\ (sqrt\ 2))$$

We use a distinct syntax for core forms to highlight the fact that they indicate the completion of macro expansion. Because macro expansion operates on partially

expanded programs, which may contain both core forms and S-expressions yet to be expanded, a model for macros must incorporate both syntactic elements.

To that end, we define an intermediate language for modeling macro expansion, called λ_m. The core forms of λ_m are based on the λ-calculus, but we add two additional forms to support macros: one for local binding of macro definitions, and one for macro application.[1]

$$
\begin{array}{lll}
e & ::= & v \mid \lambda v.\, e \mid e\ e \mid \texttt{let syntax}\ x\ \texttt{=}\ m\ \texttt{in}\ e\ \texttt{end} \mid op[\![s]\!]^{\sigma} \\
v & ::= & x \mid \texttt{?}a \\
op & ::= & v \mid m \\
m & ::= & \texttt{macro}\ p : \sigma \Rightarrow e \\
p & ::= & \texttt{?}a \mid \texttt{(}\overline{p}\texttt{)} \\
s & ::= & e \mid op \mid \texttt{(}\overline{s}\texttt{)}
\end{array}
$$

Unlike the surface syntax of ordinary Scheme, which consists exclusively of undifferentiated S-expressions, we make the syntactic roles of terms in this calculus explicit. The syntax is defined not just in terms of S-expressions but expressions e, whose syntactic structure is fixed and manifest. Of course, macros admit arbitrary syntactic extension in the form of S-expressions, so S-expressions s appear in the grammar as the arguments to macro applications. Here too, though, the syntactic structure is made apparent via a *shape type annotation* σ. We return in detail to shape types in Section 2.2. Variables v come in two sorts: *program variables* x, which are standard, and *pattern variables* $\texttt{?}a$, which are used in macro patterns and templates to bind macro arguments. Macro operators op are either variable references or macro expressions. Macros m contain a pattern p, a type annotation σ, and a template expression e. A pattern p is a tree of pattern variables (assumed not to contain duplicates). Finally, an S-expression s is a tree of expressions or macro operators. The latter form is used to pass macros as arguments to other macros.

The syntax of λ_m may seem unfamiliar compared to the simple S-expressions of Scheme. After all, Scheme applications $\texttt{(}\overline{s}\texttt{)}$ look different from λ_m applications $op[\![s]\!]^{\sigma}$ and in Scheme, pattern variables are indistinguishable from program variables. However, we can parse a surface syntax of S-expressions into λ_m easily enough, and we describe this parsing algorithm in Section 8.

2.1 Tree Locations

In order to address context-sensitive properties of terms, we use the mechanism of *tree locations* [10] to select and distinguish subterms by their position. Tree structures in our language take the general form $t ::= L \mid \texttt{(}\overline{t}\texttt{)}$ for some nonterminal of leaves L. For any such tree structure, we can select a subtree as a path from the root of the tree to the node containing the subtree. A *tree location* ℓ is an element of $\mathbb{N}^*$. Given a tree t, the subtree $t.\ell$ is defined by $t.\epsilon = t$ and $\texttt{(}\overline{t}\texttt{)}.i\,\ell = t_i.\ell$.

[1] Throughout this paper we use an overbar notation ($\overline{x}$) to represent sequences.

2.2 Binding Specifications

Macro definitions and applications in λ_m are explicitly annotated with shape types. The purpose of these annotations is to fix the structure of macros, including their scoping structure. For example, the following macro m matches four pattern variables, $\texttt{?}a, \texttt{?}b, \texttt{?}e_1$, and $\texttt{?}e_2$:

$$\begin{array}{l}\texttt{macro (?}a\ \texttt{?}b\ \texttt{?}e_1\ \texttt{?}e_2\texttt{) : (}\langle 0\rangle\ \langle 1\rangle\ \textbf{expr}^0\ \textbf{expr}^{0,1}\texttt{)} \\ \quad \Rightarrow \lambda \texttt{?}a.\,((\lambda \texttt{?}b.\,\texttt{?}e_2)\ \texttt{?}e_1)\end{array}$$

The shape type $\sigma = \texttt{(}\langle 0\rangle\ \langle 1\rangle\ \textbf{expr}^0\ \textbf{expr}^{0,1}\texttt{)}$ tells us that pattern variables $\texttt{?}a$ and $\texttt{?}b$ are placed in binding positions in the macro template, pattern variable $\texttt{?}e_1$ is used in the scope of $\texttt{?}a$ alone, and $\texttt{?}e_1$ appears inside the scope of both $\texttt{?}a$ and $\texttt{?}b$. Maintaining the bindings in order—$\texttt{?}a$ is bound outside, $\texttt{?}b$ inside—makes it possible to resolve references unambiguously even if both $\texttt{?}a$ and $\texttt{?}b$ are instantiated with the same variable. For example, this tells us that $m[\![\texttt{(}x\ x\ x\ x\texttt{)}]\!]^\sigma =_\alpha m[\![\texttt{(}x\ y\ y\ x\texttt{)}]\!]^\sigma$ but $m[\![\texttt{(}x\ x\ x\ x\texttt{)}]\!]^\sigma \neq_\alpha m[\![\texttt{(}x\ y\ x\ y\texttt{)}]\!]^\sigma$.

