A Self-Hosting Evaluator using HOAS

A Scheme Pearl

Eli Barzilay
Northeastern University
eli@barzilay.org

Abstract

We demonstrate a tiny, yet non-trivial evaluator that is powerful enough to run practical code, including itself. This is made possible using a Higher-Order Abstract Syntax (HOAS) representation — a technique that has become popular in syntax-related research during the past decade. With a HOAS encoding, we use functions to encode binders in syntax values, leading to an advantages of reflecting binders rather than re-implementing them.

In Scheme, hygienic macros cover problems that are associated with binders in an elegant way, but only when extending the language, i.e., when we work at the meta-level. In contrast, HOAS is a useful object-level technique, used when we need to *represent* syntax values that contain bindings — and this is achieved in a way that is simple, robust, and efficient. We gradually develop the code, explaining the technique and its benefits, while playing with the evaluator.

1. Introduction

Higher-Order Abstract Syntax (HOAS) is a technique for representing syntax with bindings using functions. This is a form of reflection in the sense that binders in the object level are represented using binders in the meta level. The result is simple (no need for sophisticated substitution mechanisms), robust (the meta-level is our language, which better implement scope correctly), and efficient (as it is part of the core implementation).

HOAS has been in use for a while now [13], a good overview is given by Hofmann [9], and in [1, Section 4.7]. However, it is more popular in the strictly typed world than it is in Scheme. In part, this may be due to Scheme's hygienic macro facility, which allows hooking new kinds of syntactic constructs into the language in a way that respects lexical scope. Using a high level macro system means that Schemers rarely need to represent syntax directly — instead, they work at the meta-level, extending the language itself. This is unfortunate, as HOAS can be a useful tool for syntax-related work. Furthermore, it is possible to formalize HOAS in a way that is more natural in a dynamically-typed language than it is in statically-typed ones, which corresponds to a HOAS formalization in Nuprl [3] that builds on the predicative nature of its type theory [2, 1, 12].

The purpose of this Scheme pearl is to demonstrate the use of HOAS in Scheme, with the goal of making this technique more accessible to Schemers[1]. As demonstrated below, using macros in Scheme facilitates the use of HOAS, as there is no need for an external tool for translating concrete syntax into HOAS representations. In itself, HOAS is a representation tool for object-level values, not for meta-level work where bindings are directly accessible in some way. It is, however, used in some meta-linguistic systems for implementing syntactic tools[2]. For example, it could be used as the underlying term representation in a language-manipulating tool like PLT's Reduction Semantics [11].

2. A Toy Evaluator

The presentation begins with a simple evaluator. Our goal is to evaluate a Lambda-Calculus-like language using reductions, so we need a representation for `lambda` abstractions and applications. To make this example more practical, we also throw in a conditional `if` special form, make it handle multiple arguments, and use call-by-value. A common evaluator sketch for such a language is[3]:

```
(define (ev expr)
  (cond [(not (pair? expr)) expr]
        [(eq? 'if (car expr))
         (ev (if (ev (cadr expr)) (caddr expr) (cadddr expr)))]
        [else (ev (let ([f (ev (car expr))])
                    (substitute (body-of f)
                                (args-of f)
                                (map ev (cdr expr)))))]))
```

where an application is always assumed to have an abstraction in its head position, and the `args-of` and `body-of` functions pull out the corresponding parts. As expected, the main issue here is implementing a proper substitution function. Common approaches include using symbols and renaming when needed, or using symbols 'enriched' with lexical binding information ('colors').

[1] The presentation is loosely based on a comp.lang.scheme post from October 2002.

[2] It might be possible that HOAS can be used for implementing a low-level macro facility that the high-level hygienic macro system builds on. HOAS should not be confused with current low-level syntactic systems like syntactic closures or `syntax-case`.

[3] Note that details like error checking are omitted, and that we use atomic Scheme values such as booleans and numbers to represent themselves.

Scheme and Functional Programming September 17th 2006, Portland, OR

On the other hand, we can use a higher-order abstract syntax representation for our language, which will make things easier to handle than raw S-expressions. For this, we represent an abstraction using a Scheme function, and bound occurrences are represented as bound occurrences in the Scheme code. For example,

```
(lambda (x y) (if x x y))
```

is represented by

```
(lambda (x y) (list 'if x x y))
```

— and substitution is free as it is achieved by applying the *Scheme* function.

