Macroexpansion Reflective Tower *

Christian Queinnec†
École Polytechnique & INRIA-Rocquencourt

Abstract

Macros are reflective tools that operate on the representation of programs. Though having been used, and still being useful, for more than thirty years, their semantics and pragmatics are still unclear. This paper proposes a new model to understand the macroexpansion process; this model is based on a reflective tower of macroexpansion engines.

1 Introduction

Macros are definitely a forte of Lisp dialects. They confer upon the powerful users a means to adapt their language to their problems and this is, in our mind, one of the key reasons for Lisp's longevity. Despite the various attempts to standardize macros [Ste90, CR91b] and even if most of them are just quite simple abbreviations, they still represent one of the thorniest problems that occur when porting code between dialects of Lisp: a sure sign of their ambiguous nature. Even in Scheme, macros still have a dark side with respect to their scope, their extent and their meaning *i.e.,* their definition language. This paper proposes an understandable model covering these aspects.

The problem we address and the essence of our solution are presented in section 2. Section 3 details an implementation along with some examples. Variations and applications are explored in section 4. Related works are dealt with in section 5 before the final conclusion.

2 Foundation

Macros were introduced in 1963 by Timothy P. Hart [SG93]. They rely on a unique quality of Lisp: programs are represented as data *i.e.,* S-expressions. This identification allows us to write programs that convert general S-expressions into regular programs. For a given dialect of Lisp, say L, there exists a pure denoter, say *pure-meaning*, that takes programs written in L and returns their denotation. A denotation is an entity that can be evaluated later *i.e.,* interpreted or, compiled then executed. For instance, depending on the technology used for the evaluator, a denotation may be viewed as a bytecode vector, as an immutable copy of the original program, *etc.*

Besides this pure denoter, there is often another facility, say *meaning*, that takes general S-expressions and macroexpands them into pure L programs before denoting these programs with the L pure denoter. Less informally:

$$meaning(\pi) = pure\text{-}meaning(expand(\pi))$$

It may be tempting to consider *meaning* as a pure denoter of another language, say L', based on L with additional syntax. This would be erroneous since *(i)* the L' language evolves as new macros are added, *(ii)* to define a macro is a side-effect and therefore this hypothetical pure denoter would have state which is against denotational tenets. History has shown that users are more interested in the *meaning* facility that offers a macroexpansion service rather than in the bare *pure-meaning* denoter. Moreover, the latter is often

*Revision: 1.20 typeset on February 29, 1996 at 17:03 — To appear in the proceedings of the Reflection'96 conference.

†Laboratoire d'Informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France – Email: `Christian.Queinnec@polytechnique.fr` This work has been partially funded by GDR-PRC de Programmation du CNRS.

absent from Lisp/Scheme systems while the former is generally part of the `eval` facility. Since the goal of *meaning* is to produce a denotation, let's call it a denoter though it is not a pure denoter.

In its full generality, the macroexpansion is a regular and unrestricted evaluation whose particularity is just that it handles pieces of programs and builds programs. The reflective axiom, upon which our tower is founded, is

$$expand(\pi) = eval(\texttt{(expand (quote } \pi\texttt{))})$$

To explicitly rely on an evaluator is not a surprise. It is a logical consequence of the fact that, at macroexpansion time, the introduction of a new macro is made via an S-expression that must be turned, on the fly, into an invocable procedure—an expander—by the macroexpansion engine itself. This capability requires a dynamic evaluation facility.

Although numerous macroexpansion engines were studied [Ste90, KFFD86, BR88, DFH88, CR91a, QP91, DHB93, DPS94, Que94, dM95], their properties in terms of scope and extent are unclear. Since macroexpansion is a disguised evaluation, it has to be performed with some state. Where does this state come from? What happens to it after macroexpansion? Is it shared? These are questions that must be answered to give a precise model of how macros work and to allow users to denote their programs without depending on a hidden and uncontrolled state.

To minimize the interaction with the evaluator, the macroexpansion is often performed as a preprocessing phase. This is consistent with our previous definition for *meaning* and particularly obvious for compilers since the compile-time world is disjoint from the run-time world. If macroexpansion were performed as an external process (as `cpp` does for the C compiler) then the macroexpand-time world would also be different. To clearly separate these multiple worlds is a key for the comprehension of the macroexpansion mechanism as it is also for delivery [DPS94].