Shape types are defined by the following grammar:

$$\begin{array}{l}\tau ::= \textbf{expr} \mid \sigma \to \textbf{expr} \\ \beta ::= \langle \ell \rangle \mid \textbf{expr}^{\ell,\overline{\ell}} \\ \sigma ::= \tau \mid \beta \mid \texttt{(}\overline{\sigma}\texttt{)}\end{array}$$

The base types τ include the type of expressions and the types of macros, which receive S-expressions as arguments and produce expressions. Binding types β express the scope of S-expressions. A binder type $\langle \ell \rangle$ corresponds to a variable in binding position. The location ℓ represents the position in the macro S-expression where the binder occurs. A body type $\textbf{expr}^{\ell,\overline{\ell}}$ corresponds to an expression inside the scope of one or more binders; the locations $\overline{\ell}$ indicate the positions in the macro S-expression of each of the binders that are in scope, in the order in which they are bound, outermost first.

2.3 From S-Expressions to the Lambda Calculus

Once a λ_m program has been fully expanded, it consists only of core forms, which in our simple model corresponds to the untyped λ-calculus. We say a program is in *expansion-normal form* (ENF) if it obeys the familiar grammar:

$$e ::= x \mid \lambda x.\,e \mid e\ e$$

If ENF is the internal language of the compiler or evaluator, then S-expressions are the surface language used by the programmer. The syntax of the surface language is a restricted subset of λ_m S-expressions:

$$s ::= x \mid \texttt{(}\overline{s}\texttt{)}$$

Thus we can envision an idealized pipeline for the evaluation of Scheme programs as shown in Figure 1.

Fig. 1. Pipeline for an idealized Scheme evaluator.

This is different from real Scheme implementations, since Scheme's macros do not have a type system to allow parsing to be performed up front. Rather, parsing is traditionally interleaved with macro expansion as the syntactic roles of expressions gradually become apparent. Nonetheless, with complete type information, it becomes possible to parse an S-expression before macro expansion. We return to the front end in Section 8.

3 Alpha-Equivalence

We follow Gabbay and Pitts [9] in using variable swapping to define α-equivalence. Swapping is defined by:

$$\begin{array}{lll}
(v_1\ v_2)\cdot v_1 & = v_2 & \\
(v_1\ v_2)\cdot v_2 & = v_1 & \\
(v_1\ v_2)\cdot v & = v & \text{if } v \notin \{v_1, v_2\} \\
(v_1\ v_2)\cdot \lambda v.\, e & = \lambda((v_1\ v_2)\cdot v).\,((v_1\ v_2)\cdot e) & \\
(v_1\ v_2)\cdot (\!|\overline{s}|\!) & = (\!|\overline{(v_1\ v_2)\cdot s}|\!) & \\
\textit{etc.} & &
\end{array}$$

The *support* of a term is the set of variables it contains:

$$\begin{array}{ll}
\mathit{supp}(v) & = \{v\} \\
\mathit{supp}(\lambda v.\, e) & = \{v\} \cup \mathit{supp}(e) \\
\mathit{supp}(\,(\!|\overline{s}|\!)\,) & = \bigcup_i \mathit{supp}(s_i) \\
\textit{etc.} &
\end{array}$$

A variable v is fresh with respect to a finite set of terms S, written $v \# S$, if for all terms $s \in S$, $v \notin \mathit{supp}(s)$. We write $v \# s_1, \ldots, s_n$ where $n \geq 1$ to mean $v \# \{s_1, \ldots, s_n\}$.

We also define the notion of simultaneously introducing multiple, distinct fresh variables by overloading the freshness relation for variable mappings. If S is a set of terms and Z is a mapping $\{\overline{\ell \mapsto z}\}$ then we write $Z \# S$ to mean

$$\forall \ell \in \mathit{dom}(Z)\,.\, Z(\ell) \# S \text{ and } \forall \ell, \ell' \in \mathit{dom}(Z)\,.\, Z(\ell) = Z(\ell') \Rightarrow \ell = \ell'$$

We identify the binders of a form by collecting the set of binding positions identified in the form's shape type. The function $bp(\sigma)$ produces the set of binding positions of a shape type, and the function $pp(p)$ identifies the positions of pattern variables in a macro pattern.

$$\begin{array}{llll}
bp(\,(\!|\overline{\sigma}|\!)\,) & = \bigcup_i \{i\,\ell \mid \ell \in bp(\sigma_i)\} & pp(\,(\!|\overline{p}|\!)\,) & = \bigcup_i \{i\,\ell \mid \ell \in pp(p_i)\} \\
bp(\langle \ell \rangle) & = \{\epsilon\} & pp(?a) & = \{\epsilon\} \\
bp(\mathbf{expr}^{\overline{\ell}}) & = bp(\tau) = \emptyset & &
\end{array}$$

We can use bp to compute the set of binders of a macro application $binders(\sigma, s)$ as a mapping from binding positions ℓ to their actual binders $s.\ell$:

$$binders(\sigma, s) = \{\ell \mapsto s.\ell \mid \ell \in bp(\sigma)\}$$

3.1 Shape-Directed Conversion

Consider the following Scheme expression, with all occurrences of the variable `x` labelled for the sake of explanation.

```
(let ((x¹ x²))
  (x³ (lambda (x⁴) x⁵)))
```