3. Creating HOAS Representations

Given that our representation of `lambda` abstractions uses Scheme `lambda` expressions, we need some facility to create such representation while avoiding the possible confusion when using `lambda` for different purposes. The common approach for creating such representations is to use a preprocessor that translates concrete syntax into a HOAS value. In Scheme, this is easily achieved by a macro that transforms its body to the corresponding representation:

```
;; Translates simple terms into a HOAS representation
;; values, which are:
;; Term = atom                       ; literals
;;      | (list 'if Term Term Term)  ; conditionals
;;      | (Term ... -> Term)         ; abstractions
;;      | (list Term ...)            ; applications
(define-syntax Q
  (syntax-rules (lambda if)
    [(Q (lambda args b)) (lambda args (Q b))]
    [(Q (if x y z))      (list 'if (Q x) (Q y) (Q z))]
    [(Q (f x ...))       (list (Q f) (Q x) ...)]
    [(Q x)               x]))
```

A few interaction examples can help clarify the nature of these values:

```
> (Q 1)
1
> (Q (+ 1 2))
(#<primitive:+> 1 2)
> (Q (if 1 2 3))
(if 1 2 3)
> (Q (lambda (x) (+ x 1)))
#<procedure>
```

The last one is important — the `lambda` expression is represented by a Scheme procedure, which, when applied, returns the result of substituting values for bound identifiers:

```
> (define foo (Q (lambda (x) (+ x 1))))
> (foo 2)
(#<primitive:+> 2 1)
```

Using the representations that the 'quoting' macro `Q` creates, a complete substitution-based evaluator is easily written:

```
;; ev : Term -> Val
;; evaluate an input term into a Scheme value
(define (ev expr)
  (cond [(not (pair? expr)) expr]
        [(eq? 'if (car expr))
         (ev (if (ev (cadr expr)) (caddr expr) (cadddr expr)))]
        [else (ev (apply (ev (car expr))
                         (map ev (cdr expr))))]))
```

Note that the underlined `apply` expression invokes the Scheme procedure which performs the substitution that is needed for the beta-reduction: it is a '`Term`→`Term`' function. The result of this application expression is therefore a piece of (post-substitution) syntax that requires further evaluation.

In addition, this is a simple substitution-based evaluator — no environments, identifier lookups, or mutation. Scheme values are used as self-evaluating literals (some achieved by a Scheme identifier reference), including Scheme procedures that are exposed as primitive values. Specifically, the last `cond` clause is used for both primitive function applications and beta reductions — this leads to certain limitations and possible errors, so it is fixed below.

It is easy to confuse the current representation as a trick; after all, we represent abstractions using Scheme abstractions, and beta-reduce using Scheme applications — sounds like we end up with a simple meta-circular Scheme evaluator that inherits Scheme's features. This is not the case, however: Scheme applications achieves nothing more than substitution. To demonstrate this, the evaluator can easily be changed to use a lazy evaluation regimen if it avoids evaluating abstraction arguments. This requires a distinction between strict and non-strict positions. For simplicity, we only distinguish primitive functions (all arguments are strict) and abstractions (no arguments are strict) using MzScheme's `primitive?`[4] predicate:

```
;; ev* : Term -> Val
;; evaluate an input term into a Scheme value, lazy version
(define (ev* expr)
  (cond [(not (pair? expr)) expr]
        [(eq? 'if (car expr))
         (ev* (if (ev* (cadr expr)) (caddr expr) (cadddr expr)))]
        [else (ev* (let ([f (ev* (car expr))])
                     (apply f (if (primitive? f)
                                  (map ev* (cdr expr))
                                  (cdr expr)))))]))
```

And the result is a lazy language, where we can even use the call-by-name fixpoint combinator:

```
> (ev (Q ((lambda (x y z) (if x y z))
          #t (display "true\n") (display "false\n"))))
true
false
> (ev* (Q ((lambda (x y z) (if x y z))
           #t (display "true\n") (display "false\n"))))
true
> (ev* (Q (((lambda (f)
              ((lambda (x) (f (x x)))
               (lambda (x) (f (x x)))))
            (lambda (fact)
              (lambda (n)
                (if (zero? n) 1 (* n (fact (- n 1)))))))
           5)))
120
```

4. Advantages of the HOAS Representation

At this point we can see the advantages of the HOAS representation. These are all due to the fact that the Scheme binding mechanism is *reflected* rather than *re-implemented*.

