Hygienic Quasiquotation in Scheme
Distilled Tutorial

Morten Rhiger
Roskilde University, Denmark
mir@ruc.dk

Abstract
Quasiquotation in Scheme is nearly ideal for implementing programs that generate other programs. These programs lack only the ability to generate fresh bound identifiers, as required to make such code-manipulating programs hygienic, but any Scheme programmer knows how to provide this ability using gensym.

In this tutorial we investigate hygienic quasiquotation in Scheme and in languages influenced by Scheme. Stepping back from implementation issues, we first identify the source of the freshness condition in the semantics of a hygienic quasiquotation facility. We then show how gensym is needed to break a meta-circularity in interpreters and compilers for hygienic quasiquotation. Finally, following our recent work, we present a type system for hygienic quasiquotation that supports evaluation under dynamic λ-abstraction, manipulation of open code, a first-class eval function, and mutable state.

This tutorial outlines Scheme programs implementing an interpreter, a compiler, and a macro for hygienic quasiquotation.

Keywords Quasiquotation, Program Generation, Hygiene, Lexical Scope, Types

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.3.4 [Programming Languages]: Processors—Code generation

1. Introduction
Quasiquotation is among the most distinctive and celebrated features of Scheme and other dialects of Lisp [2]. It provides concise and expressive syntax for building tree-shaped data as S-expressions. When these trees are actual abstract-syntactic trees, quasiquotation effectively separates the parts of a program that should be executed normally from the code-generating parts. Using terminology from partial evaluation, quasiquotation provides a syntax for separating the binding times of a program, namely which parts are static and which parts are dynamic, to stage this program. In this light, quasiquotation matters in areas ranging from partial evaluation and program specialization [4, 12] to modal and temporal logics [6, 7].

Let us illustrate a typical use of quasiquotation in Scheme by specializing both the linear power function and the logarithmic power function to static exponents. We first implement, by hand or as the result of a binding-time analysis, the following staged linear power function.

(let ([n n])
  (if (= n 0)
    ’1
    ’(* ,x ,(loop (- n 1))))))

In this function, the integer constant 1 and the application of multiplication (* _ _ ) are dynamic. Everything else is static. Evaluating (staged-power-lin n) produces a function from the base value to the code of the result of raising this base value to the nth power. We wish to obtain the code of a function from the base value to the result of raising this base value to the nth power. This is performed using a two-stage eta expansion [5], naively implemented as follows.

(define (eta f)
  ’(lambda (x) ,(f x)))

In this example, one would like an output α-equivalent to the S-expression (lambda (x1) (lambda (x2) x1)).

The problem is that the naive implementation of eta doesn’t preserve lexical binding: The bound variable x may become shadowed when it is substituted into the body of f. Shadowing is avoided by using gensym to introduce a fresh symbol for the residual λ-abstraction. After

(define (eta f)
  (let ([z (gensym)])
    ’(lambda (,z) ,(f z))))

we obtain a correct residual term

(eta (lambda (a) (lambda (b) a)))
⇒ (lambda (x) (lambda (x) (x x)))

In this example, one would have liked an output α-equivalent to the S-expression (lambda (x1) (lambda (x2) x1)).

The problem is that the naive implementation of eta doesn’t preserve lexical binding: The bound variable x may become shadowed when it is substituted into the body of f. Shadowing is avoided by using gensym to introduce a fresh symbol for the residual λ-abstraction. After

(define (eta f)
  (let ([z (gensym)])
    ’(lambda (,z) ,(f z))))

we obtain a correct residual term

(eta (lambda (a) (lambda (b) a)))
⇒ (lambda (g1) (lambda (g2) g1))
The composition of this corrected \( \eta \) with \( \text{staged-power-lin} \) still produces specialized linear power functions:

\[
(\eta \, (\text{staged-power-lin} \, 3))
\Rightarrow (\lambda \, (g3 \, (* \, g3 \, (* \, g3 \, 1))))
\]

The same \( \eta \) can map other functions from code to code into the corresponding code of these functions. As an example, let us repeat the previous exercise using the logarithmic power function. As before, we start from the staged logarithmic power function, defined as follows and using an auxiliary function \( \text{sq} \) that squares its argument.