In order to α-convert $\texttt{x}^1$ to a fresh name `z`, we must be careful to rename only the occurrences of `x` bound by $\texttt{x}^1$, which in this example includes only $\texttt{x}^3$. Because macros may have arbitrary shape, a structural induction on the S-expression would be insufficient to recognize which instances of `x` were which. Instead, we define a notion of *shape-directed conversion* $(Z\ X)^\sigma \cdot s$, which follows the structure of a form's binding specification rather than its syntax.

$$\begin{array}{lll}
(Z\ X)^\tau \cdot s = s & & \\
(Z\ X)^{\langle \ell \rangle} \cdot x = z & & \text{if } z = Z(\ell) \\
(Z\ X)^{\langle \ell \rangle} \cdot v = v & & \text{if } \ell \notin dom(Z) \\
(Z\ X)^{\textsf{expr}^{\ell, \overline{\ell'}}} \cdot e & = (z\ x) \cdot (Z\ X)^{\textsf{expr}^{\overline{\ell'}}} \cdot e & \text{if } z = Z(\ell) \text{ and } x = X(\ell) \\
(Z\ X)^{\textsf{expr}^{\ell, \overline{\ell'}}} \cdot e & = (Z\ X)^{\textsf{expr}^{\overline{\ell'}}} \cdot e & \text{if } \ell \notin dom(Z) \\
(Z\ X)^{(\overline{\sigma})} \cdot (\overline{s}) & = (\overline{(Z\ X)^{\sigma_i} \cdot s_i}) &
\end{array}$$

The key to the definition of shape-directed conversion is the fourth rule, which swaps a bound variable with its corresponding fresh name in an expression within its scope. Because body types order their bound variables from the outside in, occurrences of the variable x are renamed to z only after performing all inner renamings, in case x is shadowed by an inner binding.

3.2 Alpha-Equivalence

The definition of α-equivalence appears in Figure 2. The first four rules parallel the rules of α-equivalence for the λ-calculus, but note that we do not convert pattern variables $?a$ used in binding positions. The rule for macro bindings converts the macro name and proceeds inductively. The next rule is key: to compare two macro applications, their operators must be equivalent, and their arguments must be equivalent once we α-convert their bound variables. Checking these involves several conditions. First, the two expressions must bind exactly the same pattern variables, if any; we ensure this by requiring that at any binding position ℓ, $s.\ell$ binds an ordinary program variable x if and only if $s'.\ell$ binds an ordinary program variable x'. We collect the binder mappings X and X' for the two respective forms, and we choose a mapping of fresh binders Z, being careful

$$\frac{}{v =_\alpha v} \qquad \frac{e =_\alpha e'}{\lambda ?a.\, e =_\alpha \lambda ?a.\, e'} \qquad \frac{z \# e, e' \quad (z\ x) \cdot e =_\alpha (z\ x') \cdot e'}{\lambda x.\, e =_\alpha \lambda x'.\, e'} \qquad \frac{e_1 =_\alpha e_1' \quad e_2 =_\alpha e_2'}{e_1\ e_2 =_\alpha e_1'\ e_2'}$$

$$\frac{\begin{array}{c} z \# e, m, e', m' \\ m =_\alpha m' \qquad (z\ x) \cdot e =_\alpha (z\ x') \cdot e' \end{array}}{\texttt{let syntax}\ x\ \texttt{=}\ m\ \texttt{in}\ e\ \texttt{end} =_\alpha \texttt{let syntax}\ x'\ \texttt{=}\ m'\ \texttt{in}\ e'\ \texttt{end}}$$

$$\frac{\begin{array}{c} op =_\alpha op' \\ \forall \ell \in bp(\sigma)\,.\, \exists x = s.\ell \Leftrightarrow \exists x' = s'.\ell \\ X = binders(\sigma, s) \qquad X' = binders(\sigma, s') \\ Z = \{\ell \mapsto z \mid \ell \in bp(\sigma), \exists\, x = s.\ell\} \qquad Z \# s, s' \\ (Z\ X)^\sigma \cdot s =_\alpha (Z\ X')^\sigma \cdot s' \end{array}}{op[\![s]\!]^\sigma =_\alpha op'[\![s']\!]^\sigma}$$

$$\frac{\begin{array}{c} \forall \ell \in pp(p)\,.\, p.\ell = ?a_\ell \text{ and } p'.\ell = ?a'_\ell \text{ and } ?z_\ell \# e, e' \\ \forall \ell, \ell' \in pp(p)\,.\, ?z_\ell = ?z_{\ell'} \Rightarrow \ell = \ell' \\ (\overline{?z_\ell\ ?a_\ell}) \cdot p = (\overline{?z_\ell\ ?a'_\ell}) \cdot p' \\ (\overline{?z_\ell\ ?a_\ell}) \cdot e =_\alpha (\overline{?z_\ell\ ?a'_\ell}) \cdot e' \end{array}}{(\texttt{macro}\ p : \sigma \Rightarrow e) =_\alpha (\texttt{macro}\ p' : \sigma \Rightarrow e')} \qquad \frac{\forall i\,.\, s_i =_\alpha s'_i}{(\overline{s}) =_\alpha (\overline{s'})}$$

Fig. 2. Alpha-equivalence of λ_m programs.

not to α-convert at locations that bind pattern variables. Finally, we compare the α-converted arguments s and s'. The rule for comparing macros is somewhat simpler. We choose fresh pattern variables $?z_\ell$ to replace the pattern variables in either macro, and compare both their patterns and templates. Finally, compound S-expressions are compared inductively.