Free substitution: since we use Scheme functions, the Scheme implementation provides us with free substitution — we get a substituting evaluator, without the hassle of implementing substitution.

Robust: dealing with the subtleties of identifier scope (substitution, alpha renaming, etc) is usually an error-prone yet critical element in code that deals with syntax. In our evaluator, we

[4] A predicate that identifies primitive built-in procedures.

need not worry about these issues, since it reflects the mechanism that already exists in the implementation we use.

Efficient: the representation lends itself to efficient substitution for two reasons. First, function calls are an essential part of functional languages that must be very efficient; a feature that our substitution inherits. Second, if we incrementally substitute some syntax value with multiple binding levels, then the substitutions are not carried out immediately but pushed to substitution cache contexts (=environments), which are the implementation's efficient representation of closures.

Good integration: representing concrete syntax with Scheme S-expressions is superior to the flat string representations that is found in other languages because the structure of the syntax is reflected in syntax values (values are "pre-parsed" into trees). In a similar way, HOAS adds yet another dimension to the representation — scope is an inherent part of representations (lexical scopes are already identified and turned to closures). We therefore enjoy all functionality that is related to scope in our implementation. For example, the unbound identifiers are caught by the implementation, analysis tools such as DrScheme's "Check Syntax" [6] work for bindings in the representation, macros can be used, etc.

These advantages, however, do not come without a price. More on this below.

5. Improving the Code

So far, the evaluator is simple, but the code is somewhat messy: `lambda` abstractions and primitive procedures are conflated, lists are used both as values and as syntax representations. Furthermore, the evaluator is not as complete as it needs to be to host itself. We begin by improving the code, and in the following section we will extend it so it can run itself.

We begin by introducing a new type for our syntax objects, and use this type to create tagged values for lambda abstractions and applications:

```
;; A type for syntax representation values
;; Term = atom                              ; literals
;;      | (term 'if Term Term Term)         ; conditionals
;;      | (term 'lam (Term ... -> Term))    ; abstractions
;;      | (term 'app Term ...)              ; applications
(define-struct term (tag exprs) #f)
(define (term tag . args) (make-term tag args))

;; Translates simple terms into a HOAS representation
(define-syntax Q
  (syntax-rules (lambda if)
    [(Q (lambda args b)) (term 'lam (lambda args (Q b)))]
    [(Q (if x y z))      (term 'if  (Q x) (Q y) (Q z))]
    [(Q (f x ...))       (term 'app (Q f) (Q x) ...)]
    [(Q x)               x]))
```

`ev` is then adapted to process values of this type.

```
;; ev : Term -> Val
;; evaluate an input term into a Scheme value
(define (ev expr)
  (if (term? expr)
    (let ([subs (term-exprs expr)])
      (case (term-tag expr)
        [(lam) expr]
        [(if)  (ev (if (ev (car subs)) (cadr subs) (caddr subs)))]
        [(app) (let ([f (ev (car subs))]
                     [args (map ev (cdr subs))])
                 (cond [(and (term? f) (eq? 'lam (term-tag f)))
                        (ev (apply (car (term-exprs f)) args))]
                       [(procedure? f)
                        (apply f args)]
                       [else (error 'ev "bad procedure")]))]
        [else (error 'ev "bad tag")]))
    expr))
```

We can now test this evaluation procedure:

```
> (ev (Q (lambda (x) (+ x 1))))
#3(struct:term lam (#<procedure>))
> (ev (Q ((lambda (x) (+ x 1)) 2)))
3
> (define plus1 (Q (lambda (x) (+ x 1))))
> (ev (Q (plus1 2)))
3
```

As the previous version, this evaluator does not maintain its own environment, instead, it uses the Scheme environment (in cooperation with the quotation macro that leaves bindings untouched). This is used as a definition mechanism that is demonstrated in the last example — but we have to be careful to use such values only in a syntax-representation context. Because the representation is using Scheme closures, we can use recursive Scheme definitions to achieve recursion in our interpreted language:

```
> (define fact
    (Q (lambda (n)
         (if (zero? n) 1 (* n (fact (- n 1)))))))
> (ev (Q (fact 30)))
265252859812191058636308480000000
```