\[
(\text{define} \, (\text{sq} \, n) \, (* \, n \, n))
\]

\[
(\text{define} \, (\text{staged-power-log} \, n)
(\lambda \, (x))
(\text{if} \, \text{(even?} \, \, n))
(\text{if} \, \text{(!=} \, \, n \, 0))
'x'
'(\text{sq} \, ,(\text{loop} \, (\text{div} \, n \, 2))))
(\text{if} \, \text{(!=} \, \, n \, 1))
'(* \, x \, (\text{sq} \, ,(\text{loop} \, (\text{div} \, n \, 2))))))))))
\]

In \( \text{staged-power-log} \), the integer constant 1 and the application of multiplication \( (* \, \_ \, \_ \) and of squaring \( \text{sq} \, \_ \) are dynamic. Everything else is static.

The composition of \( \eta \) with \( \text{staged-power-log} \) produces specialized logarithmic power functions. For example,

\[
(\eta \, (\text{staged-power-log} \, 6))
\Rightarrow (\lambda \, (g4 \, (\text{sq} \, (* \, g4 \, (\text{sq} \, g4)))))
\]

Programs that preserve lexical binding while generating residual programs are called hygienic [1]. In this tutorial, we investigate quasiquotation for hygienic construction of residual programs. Hygienic quasiquotations are sufficient to implement lexically scoped generating extensions [10], such as the composition of \( \eta \) with \( \text{staged-power-lin} \) or \( \text{staged-power-log} \) presented above. We do not also investigate hygienic inspection of residual programs and we require that the \( \lambda \)-abstraction is the only syntax that binds variables. Program inspection and the ability to introduce new binding forms are necessary to implement hygienic macro expansion [14] such as embodied in Scheme [8]. Our thesis is that, for the construction of residual programs,

\[
\text{Hygiene} = \text{Lexical scope across stages}
\]

and that hygiene can be dealt with using the same techniques that define lexical scope. Starting from a semantics of hygienic quasiquotation, we identify the source of the hygiene condition (Section 2). We then implement an interpreter for the \( \lambda \)-calculus with hygienic quasiquotation corresponding to this semantics (Section 2.5) and we derive a compiler for quasiquotation (Section 2.6).

2. The semantics of hygienic quasiquotation

Quasiquotes (\( ' \)) and unquotes (\( , \)) separate a program into those parts that must be evaluated normally and those that produce code: Program parts that appear under the same number of quasiquotes and unquotes are evaluated normally whereas those that appear under more quasiquotes than unquotes produce code. (It is syntactically illegal for a program part to appear under fewer quasiquotes than unquotes.) Thus, the semantics of a language construct depends on the stage \( n \) at which it appears, where the stage of a program

\[
[x]_n \rho = \rho (x_n)
\]

\[
\[e\]_0 \rho = [e]_1 \rho
\]

\[
\[e\]_n \rho = \lambda \, ([e]_{n+1} \rho)
\]

\[
\lambda \, [e]_1 \rho = [e]_0 \rho
\]

\[
\lambda \, [e]_n \rho = \lambda \, ([e]_{n-1} \rho), \text{ if } n > 1
\]

\[
[\lambda \, x, e]_0 \rho = \lambda \, [e]_0 \rho (x_0 := v)
\]

\[
[\lambda \, x, e]_n \rho = \lambda \, ([e]_n \rho (x_n := z)
\]

\[
[e_0 \, \text{sq} \, e_1]_n \rho = ([e_0]_n \rho) \, \text{sq} \, ([e_1]_n \rho)
\]

Underlined operators build syntax. For example, \( "e_0 \, \text{sq} \, e_1" \) constructs the syntax of an application node whose child nodes are \( e_0 \) and \( e_1 \).

\[
\text{Figure 1.} \quad \text{The semantics of the } \lambda \text{-calculus with quasiquotation}
\]

2.1 Stages

For notational convenience, we distinguish constructs that build program text from “ordinary” mathematical constructs by underlining the former. Thus the underlined parts build abstract syntax trees on the right-hand sides and they pattern match against abstract-syntax trees on the left-hand sides.