3.3 Instantiation

Identifying binders in a shape type positionally is convenient for the theory, since it results in one canonical representation for each distinct type. However, for some operations it is necessary to identify binders by name. We present an alternate form of shape types $\hat{\sigma}$ which use variables rather than locations to represent their binding structure:

$$\begin{array}{l} \hat{\beta} ::= \langle v \rangle \mid \mathbf{expr}^{v,\overline{v}} \\ \hat{\sigma} ::= \tau \mid \hat{\beta} \mid (\overline{\hat{\sigma}}) \end{array}$$

We write $\hat{\sigma} = \sigma[X]$ to denote the instantiation of a nameless shape type σ with the concrete variable names of X.

The free and bound variables of an expression are computed via shape-directed generalizations of the standard operations $FV(s, \hat{\sigma})$ and $BV(s, \hat{\sigma})$ (omitted for space). The following theorem ensures that we can always replace an S-expression with an α-equivalent S-expression with fresh binders.

Theorem 1 (Freshness). *Let s be an S-expression and S be a finite set of S-expressions. Then there exists an S-expression $s' =_\alpha s$ such that $BV(s', \hat{\sigma}) \# S$.*

Proof. Induction on the structure of s. For each binding in s, choose fresh binders that are not in $supp(S)$.

The next theorem is easily proved and assures us that λ_m α-equivalence generalizes the standard α-equivalence relation ($\equiv$) for the λ-calculus.

Theorem 2. *If e and e' are in ENF, then $e =_\alpha e'$ iff $e \equiv e'$.*

4 Type Checking

The job of the type checker is to confirm that each macro definition conforms to its specification and that each use of a macro conforms to its interface. Excerpts of the type checking algorithm are presented in Figure 3. The type system uses two environments: the *program environment* $\Gamma ::= \bullet \mid \Gamma[v := \tau]$, which tracks the bindings of variables in the program, and the *pattern environment* $\Phi \in \{\bullet\} \cup PVar \rightarrow Shape$, which is used in checking the body of a macro to map the pattern variables to their annotated shape types. This environment is constructed by pairing the structure of a macro pattern p with an instantiation of the macro's type annotation:

$$penv(\textbf{(}\overline{p}\textbf{)}, \textbf{(}\overline{\hat{\sigma}}\textbf{)}) = \bigcup_i penv(p_i, \hat{\sigma}_i)$$
$$penv(?a, \hat{\sigma}) = \{?a \mapsto \hat{\sigma}\}$$

We now describe the more subtle type rules. Rule [T-MacApp] checks the argument with the annotated type instantiated with the actual binders. Rule [T-PBody] checks a pattern variable reference with a body type, ensuring that all the necessary pattern variables have been bound in the proper order. Rule [T-Body] binds a variable from a body type in the program environment. Rule [T-Macro] constructs a pattern environment Φ and checks the template, masking out any pattern variables from the program environment; the first-order macros of λ_m cannot refer to pattern variables outside their own scope.

4.1 The Aliasing Problem

The design of our type system led us to discover a peculiarity of Scheme macros. Consider the following macro:

```
(define-syntax K
  (syntax-rules ()
    ((K a b)
     (lambda (a)
       (lambda (b) a)))))
```

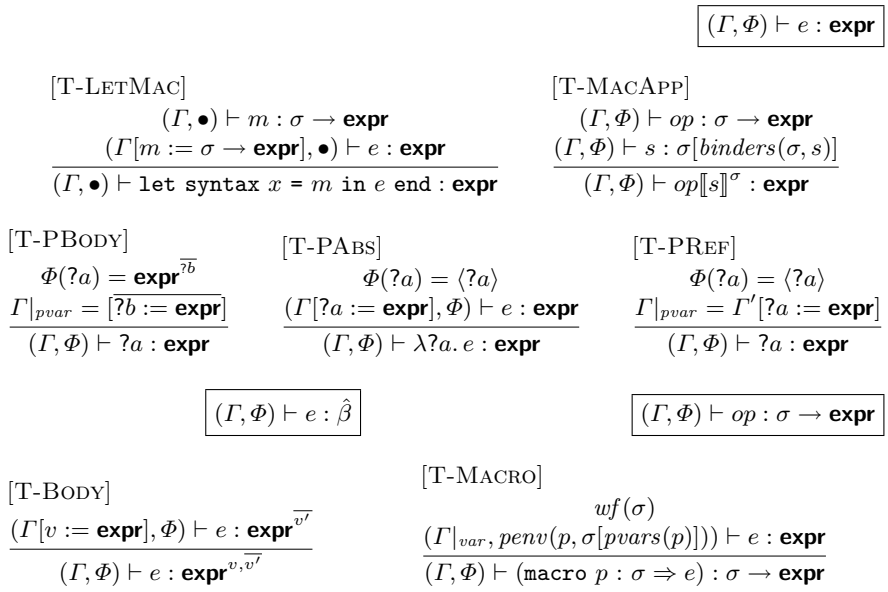


Fig. 3. Excerpts from the λ_m type system.