Again, making this evaluator lazy is easy: we only need to avoid evaluating the arguments on beta reductions. In fact, we can go further and 'compile' `lambda` expressions into Scheme closures that will do the reduction and use `ev*` to continue evaluating the result. To deal with strict primitives properly, we evaluate them to a wrapper function that evaluates its inputs[5]:

```
;; ev* : Term -> Val
;; evaluate an input term into a Scheme value,
;; this version is lazy, and 'compiles' closures to Scheme procedures
(define (ev* expr)
  (cond
    [(term? expr)
     (let ([subs (term-exprs expr)])
       (case (term-tag expr)
         [(lam) (lambda args (ev* (apply (car subs) args)))]
         [(if)  (ev* (if (ev* (car subs))
                         (cadr subs)
                         (caddr subs)))]
         [(app) (apply (ev* (car subs)) (cdr subs))]
         [else (error 'ev "bad tag")]))]
    [(primitive? expr)
     (lambda args (apply expr (map ev* args)))]
    [else expr]))
```

On first look, this change seems a bit dangerous — not only is a `lambda` expression represented as a Scheme closure, evaluating it returns a Scheme closure. In fact, this approach works as the types demonstrate: the function that is part of the representation is '`Term`→`Term`', whereas the 'compiled' closure is a '`Term`→`Val`' function. Note also that Scheme primitives act as primitives of the interpreted language ('`Val`→`Val`'), and the evaluator wraps them as a '`Term`→`Val`' function that allows uniform treatment of both cases in applications.

Here are a few examples to compare with the previous evaluator:

[5] Ideally, *any* procedure that is not the result of evaluating a `lambda` expression should be wrapped. In MzScheme it is possible to tag some closures using applicable structs, but in this paper the code is kept short.

```
> (ev* (Q (lambda (x) (+ x 1))))
#<procedure>
> (ev* (Q ((lambda (x) (+ x 1)) 2)))
3
> (ev* (Q ((lambda (x y) (+ x 1)) 2 (/ 2 0))))
3
> ((ev* (Q (lambda (x) (+ x 1))))
   (Q 2))
3
```

In the last example, the 'Term→Val' procedure that is the result of evaluating the first part is directly applied on `(Q 2)` (a syntax) which is essentially how the outermost application of the second example is handled. This application jumps back into the evaluator and continues the computation.

Using the Scheme toplevel for definitions, we can define and use a lazy fixpoint combinator:

```
> (define Y
    (ev* (Q (lambda (f)
              ((lambda (x) (f (x x)))
               (lambda (x) (f (x x))))))))
> (define fact0
    (ev* (Q (lambda (fact)
              (lambda (n)
                (if (zero? n) 1 (* n (fact (- n 1)))))))))
> (ev* (Q (Y fact0)))
#<procedure>> (ev* (Q ((Y fact0) 30)))
265252859812191058636308480000000
```

Finally, as an interesting by-product of this namespace sharing, we can even use the call-by-name fixpoint combinator with Scheme code, as long as we use `ev*` to translate the function into a Scheme function:

```
> ((Y (lambda (fact)
        (lambda (n)
          (let ([fact (ev* fact)])
            (if (zero? n) 1 (* n (fact (- n 1))))))))
   30)
265252859812191058636308480000000
```

6. Making `ev` Self-Hosting

In preparation for making our evaluator self-hosting, we need to deal with representations of all forms that are used in its definition. Again, to make things simple, we avoid adding new core forms — instead, we translate the various forms to ones that the evaluator already knows how to deal with. We will use 'nested' instantiations of our evaluator, which will require nested use of the `Q` quotation form — this is a minor complication that could be solved using macro-CPS [8, 10], but in MzScheme [7] it is easier to write a `syntax-case`-based macro, which uses a simple loop for nested occurrences of `Q`. The new (and final) definition is in Figure 1. On first look it seems complex, but it is merely translating additional forms into known ones, and propagates the transformation into the `delay` special form (which will be needed shortly).