Of course, quasiquotation is itself a notation for distinguishing object and meta-level program parts. Indeed, Section 2.5 below shows an implementation of the interpreter that uses quasiquotations to build the underlined parts.

To emphasize the symmetry inherent to the interpreter in Figure 1, we have marked both object and meta-level application using an infix at-sign (\( \circ \)).

2.2 Namespaces

As illustrated in the introduction, identifiers at the same dynamic stage can shadow each other. On the other hand, an identifier that occurs at one stage can never shadow an identifier that appears at a different stage. For example, in both of the following cases, the attempt to let the inner occurrence of \( x \) shadow the outer occurrence of \( x \) fails, because the two \( x \)s are at different stages.

\[
(\text{let} \, ([x \, \text{"outer"}], \, (\text{let} \, ([x \, \text{42}]) \, ,x))
\Rightarrow (\text{let} \, ([x \, \text{42}]) \, \text{"outer"})
\]

\[
(\text{let} \, ([x \, \text{"outer"}], \, (\text{let} \, ([x \, \text{42}]) \, \text{"x"}))
\Rightarrow (\text{let} \, ([x \, \text{"outer"}]\) \text{ x})
\]

The interpreter in Figure 1 specifies such stratified namespaces of identifiers by associating identifiers with the stage at which they are declared. In particular, the environment \( \rho \) binds stage-annotated identifiers \( x_n \) to values.

2.3 Hygiene

The semantics in Figure 1 adopts Barendregt’s variable convention as the means to be hygienic. This convention states that we must avoid covert (i.e., consistently rename) when necessary, to make bound and free variable differ [1, Section 2.1].

In Figure 1, the variable convention applies in the stage-\( n \) clause for \( \lambda \)-abstractions: In this clause, the identifier \( z \) is bound. If it is
also free (in particular, if it appears as a value in the environment \( \rho \)) then we must rename its bound occurrence. (In practice, an interpreter can always rename the bound occurrence and keep track of the renaming in the environment.)

### 2.4 Meta-circularity

The use of the variable convention makes Figure 1 look pleasantly symmetric. However, it introduces a circularity in the definition of the variable-binding mechanism in the clause for dynamic \( \lambda \)-abstractions: The variable convention states that the occurrence \( z \) in

\[
\lambda z. [e]_n, \rho | x_n := z
\]

is lexically bound since no other \( z \) can shadow this \( z \). (Any other \( z \) bound inside \( e \) is consistently renamed.) Thus, the clause for dynamic \( \lambda \)-abstractions in Figure 1 defines the meaning of lexically scoped variable binding in terms of lexically scoped variable binding! This circularity can be avoided by constructing the syntax of a (potentially) capturing \( \lambda \)-abstraction with a fresh identifier. Using the notation \( \lambda x. \) for such a (potentially) capturing \( \lambda \)-abstraction, we can remove the circular definition of lexical scope by replacing the clause for dynamic \( \lambda \)-abstractions in Figure 1 by

\[
\lambda z. [e]_n, \rho | x_n := z
\]

With this definition, the interpreter in Figure 1 becomes the standard definition of hygienic quasiquotation. For example, the mapping \( [] \) in Figure 1 is equivalent to the interpretation of the \( \Lambda \)-calculus with Bracket and Escape in Section 4.3 of Walid Taha’s thesis [21]. (Taha’s Bracket and Escape correspond to hygienic quasiquote and unquote.)

There are other meta-circularities in Figure 1. The semantics of functions, applications, and quasiquotations are all defined in terms of these concepts themselves. These meta-circularities can be avoided, for example, by encoding functions as closures [17] and by using stage-annotated syntax constructors as in Section 2.7 below.

### 2.5 Interpreting hygienic quasiquotation in Scheme

In this section, we implement the interpreter presented above in Scheme. The implementation must correctly generate fresh symbols and it must use a concrete representation of the underlined parts of Figure 1.