One might expect that any application of `K` would produce an expression equivalent to $\lambda x.\, \lambda y.\, x$. But consider the application `(K x x)`: even in a hygienic macro system, this would expand into $\lambda x.\, \lambda x.\, x$! The binding structure of `K` is thus dependent on its actual arguments. We call this dependency the *aliasing problem*.

To resolve this ambiguity, we propose a simple rule we call the *shadow restriction*. The type rule [T-PRef] only allows bound references to pattern binders when no other intervening pattern binders are in scope. This restriction might seem draconian, but in fact the above macro can easily be rewritten:

```
(define-syntax K′
  (syntax-rules ()
    ((K′ a b)
     (lambda (a)
       (let ((tmp a))
         (lambda (b) tmp))))))
```

Note that even with standard, untyped Scheme macros, this new definition always exhibits the intended behavior, in that even `(K′ x x)` expands into an expression equivalent to $\lambda x.\, \lambda y.\, x$.

4.2 Alpha-Equivalence Preserves Type

Theorem 3 gives us the freedom to use α-equivalent S-expressions without affecting the types.

Lemma 1. $(\Gamma, \Phi) \vdash s : \sigma[X] \Leftrightarrow (\Gamma, \Phi) \vdash (Z\ X)^{\sigma} \cdot s : \sigma[Z]$

Theorem 3 (Alpha-equivalence preserves type). *If* $(\Gamma, \Phi) \vdash s : \hat{\sigma}$ *and* $s =_{\alpha} s'$ *then* $(\Gamma, \Phi) \vdash s' : \hat{\sigma}$.

5 Macro Expansion

In this section, we specify our macro expansion semantics. We begin with a notion of compatibility, defined via expansion contexts.

5.1 Expansion Contexts

An expansion context C^{σ} is an S-expression with a hole $[\,]$, which produces an S-expression of shape σ when filled with an expression e. When the shape of a context is clear or irrelevant, we omit it for brevity.

$$
\begin{array}{lll}
C^{\mathsf{expr}^{\overline{\ell}}} & ::= [\,] \mid \lambda v.\, C^{\mathsf{expr}} \mid C^{\mathsf{expr}}\ e \mid e\ C^{\mathsf{expr}} & \\
& \mid \texttt{let syntax}\ x\ \texttt{=}\ C^{\sigma \to \mathsf{expr}}\ \texttt{in}\ e\ \texttt{end} & \\
& \mid \texttt{let syntax}\ x\ \texttt{=}\ m\ \texttt{in}\ C^{\mathsf{expr}}\ \texttt{end} & \\
& \mid C^{\sigma \to \mathsf{expr}}(s) \mid op[\![C^{\sigma}]\!]^{\sigma} & \\
C^{(\overline{\sigma})} & ::= (\overline{s}^{1..i-1}\ C^{\sigma_i}\ \overline{s}^{i+1..|\overline{\sigma}|}) & i \in 1..|\overline{\sigma}| \\
C^{\sigma \to \mathsf{expr}} & ::= \texttt{macro}\ p : \sigma \Rightarrow C^{\mathsf{expr}} &
\end{array}
$$

5.2 Variable Conventions

The heart of hygienic macro expansion is the management of bindings to prevent accidental capture. Different expansion algorithms achieve this in different ways. For the *specification* of hygienic macro expansion, we simply specify the necessary conditions on variables under which expansion can proceed.

Analogous to the Barendregt variable convention [11], the condition *transparent* allows a macro definition to be substituted into an application only if no intervening bindings can capture free variable references in the macro template. This condition is sometimes referred to in the community as *referential transparency*.

$$transparent(s, \hat{\sigma}, s', \hat{\sigma}') \Leftrightarrow BV(s, \hat{\sigma}) \cap FV(s', \hat{\sigma}') = \emptyset$$

This condition alone is not enough to prevent unintended capture. The *hygienic* variable convention requires that a macro template's bindings be fresh before performing an application. This prevents the bindings in the template from capturing references in the macro's arguments.

$$hygienic(s, \hat{\sigma}, s') \Leftrightarrow BV(s, \hat{\sigma}) \# s'$$

5.3 Expansion Semantics

The semantics of macro expansion involves two rules. The first rule connects macro applications to their definitions via the substitution operation $s[x := m]^{\hat{\sigma}}$, which uses the shape type $\hat{\sigma}$ to traverse the (binding) structure of s.

$$\begin{array}{lll}
v[x := m]^{\mathbf{expr}} & = v & (v \neq x) \\
(\lambda x.\, e)[x := m]^{\mathbf{expr}} & = \lambda x.\, e & \\
(\lambda y.\, e)[x := m]^{\mathbf{expr}} & = \lambda y.\, (e[x := m]^{\mathbf{expr}}) & (v \neq x) \\
x[x := m]^{\sigma \to \mathbf{expr}} & = m & \\
v[x := m]^{\sigma \to \mathbf{expr}} & = v & (v \neq x) \\
e[x := m]^{\mathbf{expr}^{x, \overline{v}}} & = e & \\
e[x := m]^{\mathbf{expr}^{v, \overline{v'}}} & = e[x := m]^{\mathbf{expr}^{\overline{v'}}} & (v \neq x) \\
\textit{etc.} & &
\end{array}$$

A macro substitution step is defined by the rule:

$$\begin{array}{l}
\texttt{let syntax}\ x\ \texttt{=}\ m\ \texttt{in}\ e\ \texttt{end} \longmapsto_{\mathsf{subst}} e[x := m]^{\mathbf{expr}} \\
\quad \text{if } \mathit{transparent}(e, \mathbf{expr}, m, \mathit{type}(m))
\end{array}$$

Note that the variable convention must be fulfilled to prevent the context of the macro application from capturing free variable references in the macro template.