The lazy evaluator is slightly modified: call-by-name is too slow to be usable when nesting multiple evaluators, so we change it to use call-by-need instead. For this, we make it create `ev*` promises for function arguments, and automatically force promise values so they are equivalent to plain values. We also need to treat the `term` constructor as a primitive (otherwise it will contain promises instead of values). The definition follows.

```
;; ev* : Term -> Val
;; evaluate an input term into a Scheme value, uses call-by-need
(define (ev* expr)
  (cond
    [(term? expr)
     (let ([subs (term-exprs expr)])
       (case (term-tag expr)
         [(lam) (lambda args
                  (ev* (apply (car subs)
                              (map (lambda (a) (delay (ev* a)))
                                   args))))]
         [(if)  (ev* (if (ev* (car subs))
                         (cadr subs)
                         (caddr subs)))]
         [(app) (apply (ev* (car subs)) (cdr subs))]
         [else (error 'ev* "bad tag")]))]
    [(promise? expr) (ev* (force expr))]
    [(primitive*? expr)
     (lambda args (apply expr (map ev* args)))]
    [else expr]))

(define (primitive*? x)
  (or (primitive? x) (eq? x term)))
```

And with this change we are finally ready to run the evaluator code in itself.

7. Bootstrapping the Evaluator

First, we use the strict evaluator to evaluate a nested copy of itself. In this definition, `ev` is the same as the strict version above, and its code is used with no change.

```
(define ev1
  (ev (Q (Y (lambda (ev)
              (lambda (expr)
                (cond
                  [(term? expr)
                   (let ([subs (term-exprs expr)])
                     (case (term-tag expr)
                       [(lam) (lambda args
                                (ev (apply (car subs) args)))]
                       [(if)  (ev (if (ev (car subs))
                                      (cadr subs)
                                      (caddr subs)))]
                       [(app) (apply (ev (car subs))
                                     (map ev (cdr subs)))]
                       [else (error 'ev1 "bad tag")]))]
                  [else expr])))))))
```

We can verify that this evaluator works as expected:

```
> (ev (Q (ev1 (Q (+ 1 2)))))
3
> (ev (Q (ev1 (Q ((lambda (x) (+ x 2)) 1)))))
3
```

We can continue this and implement a third evaluator in `ev1` — using the same definition once again. The result is again working fine.

```
> (define ev2
    (ev (Q (ev1 (Q (lambda (expr)
                     ;; Same code as ev1, substituting 'ev2' for 'ev1'
                     ))))))
> (ev (Q (ev1 (Q (ev2 (Q (+ 1 2)))))))
3
> (ev (Q (ev1 (Q (ev2 (Q ((lambda (x) (+ x 2)) 1)))))))
3
```

It is interesting to compare the performance of the three evaluators. We do this by defining a Fibonacci function in each of the three levels and in Scheme:

```
;; Translates terms into a HOAS representation
(define-syntax (Q s)
  (let transform ([s s])
    (syntax-case s (Q quote lambda if let and or cond case else delay)
      [(Q (Q x)) ; transform once, then reprocess:
       (with-syntax ([1st-pass (transform (syntax (Q x)))])
         (syntax (Q 1st-pass)))]
      [(Q (quote x))           (syntax 'x)]
      [(Q (lambda args b))     (syntax (term 'lam (lambda args (Q b))))]
      [(Q (if x y z))          (syntax (term 'if  (Q x) (Q y) (Q z)))]
      [(Q (let ([x v] ...) b)) (syntax (Q ((lambda (x ...) b) v ...)))]
      [(Q (and))               (syntax #t)]
      [(Q (and x))             (syntax x)]
      [(Q (and x y ...))       (syntax (Q (if x (and y ...) #f)))]
      [(Q (or))                (syntax #f)]
      [(Q (or x))              (syntax x)]
      [(Q (or x y ...))        (syntax (Q (let ([x* x]) (if x* x* (or y ...)))))]
      [(Q (cond))              (syntax 'unspecified)]
      [(Q (cond [else b]))     (syntax (Q b))]
      [(Q (cond [test b] clause ...))
                               (syntax (Q (if test b (cond clause ...))))]
      [(Q (case v))            (syntax 'unspecified)]
      [(Q (case v [else b]))   (syntax (Q b))]
      [(Q (case v [(tag) b] clause ...)) ; (naive translation)
                               (syntax (Q (if (eqv? v 'tag) b (case v clause ...))))]
      [(Q (delay x))           (syntax (delay (Q x)))]
      [(Q (f x ...))           (syntax (term 'app (Q f) (Q x) ...))]
      [(Q x)                   (syntax x)])))
```