In Scheme, it is natural to represent the underlined operations (i.e., the code-generating operations) themselves using quasiquotations. We then distinguish between the hygienic quasiquotation of the source language and the non-hygienic quasiquotation of the implementation language (Scheme).

The implementation of the interpreter is as follows.\(^1\)

```scheme
(define (ev n exp rho)
  (match exp
    [('? symbol?) (lookup rho exp n)]
    [('? quasiquote e) (if (= n 0)
                      (cm 0 e rho)
                      '(((? quasiquote ,,(cm (+ n 1) e rho)))))]
    [('? unquote e) (if (= n 1)
                      (cm 0 e rho)
                      '(((? unquote ,,cm (- n 1) e rho))))]
    [('? lambda (x) e)
     (if (= n 0)
      (let (\[z (gensym)]
          '((lambda ,z ,cm 0 e (ext rho x 0 z)))
          (let (\[z (gensym)]
            '((lambda ,z ,cm n e (ext rho x n z)))))
        '(((lambda ,z ,cm n e (ext rho x n z)))))]
     (if (= n 0)
      '(((cm 0 e0 rho) ,cm 0 e1 rho))
      '(((cm n e0 rho) ,cm n e1 rho))))]

1 The interpreter uses the simple implementation of pattern matching found in the appendix.
```

Let us verify that the example in Section 1 behaves as expected:

```scheme
(define test

  (lambda (eta)
    (eta (lambda (a) (eta (lambda (b) a))))))

(ev 0 test '())

⇒ (lambda (g5) (lambda (g6) g5))
```

Notice that the interpreter uses the idioms `'quasiquote to generate the literal symbol quasiquote in the head of a list. This notation is syntactically equal to `(quote quasiquote). (Ditto for the compiler presented below.)

Notice also that, unlike in Scheme, the interpreter handles the identifiers quasiquote, unquote, and lambda as global keywords that cannot also be used as variables. (Ditto for the compiler presented below.)

### 2.6 Compiling hygienic quasiquotation in Scheme

Instead of interpreting hygienic quasiquotations, we can compile them into the ordinary quasiquotations of Scheme. Such a compiler will surround each dynamic \( \lambda \)-abstraction in the input with a gensym that generates a fresh formal parameter for that \( \lambda \)-abstraction.

The compiler is derived from the interpreter by changing the parts that produce values to parts that produce code instead, as follows.

```scheme
(define (cm n exp rho)
  (match exp
    [('? symbol?) (lookup rho exp n)]
    [('? quasiquote e) (if (= n 0)
                      (cm 1 e rho)
                      '(((? quasiquote ,,cm (+ n 1) e rho))))]
    [('? unquote e) (if (= n 1)
                      (cm 0 e rho)
                      '(((? unquote ,,cm (- n 1) e rho))))]
    [('? lambda (x) e)
     (if (= n 0)
      (let (\[z (gensym)]
          '((lambda ,z ,cm 0 e (ext rho x 0 z)))
          (let (\[z (gensym)]
            '((lambda ,z ,cm n e (ext rho x n z)))))
        '(((lambda ,z ,cm n e (ext rho x n z)))))]
     (if (= n 0)
      '(((cm 0 e0 rho) ,cm 0 e1 rho))
      '(((cm n e0 rho) ,cm n e1 rho))))]
```

\[2012/10/29\]
Again, let us verify that the example in Section 1 behaves as expected. (We avoid repeating the intermediate output from the compiler. In a real interactive session with Scheme, this output must be pasted back into the prompt to run it.)

\(\text{cm} \ 0 \ \text{test} \ '(())\)

\(⇒\)

\((\text{lambda} \ (g7)
\quad (g7 \ (\text{lambda} \ (g8) \ (g7 \ (\text{lambda} \ (g9) \ g8))))))\)

\(\text{cm} \ 1 \ \text{test} \ '(())\)

\(⇒\)

\((\text{lambda} \ (g10)
\quad \text{let} \ ([g11 \ (\text{gensym})])
\quad '((\text{lambda} \ ,(g11) \ ,(g10 \ g11))))))\)

\(⇒\)

\((\text{lambda} \ (g12) \ (\text{lambda} \ (g13) \ g12))\)

### 2.7 Macro-expanding hygienic quasiquotation in Scheme

It is tempting to use the compiler \text{cm} from above as the backbone of a Scheme macro for hygienic quasiquotation. However, Scheme recursively expands the output of an expanded macro so a naive macro expander that translates (hygienic) quasiquotations into (ordinary) quasiquotations would loop forever.