The second rule of macro expansion performs a macro transcription step, expanding an individual macro application. This rule happens in two parts. The first part, *pattern matching*, matches the macro pattern against the actual subexpressions, producing a substitution ρ:

$$\begin{array}{ll}
\mathit{match}(\texttt{(}\overline{p}\texttt{)}, \texttt{(}\overline{s}\texttt{)}) & = \bigcup_i \mathit{match}(p_i, s_i) \\
\mathit{match}(\texttt{?}a, s) & = \{\texttt{?}a \mapsto s\}
\end{array}$$

Next, *transcription* instantiates all pattern variables in the template with the substitution function ρ:

$$\begin{array}{ll}
\mathit{transcribe}(x, \rho) & = x \\
\mathit{transcribe}(\texttt{?}a, \rho) & = \rho(\texttt{?}a) \\
\mathit{transcribe}(\lambda v.\, e, \rho) & = \lambda(\mathit{transcribe}(v, \rho)).\, (\mathit{transcribe}(e, \rho)) \\
\mathit{transcribe}(e_1\ e_2, \rho) & = (\mathit{transcribe}(e_1, \rho))\ (\mathit{transcribe}(e_2, \rho)) \\
\mathit{transcribe}(op[\![s]\!]^{\sigma}, \rho) & = (\mathit{transcribe}(op, \rho))[\![\mathit{transcribe}(s, \rho)]\!]^{\sigma} \\
\mathit{transcribe}(m, \rho) & = m \\
\mathit{transcribe}(\texttt{(}\overline{s}\texttt{)}, \rho) & = \texttt{(}\overline{(\mathit{transcribe}(s, \rho))}\texttt{)}
\end{array}$$

The macro transcription step is defined as the rule:

$$\begin{array}{l}
(\texttt{macro}\ p : \sigma \Rightarrow e)[\![s]\!]^{\sigma} \longmapsto_{\mathsf{trans}} \mathit{transcribe}(e, \mathit{match}(p, s)) \\
\quad \text{if } \mathit{transparent}(s, \hat{\sigma}, e, \mathbf{expr}) \text{ and } \mathit{hygienic}(e, \mathbf{expr}, s) \\
\quad \text{where } \hat{\sigma} = \sigma[\mathit{binders}(\sigma, s)]
\end{array}$$

The first variable convention also applies to this rule, since binders introduced in the actual arguments of the macro application should not capture free references

from the template. The second convention prevents binders introduced from the body of the template from capturing references in the actual arguments.

We define the binary relation $\longmapsto_\varepsilon$ to be the compatible closure of the combined rules $\longmapsto_{\mathsf{subst}} \cup \longmapsto_{\mathsf{trans}}$ on S-expressions up to α-equivalence, i.e., the least relation such that $s_1 \longmapsto_\varepsilon s_2$ if there exist S-expressions s_1', s_2', a context C, and expressions e_1, e_2 such that $s_1 =_\alpha s_1'$, $s_2 =_\alpha s_2'$, $s_1' = C[e_1]$, $s_2' = C[e_2]$, and either $e_1 \longmapsto_{\mathsf{subst}} e_2$ or $e_1 \longmapsto_{\mathsf{trans}} e_2$. The binary relation $\longmapsto\!\!\!\!\rightarrow_\varepsilon$ is the reflexive, transitive closure of $\longmapsto_\varepsilon$.

6 Type Soundness

The type soundness proof is in the style of Wright and Felleisen [12]. The Preservation Lemma is proved for any S-expression s; it is reused in this more general form for the proof of confluence.

Lemma 2 (Preservation). *If $(\Gamma, \Phi) \vdash s : \hat{\sigma}$ and $s \longmapsto_\varepsilon s'$ then $(\Gamma, \Phi) \vdash s' : \hat{\sigma}$.*

Proof. The proof depends on three lemmas that guarantee that macro substitution, pattern matching, and transcription respectively preserve type, as well as a decomposition lemma. Theorem 3 ensures that choosing α-equivalent terms to satisfy the variable conventions is also type-preserving.

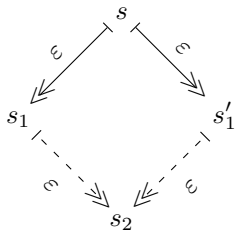
Lemma 3 (Progress). *If $\vdash e :$ **expr** then either e is in ENF or there exists an e' such that $e \longmapsto_\varepsilon e'$.*

Proof. Macro substitution is defined for all well-typed S-expressions, as is *match*. Theorem 1 allows us to choose α-equivalent terms that satisfy the variable conventions for the expansion rules.