Figure 1. Full quotation code

```
(define fib
  (lambda (n)
    (if (<= n 1) n (+ (fib (- n 1)) (fib (- n 2))))))
(define fib0
  (ev (Q (lambda (n) ...))))
(define fib1
  (ev (Q (ev1 (Q (lambda (n) ...))))))
(define fib2
  (ev (Q (ev1 (Q (ev2 (Q (lambda (n) ...))))))))
```

Measuring their run-time shows the expected blowup with each layer of representation, and that even at three levels of nesting it is still usable.

```
> (time (fib 18))
cpu time: 1 real time: 1 gc time: 0
2584
> (time (ev (Q (fib0 18))))
cpu time: 105 real time: 133 gc time: 65
2584
> (time (ev (Q (ev1 (Q (fib1 18))))))
cpu time: 618 real time: 637 gc time: 394
2584
> (time (ev (Q (ev1 (Q (ev2 (Q (fib2 18))))))))
cpu time: 3951 real time: 4131 gc time: 2612
2584
```

To make things more interesting, we can try variations on this theme. For example, we can nest a strict evaluator in the lazy one, and use the `Y` combinator to get recursion:

```
(define ev*1
  (ev* (Q (Y (lambda (ev)
               (lambda (expr)
                 (cond
                   [(term? expr)
                    (let ([subs (term-exprs expr)])
                      (case (term-tag expr)
                        [(lam) (lambda args
                                 (ev (apply (car subs) args)))]
                        [(if)  (ev (if (ev (car subs))
                                       (cadr subs)
                                       (caddr subs)))]
                        [(app) (apply (ev (car subs))
                                      (map ev (cdr subs)))]
                        [else (error 'ev*1 "bad tag")]))]
                   [else expr])))))))
```

The definition of this evaluator is not really strict. In fact, it does not enforce any evaluation strategy — it just inherits it from the language it is implemented in. In this case, this definition is running in `ev*`'s lazy context, which makes the resulting language lazy as well:

```
> (ev* (Q (ev*1 (Q (+ 1 2)))))
3
> (ev* (Q (ev*1 (Q ((lambda (x y) y) (+ 1 "2") 333)))))
333
```

Again, we can repeat this definition to get a third level, then measure the performance of the three levels using `fib` definitions:

```
> (time (ev* (Q (fib*0 18))))
cpu time: 198 real time: 227 gc time: 129
2584
> (time (ev* (Q (ev*1 (Q (fib*1 18))))))
cpu time: 575 real time: 589 gc time: 357
2584
> (time (ev* (Q (ev*1 (Q (ev*2 (Q (fib*2 18))))))))
cpu time: 1186 real time: 1266 gc time: 780
2584
```

It is interesting to note that the blowup factor is much smaller than in the `ev` case. The conclusion is still the same: each evaluator layer increases run-time, but the blowup is small enough to still be practical. (E.g., it is a feasible strategy for implementing DSLs.)

8. HOAS Disadvantages

As mentioned above, HOAS does not come without a price. The two major problems with HOAS representations are well-known:

Exotic terms: we have seen that the functions that are used in HOAS representations are syntactic '`Term`→`Term`' transformers. However, not all of these functions are proper representations — some are not a quotation of any concrete syntax. For example, we can manually construct the following term, which

does hold a '`Term`→`Term`' function:

```
(term 'lam
      (lambda (x)
        (if (equal? x (Q 1)) (Q 2) (Q 3))))
```

but it is not a valid representation — there is no `lambda` term that has this as its quotation. Briefly, the problem is that the function is trying to *inspect* its values (it would have been a valid representation had the whole `if` expression been quoted).

This means that we should not allow arbitrary functions into the representation; indeed, major research efforts went into formalizing types in various ways from permutations-based approaches [15, 16, 17], to modal logic [4, 5] and category theory [14]. In [1], another formalism is presented which is of particular interest in the context of Scheme. It relies on a predicative logic system, which corresponds to certain dynamic run-time checks that can exclude formation of exotic terms. This formalism is extended in [12].

Induction: another common problem is that the representation contains functions (which puts a function type in a negative position), and therefore does not easily lend itself to induction. Several solutions for this problem exist. As previously demonstrated [1], these functions behave in a way that makes them directly correspond to concrete syntax. In Scheme, the quotation macro can add the missing information — add the syntactic information that contains enough hints to recover the structure (but see [1] for why this is not straightforward).