In the implementation that follows, we circumvent this problem by not generating quasiquotations in the output. Instead we produce terms in which dynamic parts are built using stage-annotated syntax constructors.

#### 2.7.1 Stage-annotated syntax constructors

This is the definition of hygienic stage-annotated syntax constructors. Similar constructors were used by Glück and Jørgensen as a library for code generation in multi-stage program specialization [11]:

\[
\begin{align*}
&\text{define (mk-atom n a)} \\
&\quad \text{if (}= n 0) \quad '(','quote ,a) \\
&\quad \text{else} \quad (\text{mk-atom},(- n 1),a)\))
\end{align*}
\]

\[
\begin{align*}
&\text{define (mk-app n e0 e1)} \\
&\quad \text{if (}= n 0) \quad '(',e0 ,e1) \\
&\quad \text{else} \quad (\text{mk-app},(- n 1),e0 ,e1)\))
\end{align*}
\]

\[
\begin{align*}
&\text{define (mk-lam n x e)} \\
&\quad \text{if (}= n 0) \quad '('\text{let} ([,x (\text{gensym})]) \\
&\quad \text{mk-lam},(- n 1),x ,e)\))
\end{align*}
\]

Calls to these constructors reproduce themselves with a decreased value of \(n\), except that when \(n\) is zero, ordinary syntax is constructed. For example,

\[
\begin{align*}
&\text{(mk-app 1 (mk-lam 1 'x 'x) (mk-atom 1 42))} \\
&⇒\ 
\text{(mk-app 0}
\quad \text{let} \ ([,x (\text{gensym})]) \ (\text{mk-lam} \ 0 \ x \ x)) \\
&\quad \text{(mk-atom} \ 0 \ 42))
\end{align*}
\]

\[
\begin{align*}
&\text{⇒} \\
&\quad \text{((lambda} \ (g14) \ g14) \ '42)
\end{align*}
\]

#### 2.7.2 Generating stage-annotated syntax

The translation of quasiquotations into stage annotated syntax is straightforward. We simply replaces syntax with the corresponding staged-annotated syntax:

\[
\begin{align*}
&\text{define (xp n exp)} \\
&\quad \text{(match exp}
\quad [\text{? symbol?}] \ \text{exp} \\
&\quad [\text{? atom?}] \ (\text{mk-atom} n \ exp)] \\
&\quad [\text{? quasiquote e}] \ (\text{xp} \ (+ n 1) \ e)] \\
&\quad [\text{? unquote e}] \ (\text{xp} \ (- n 1) \ e)] \\
&\quad [\text{? lambda (x) e}] \ (\text{mk-lam} \ n \ x \ (\text{xp} \ n \ e))] \\
&\quad [\text{e0 e1}] \\
&\quad (\text{mk-app} n \ (\text{xp} \ n \ e0) \ (\text{xp} \ n \ e1))\])
\end{align*}
\]

\[
\begin{align*}
&\quad \text{else} \\
&\quad \text{(error 'xp "syntax error" exp))})
\end{align*}
\]

### 2.7.3 A Scheme macro for hygienic quasiquotation

The definition of a Scheme macro for hygienic quasiquotation can now be defined using syntax-case. We overwrite the built-in \text{quasiquote} keyword, and we translate back and forth between S-expression and syntax: (This translation via S-expressions makes our hygienic quasiquotation incompatible with Scheme’s hygienic macro system.)

\[
\begin{align*}
&\text{(define-syntex quasiquote}
\quad \text{(lambda (stx)} \\
&\quad \text{(syntax-case stx ()}
\quad [(k e)
\quad \text{(datum->syntax #'k)
\quad (xp 1 (syntax->datum #'e)))])\}))
\end{align*}
\]