Theorem 4 (Type soundness). *If $\vdash e :$ **expr** and $e \longmapsto\!\!\!\!\rightarrow_\varepsilon e'$ and $e' \not\longmapsto_\varepsilon$, then e' is in ENF and $\vdash e' :$ **expr**.*

7 Hygiene

Theorem 5 (Confluence). *Let s be an S-expression such that $(\Gamma, \Phi) \vdash s : \hat{\sigma}$.*



Proof. In the style of Barendregt [11], Chapter 11, §1. The proof involves marking a redex and tracking the marked redex and any copies or expansions of that marked term through multiple expansion steps. The central lemma shows that both macro substitution and transcription commute with expansion of marked redexes.

At last, the final Hygiene Theorem follows immediately from confluence.

Theorem 6 (Hygiene). *Let e_0 be an expression such that $\vdash e_0 : \textbf{expr}$. If $e_0 =_\alpha e_0'$, $e_0 \longmapsto\!\!\!\!\!\rightarrow_\varepsilon e$, and $e_0' \longmapsto\!\!\!\!\!\rightarrow_\varepsilon e'$ such that e and e' are in ENF, then $e =_\alpha e'$.*

This theorem provides the crucial guarantee of hygienic macros, namely that α-conversion of λ_m programs is semantics-preserving.

8 Front End

The parsing algorithm uses the same environments as the type system in order to distinguish the sorts of variables as well as annotate macro applications with types. Excerpts of this parsing algorithm are presented in Figure 4. Because function application in Scheme is denoted by parenthesization rather than invoking a special application macro, the rule for parsing function applications inserts an explicit reference to a built-in macro `@`. This is similar to the technique used in PLT Scheme [13], in which implicit function applications are rewritten to explicit applications of `#%app`.

Scheme implementations generally provide a standard library of macros. The primitive forms `lambda` and `@` can be implemented as macros in the initial context of a Scheme program:

$C_0 =$ `let syntax`
`lambda` $= (\texttt{macro } \texttt{((?}a\texttt{)} \ \texttt{?}e\texttt{)} : \texttt{((}\langle 00\rangle\texttt{)} \ \textbf{expr}^{00}\texttt{)} \Rightarrow \lambda \texttt{?}a.\, \texttt{?}e)$
`@` $= (\texttt{macro } \texttt{(?}e_1 \ \texttt{?}e_2\texttt{)} : \textbf{(expr expr)} \Rightarrow \texttt{?}e_1 \ \texttt{?}e_2)$
`in` [ ] `end`

Parsing accounts for these macros in the type environment:

$$\begin{aligned} \Gamma_0(\texttt{lambda}) &= \texttt{((}\langle 00\rangle\texttt{)} \ \textbf{expr}^{00}\texttt{)} \rightarrow \textbf{expr} \\ \Gamma_0(\texttt{@}) &= \textbf{(expr expr)} \rightarrow \textbf{expr} \end{aligned}$$

9 Related and Future Work

Hygienic macros are over twenty years old, and many macro systems have been designed to facilitate or guarantee hygiene [2, 5, 3, 6]. Several have been defined in a rigorous and formal way, but none provides a specification for hygiene, nor any satisfying account for the guarantees it provides. Our work shares a common observation with the *syntactic closures* macro system [4], namely that macro programmers know the binding structure of macros *a priori*; their work provides an API rather than a theory. The primitive macros `lambda` and `@` resemble the *micros* of Krishnamurthi [14].

Several syntactic extension mechanisms have been designed for languages other than Scheme [15, 16]. MacroML [17] is particularly relevant since it automatically prevents unintended variable capture. Their system is restrictive: binding forms can only extend ML's `let` form, and macros cannot inspect or destructure their syntactic arguments. Our work allows destructuring of S-expressions

$$
\begin{array}{l}
parse(\Gamma, \Phi, x, \mathbf{expr}) \qquad\qquad = \begin{cases} \texttt{?}x \text{ if } x \in dom(\Phi) \\ x \quad \text{if } x \notin dom(\Phi) \end{cases} \\
parse(\Gamma, \Phi, \texttt{(let-syntax ((}x\ s_1\texttt{))}\ s_2\texttt{)}, \mathbf{expr}) = \texttt{let syntax } x \texttt{ = } m \texttt{ in } e \texttt{ end} \\
\quad \text{where } parseMacro(s_1) = m \\
\quad\quad \text{and } parse(\Gamma[x := type(m)], \Phi, s_2, \mathbf{expr}) = e \\
parse(\Gamma, \Phi, \texttt{(}x\ \overline{s}\texttt{)}, \mathbf{expr}) \qquad\qquad = op[\![s']\!]^{\sigma} \\
\quad \text{where } parseOperator(\Gamma, \Phi, x) = (op, \sigma \rightarrow \mathbf{expr}) \\
\quad\quad \text{and } binders(\Phi, \sigma, \texttt{(}\overline{s}\texttt{)}) = X \\
\quad\quad \text{and } parse(\Gamma, \Phi, \texttt{(}\overline{s}\texttt{)}, \sigma[X]) = s' \\
parse(\Gamma, \Phi, \texttt{(}s_1\ s_2\texttt{)}, \mathbf{expr}) \qquad\qquad = parse(\Gamma, \Phi, \texttt{(@ }s_1\ s_2\texttt{)}, \mathbf{expr}) \\
\quad \text{if } s_1 \notin dom(\Gamma) \text{ and } \texttt{?}s_1 \notin dom(\Phi)
\end{array}
$$

Fig. 4. Excerpts of the type-directed parsing algorithm.

while still preserving the integrity of expressions. Our work also provides a theory of α-equivalence. Previous work on *staged notational definitions* [18] provides a meta-language SND for reasoning about MacroML programs; we believe our system more closely matches the informal reasoning used by macro programmers.