As a side note, it is clear that using HOAS is very different in its nature than using high-level Scheme macros. Instead of plain pattern matching and template filling, we need to know and encode the exact lexical structure of any binding form that we wish to encode. Clearly, the code that is presented here is simplified as it has just one binding form, but we would face such problems if we would represent more binding constructs like `let`, `let*` and `letrec`. This is not necessarily a negative feature, since lexical scope needs to be specified in any case.

9. Conclusion

We have presented code that uses HOAS techniques in Scheme. The technique is powerful enough to make it possible to write a small evaluator that can evaluate itself, and — as we have shown — be powerful enough to model different evaluation approaches. In addition to being robust, the encoding is efficient enough to be practical even at three levels of nested evaluators, or when using lazy semantics. HOAS is therefore a useful tool in the Scheme world in addition to the usual host of meta-level syntactic tools.

We plan on further work in this area, specifically, it is possible that using a HOAS representation for PLT's Reduction Semantics [11] tool will result in a speed boost, and a cleaner solution to its custom substitution specification language. Given that we're using Scheme bindings to represent bindings may make it possible to use HOAS-based techniques *combined* with Scheme macros. For example, we can represent a language in Scheme using Scheme binders, allowing it to be extended via Scheme macros in a way that still respects Scheme's lexical scope rules.

References

[1] Eli Barzilay. *Implementing Direct Reflection in Nuprl*. PhD thesis, Cornell University, Ithaca, NY, January 2006.

[2] Eli Barzilay and Stuart Allen. Reflecting higher-order abstract syntax in Nuprl. In Victor A. Carreño, Cézar A. Muñoz, and Sophiène Tahar, editors, *Theorem Proving in Higher Order Logics; Track B Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2002), Hampton, VA, August 2002*, pages 23–32. National Aeronautics and Space Administration, 2002.

[3] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, NJ, 1986.

[4] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96*, pages 258–270. ACM Press, New York, 1996.

[5] Joëlle Despeyroux and Pierre Leleu. A modal lambda-calculus with iteration and case constructs. In *Proc. of the annual Types seminar*, March 1999.

[6] Robert Bruce Findler. PLT DrScheme: Programming environment manual. Technical Report PLT-TR2006-3-v352, PLT Scheme Inc., 2006. See also: Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme, Journal of Functional Programming, 12(2):159–182, March 2002. http://www.ccs.neu.edu/scheme/pubs/.

[7] Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2006-1-v352, PLT Scheme Inc., 2006. `http://www.plt-scheme.org/techreports/`.

[8] Erik Hilsdale and Daniel P. Friedman. Writing macros in continuation-passing style. In *Scheme and Functional Programming 2000*, page 53, September 2000.

[9] Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *14th Symposium on Logic in Computer Science*, page 204. LICS, July 1999.

[10] Oleg Kiselyov. Macros that compose: Systematic macro programming. In *Generative Programming and Component Engineering (GPCE'02)*, 2002.

[11] Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *International Conference on Rewriting Techniques and Applications*, 2004.

[12] Aleksey Nogin, Alexei Kopylov, Xin Yu, and Jason Hickey. A computational approach to reflective meta-reasoning about languages with bindings. In *MERLIN '05: Proceedings of the 3rd ACM SIGPLAN workshop on Mechanized reasoning about languages with variable binding*, pages 2–12, Tallinn, Estonia, September 2005. ACM Press.

[13] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)*, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[14] Carsten Schürmann. Recursion for higher-order encodings. In *Proceedings of Computer Science Logic (CSL 2001)*, pages 585–599, 2001.

[15] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *Eighth ACM SIGPLAN International Conference on Functional Programming (ICFP 2003), Uppsala, Sweden*, pages 263–274. ACM Press, August 2003.

[16] C. Urban, A. M. Pitts, and M. J. Gabbay. Nominal unification. In M. Baaz, editor, *Computer Science Logic and 8th Kurt Gödel Colloquium (CSL'03 & KGC), Vienna, Austria. Proccedings*, Lecture Notes in Computer Science, pages 513–527. Springer-Verlag, Berlin, 2003.

[17] Christian Urban and Christine Tasson. Nominal reasoning techniques in isabelle/hol. In *Proceedings of the 20th Conference on Automated Deduction (CADE 2005)*, volume 3632, pages 38–53, Tallinn, Estonia, July 2005. Springer Verlag.