With this definition, we can repeat the example from Section 1.

\[
\begin{align*}
&\text{(define (eta f) '(lambda (x) ,(f 'x))} \\
&\quad \text{(eta (lambda (y) y))}
\end{align*}
\]

\[
\begin{align*}
&\quad ⇒ \quad (\text{lambda} \ (g15) \ g15) \\
&\quad \text{(eta (lambda (a) (eta (lambda (b) a)))))}
\end{align*}
\]

\[
\begin{align*}
&\quad ⇒ \quad (\text{lambda} \ (g16) \ (\text{lambda} \ (g17) \ g16))
\end{align*}
\]

Notice that the definition of hygienic quasiquotation collapses the namespaces of identifiers into one. For example, in contrast with Section 2.2, the following attempt to let one identifier shadow another identifier with the same name defined at a different stage succeeds.

\[
\begin{align*}
&\quad \text{((lambda} \ (x) \ '((\lambda \ (x) ,x) 42)) \ \text{"outer"}} \\
&\quad ⇒ \quad ((\text{lambda} \ (g18) \ g18) \ '42)
\end{align*}
\]

Kiselyov and Shan have developed a more complete macro for multi-stage programming which handles hygienic quasiquotation, lifting of values across stages, and other binding constructs such as let-binding [13]. They require the macro to be called with a keyword different from \text{quasiquote}, thereby avoiding a looping macro expansion.

### 3. A type system for hygienic quasiquotation

Multi-stage programming using quasiquotation is a useful feature in not only Lisp and Scheme. In statically typed languages, the existence of quasiquotation demands a type system that guarantees that well-typed programs produce only well-typed code. In sufficiently simple languages, such type systems have been known since the mid-1990s [6]. However, if the language also includes an eval procedure and it allows side effects, then such type systems have been difficult to get right.

In this section, we outline a simple type system, \(λ[^1]\), for hygienic quasiquotation [18]. In this type system, the type of a quoted expression carries the “scope” in which the expression can be inserted. Such a scope is described by the set of (dynamic) identifiers that the quoted expression can access in its surroundings. Since these are actual program variables, and since we insist on lexical
Kinding rules:

\[ \lambda \text{The typing rules for variables, abstractions and application closely} \]

\[ \text{denotes the variables in the type environment that are bound at} \]

\[ \text{the rule for quasiquote, } \Gamma \text{maps stage-annotated variables to types. In} \]

\[ \text{particular, a variable that occurs in a} \]

\[ \text{term requires the variables} \]

\[ \lambda \text{binding constructs in terms.) In particular, a variable that occurs in a} \]

\[ \text{quasiquotation} \]

\[ \text{Figure 2. In this figure, the short-hand notation} \]

\[ \text{denotes the stage at which the term occurs.} \]

\[ \text{The typing rules specifies stratified namespaces for identifiers. The} \]

\[ \text{type environment } \Gamma \text{maps stage-annotated variables to types. In} \]

\[ \text{the rule for quasiquote, } \Gamma|_{n+1} \text{contains the variables that go into} \]

\[ \text{the code type. In the rule for unquote, these are the surrounding} \]

\[ \text{variables that the unquoted expression can access.} \]

3.3 Hygiene

The typing rules in Figure 2 adopt Barendregt’s variable convention. In the rule for \( \lambda \)-abstractions, the variable \( x \) is bound. If it is also free (in particular, if it appears in a type in the type environment \( \Gamma \)) then we must rename its bound occurrence. (In practice, a type checker can always rename the bound occurrence and keep track of the renaming in the type environment.)

The condition \( \Gamma \vdash_n t' \) in the rule for \( \lambda \)-abstraction ensures that \( t \) is a well-formed type. In the simply-typed \( \lambda \)-calculus without quasiquotation, any type is well formed. But in \( \lambda^3 \), a type can include the names of variables, and \( \Gamma \vdash_n t' \) holds only if the variables mentioned in \( t \) are in scope at the correct stage.