The shape types of Culpepper and Felleisen [8] are similar in expressive power to ours, allowing destructuring of S-expressions and synthesis of arbitrary binding forms. Our work extends theirs by accounting for binding structures. Crucially, this provides us with our account of α-equivalence and hygiene. Our use of types for expressing bindings was inspired by the nominal datatypes of Gabbay and Pitts [9].

Gasbichler [19] and Bove and Arbilla [20] both provide formal accounts for the semantics of macros. Gasbichler's formalism explores a much richer and more detailed macro language in order to study the interaction of hygienic macros and module systems, whereas our work is concerned with the guarantees provided by hygiene. Bove and Arbilla study the interaction between macro expansion and program evaluation. Both systems employ explicit substitutions and de Bruijn indices rather than renaming. We have taken the approach of α-conversion in order to explore the connection between hygiene and α-equivalence. Bove and Arbilla provide a proof of confluence, but in the context of a language with only top-level macro definitions, i.e., without lexically scoped macros.

Finally, we note that the design of our shape types bears some resemblance to the *locally nameless* approach to binding structures [21–23]. In particular, our macro types use tree locations ℓ in order to avoid using an α-equivalence relation on shape types, but when destructuring a type, we instantiate these locations with concrete names. We intend to investigate this relationship further.

There is much more to discover of the theory of hygienic macros. Our elementary type system is not yet expressive enough to permit important idioms in common use, including recursive macros, variable-length lists and list-patterns [24], and case dispatch. Another important next step will be to understand the type structure of *higher-order macros*, which expand into subsequent macro defini-

tions. We intend to investigate the connection to staged types for this question. Other areas for future exploration include procedural macros, inference for shape types, and support for intentional capture.

References

1. Kiselyov, O.: How to write seemingly unhygienic and referentially opaque macros with syntax-rules. In: Scheme Workshop. (2002)
2. Kohlbecker, E., Friedman, D.P., Felleisen, M., Duba, B.: Hygienic macro expansion. In: LISP and Functional Programming. (1986)
3. Clinger, W., Rees, J.: Macros that work. In: Principles of Programming Languages. (1991)
4. Bawden, A., Rees, J.: Syntactic closures. In: LISP and Functional Programming. (1988) 86–95
5. R. Kent Dybvig, R.H., Bruggeman, C.: Syntactic abstraction in Scheme. Lisp and Symbolic Computation **5**(4) (December 1993) 295–326
6. van Tonder, A.: SRFI 72: Hygienic macros. Online (September 2005)
7. Clinger, W., Rees, J.: Revised^4 report on the algorithmic language Scheme. Technical report (1991)
8. Culpepper, R., Felleisen, M.: Taming macros. In: GPCE. (2004)
9. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Formal Aspects of Computing **13**(3–5) (2001) 341–363
10. Gorn, S.: Explicit definitions and linguistic dominoes. In: Systems and Computer Science, Proceedings of the Conference held at Univ. of Western Ontario. (1967)
11. Barendregt, H.P.: The Lambda Calculus: Its Syntax and Semantics. revised edn. North-Holland, Amsterdam (1984)
12. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (1994) 38–94
13. Flatt, M.: PLT MzScheme: Language manual. Technical Report PLT-TR2007-1-v371, PLT Scheme Inc. (2007) http://www.plt-scheme.org/techreports/.
14. Krishnamurthi, S.: Linguistic Reuse. PhD thesis, Rice University (May 2001)
15. de Rauglaudre, D.: Camlp4 reference manual. Online (September 2003)
16. Sheard, T., Peyton Jones, S.: Template metaprogramming for Haskell. In Chakravarty, M.M.T., ed.: Haskell Workshop. (2002) 1–16
17. Ganz, S.E., Sabry, A., Taha, W.: Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. In: ICFP 2001. (2001) 74–85
18. Taha, W., Johann, P.: Staged notational definitions. In: GPCE. (2003) 97–116
19. Gasbichler, M.: Fully-parameterized, first-class modules with hygienic macros. PhD thesis, University of Tübingen (August 2006)
20. Bove, A., Arbilla, L.: A confluent calculus of macro expansion and evaluation. In: LISP and Functional Programming, ACM Press (June 1992) 278–287
21. McKinna, J., Pollack, R.: Some lambda calculus and type theory formalized. Journal of Automated Reasoning **23** (1999) 373–409
22. Gordon, A.D.: A mechanisation of name-carrying syntax up to alpha-conversion. In: Workshop on Higher Order Logic Theorem Proving and its Applications. (1994) 413–425
23. McBride, C., McKinna, J.: Functional pearl: I am not a number—I am a free variable. In: Haskell Workshop. (2004) 1–9
24. Kohlbecker, E.E., Wand, M.: Macro-by-example: Deriving syntactic transformations from their specifications. In: Principles of Programming Languages. (1987)