3.4 Examples

Here we encode the linear and the logarithmic power functions in a typed setting. To do so, first notice that the formal parameter \( \ell \) of \( \text{eta} \) is passed a code value that contains the variable \( x \) and that this variable is not in scope at the binding of \( \ell \). Thus, \( \text{eta} \) is not well typed in the type system presented in Figures 2 and 3. (Remember that \( x \) will actually be renamed during evaluation, to enforce lexical scope.) Instead of using \( \text{eta} \) as a stand-alone function, we therefore inline it into \( \text{staged-power-log} \) to produce the following specializing version of the linear power function.

\[
\begin{align*}
(\text{define (gen-power-lin n)} & \text{ '(lambda (x)} \\
& \text{ , (let loop ([n n])} \\
& \text{ \quad (if (= n 0)} \\
& \text{ \quad \quad ('1)} \\
& \text{ \quad \quad \quad (* x , (loop (- n 1)))))}\) \\
\end{align*}
\]

In this program, the dynamically bound \( x \) and both of the statically bound occurrences of \( n \) have type \( \text{int} \). The function \( \text{gen-power-lin} \) has type

\[ \text{int} \rightarrow \text{[int \rightarrow int]} \]

Passing it a static argument \( n \) yields the text of a function that takes a dynamic argument \( x \) and that returns \( x^n \), using \( n \) residualized calls to the function \( \ast \). For example, passing it the value 3 yields the following text of a function that raises it argument to the power of 3.

\[
\begin{align*}
(\text{gen-power-lin 3} & \text{ => (lambda (g22) (* g22 (* g22 (* g22 1)))}) \\
\end{align*}
\]

Similarly, we inline \( \text{eta} \) into \( \text{staged-power-log} \) to produce the following specializing version of the logarithmic power function.

\[
\begin{align*}
(\text{define (gen-power-log n)} & \text{ '(lambda (x)} \\
& \text{ , (let loop ([n n])} \\
& \text{ \quad (if (= n 0)} \\
& \text{ \quad \quad ('1)} \\
& \text{ \quad \quad \quad (sqr , (loop (div n 2)))))} \\
& \text{ \quad \quad \quad ('x)} \\
& \text{ \quad \quad \quad \quad (* x \text{, (loop (div n 2))}))}) \\
\end{align*}
\]

Again, the dynamically bound \( x \) and both of the statically bound occurrences of \( n \) have type \( \text{int} \). The function \( \text{gen-power-log} \) has type

\[ \text{int} \rightarrow \text{[int \rightarrow int]} \]
Type rules:

\[
\frac{\Gamma' \vdash n : t' \quad \Gamma' \subseteq \Gamma \quad t' \leq t}{\Gamma \vdash n : t} \quad \text{(Subsumption)}
\]

Subtyping rules:

\[
\frac{b \leq b}{(b \leq)}
\]
\[
\frac{u_1 \leq t_1 \quad t_2 \leq u_2}{t_1 \rightarrow t_2 \leq u_1 \rightarrow u_2} \quad \text{(→≤)}
\]
\[
\frac{\{x, \ldots \} \subseteq \{y, \ldots \} \quad t \leq u}{[x, \ldots]t \leq [y, \ldots]u} \quad \text{([] ≤)}
\]

Figure 3. Subtyping for hygienic quasiquotation

Passing it a static argument \(n\) yields the text of a function that takes a dynamic argument \(x\) and that returns \(x^n\), using \([\log_2(n)]\) residualized calls to the function \(\text{aqr}\) and up to \([\log_2(n)]\) residualized calls to the function \(*\). For example, passing it the value 6 yields the following text of a function that raises it power to the 6.

\[
\begin{align*}
\text{(gen-power-log 6)}
\implies & \quad (\text{lambda} \ (g23) \ (\text{aqr} \ (* \ g23\ g23)))
\end{align*}
\]

3.5 Subtyping

The type system in Figure 2 prevents a dynamic term from ever ending up under a set of bindings different from that under which it is constructed. The reason is that this set of variables is part of the type of such a dynamic term, and the rules for quasiquotation and unquote require exact matches between these variables and those found in the type environment. For example, if the term \(\text{quasiquote}\ x\) appears under two (dynamic) bindings, one for \(x\) (of type \(t\)) and one for \(y\), then this term has type \([x, y]t\) and it can only be used as a value under these dynamic bindings.

As a type system for a practical language, this is a restriction. We can lift it by the subtyping extension shown in Figure 3. This extension allows row subtyping to the code type and to the type environment. Intuitively, it allows the type system (1) to remove variables from the type environment when they are not needed and (2) to add variables to code types in covariant position and to remove variables from code types in contravariant position. For example, without subtyping, the program

\[
\text{(let \((c \ '1') \ (\text{lambda} \ (x) ,c))}
\]

fails to type check since \(c\) of the closed code type \([\ ]\int\) cannot be inserted in a scope that contains \(x\). (This restriction is lifted by the part of the subsumption rule that allows the type system to continue with a greater type of \(c\).

As another example, without subtyping, the (incomplete) program

\[
\text{(lambda} \ (x) ,\ldots \text{ (eval '1) ...})
\]

also fails to type check, since \('1\)' must have type \([x]\int\) and \(\text{eval}\) expects closed code as input. (This restriction is lifted by the part of the subsumption rule that allows the type system to remove the variable \(x\) from the type environment before type checking \('1\).)

4. Hygienic quasiquotation in other languages

Several existing languages have been extended with features inspired by (but not necessarily identical to) quasiquotation. Examples include \(\text{'C}\), an extension of \(\text{C}\) [9], the quote-antiquote mechanism of Standard ML [15], DynJava, an extension of Java [16], the quotations of \(\text{F#}\) [20], and Template Haskell [19]. However, these languages either are not higher-order \(\text{'C}\), do not allow general expressions under unquote (DynJava, \(\text{F#}\)), bypass the type system of the host language (Template Haskell), represent residual code as unstructured strings (Standard ML), or do not allow multiple stages \(\text{'C},\ DynJava,\ Template\ Haskell\).

MetaOCaml is a full-blown multi-stage programming language that combines higher order features with a hygienic quasiquotation mechanism [3]. MetaOCaml is an important framework for the ongoing study of type systems for multi-stage programming. (Its type system is sound under the assumption that one doesn’t use side effects.)

5. Conclusions

Lisp and Scheme have had a remarkable impact on the field of higher-order code generation. Indeed, the syntax, semantics, and many motivating applications of quasiquotation as an elegant notation for constructing code takes their origin in Lisp and Scheme. We have demonstrated how to specify and implement a hygienic quasiquotation facility in Scheme, and we have outlined a type system for hygienic quasiquotation which is significantly simpler than existing type systems for hygienic quasiquotation and which provides one type of lexically scoped code that precisely accounts for the contexts in which code can be inserted.

Acknowledgments

This tutorial has benefited from Michael D. Adams’s comments.

References

A. Auxiliary definitions

A.1 A pattern matcher

(define-syntax match
  (syntax-rules (else quote ?)
    [(_ e cs ...) (not (identifier? #'e))
      (let ((v e)) (match v cs ...))]
    [(_ v) (error 'match "no match" v)]
    [(_ v [else e]) e]
    [(_ v [x e] cs ...) (identifier? #'x)
      (let ([x v]) e)]
    [(_ v ['d e] cs ...) (if (equal? v 'd) e (match v cs ...))]
    [(_ v [(? p) e] cs ...) (if (pair? v)
      (match (car v)
        [p (match (cdr v) [q e] [else (f)])
          (else (f))]
        (f))])
    [(_ v [d e] cs ...) (if (equal? v d) e (match v cs ...))])

A.2 Procedures on environments

(define (ext rho x n v)
  (cons (list x n v) rho))

(define (lookup rho x n)
  (cond [(null? rho)
          (error 'lookup "unbound variable" x)]
        [(and (equal? x (list-ref (car rho) 0))
              (equal? n (list-ref (car rho) 1)))
          (list-ref (car rho) 2)]
        [else (lookup (cdr rho) x n)]))