2.1 Upper levels

To keep worlds separate means that if a denotation is asked for in some world, there must exist an associated world where the macroexpansion is performed. Just as in the reflective evaluation towers [dRS84, WF88] the associated world is said to stand at the upper level. The macroexpansion tower is built accordingly.

To denote an S-expression at level n requires that it be expanded before.

$$meaning_n(\pi) = pure\text{-}meaning_n(expand_n(\pi))$$

According to our axiom, to expand an S-expression at level n is simply to evaluate a call to the `expand` procedure at the upper level. That is, we define the world where the macroexpansion required by level n is to be performed, to be level $n+1$.

$$expand_n(\pi) = eval_{n+1}(\texttt{(expand (quote } \pi\texttt{))})$$

But to evaluate a program requires it to be denoted before. For the sake of the following explanation, we consider *eval* to be the composition of *meaning* and *run*. The *run* procedure is able to execute the denotation produced by *meaning* and is sufficiently abstract to encompass interpretation and compilation.

$$eval_{n+1}(\pi) = run_{n+1}(meaning_{n+1}(\pi))$$

Therefore, combining all these facts yields:

$$meaning_n(\pi) = pure\text{-}meaning_n(run_{n+1}(meaning_{n+1}(\texttt{(expand (quote } \pi\texttt{))})))$$

The tower is now apparent in its infinite regressive beauty: to obtain the denotation of an S-expression π at level n involves the denotation of the new (and simpler) program `(expand (quote` π`))` at level $n+1$. If π defines and/or uses macros, they will be created at level $n+1$ and stored in the state of the evaluator of level $n+1$, thus they will not clutter level n. This model, see figure 1, is not restricted to its first two levels since macros (at level n) may themselves be defined with the help of macros (of level $n+1$) that do

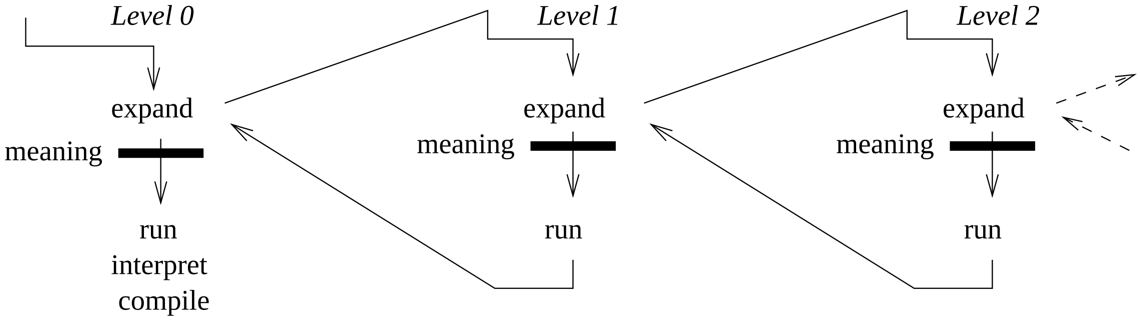

Figure 1: Macroexpansion reflective tower

not need to be visible from other levels. Fortunately, the upper levels of the tower do not use macros and therefore may be shunted since, at these macro-less levels, `(expand (quote` π`))` $= \pi$.

Our model does not impose restrictions on macroexpansion engines provided they can be triggered with a call to the `expand` procedure. Apart from the ground level where the language designer may restrict or forbid access to $expand_0$ and $eval_0$, other levels are regular evaluators with unrestricted evaluation capabilities. In particular, expanders may use external utility procedures or variables and complex macros may be defined by a bunch of files and dynamically loaded. Finally, once the expansion is finished, no dependency remains: the result of the expansion is immediately given back to the pure (stateless) denoter, *pure-meaning*, and turned into a denotation that owes nothing[1] to the upper level.

From now on, we assume the macroexpansion procedure to be the value of the `expand` variable of the upper level. Its result is granted to be the expanded program.

3 Consolidation

We will detail our solution using Scheme but other languages may adopt our proposal provided they allow the building and evaluating of representations of programs. We will first explain how the tower is implemented and then we will focus on the macroexpansion protocol to show that it is expressive enough for the usual needs.

To be independent of any particular evaluation mechanism, we abstract from them with the following five procedures:

- `pure-meaning` takes a program written in pure Scheme *i.e.,* without macros, and yields its denotation (a tuple made of its code and its list of free variables; these two components may be accessed with `meaning-code` and `meaning-free-variables`). Note that `pure-meaning` is a real mathematical function.

- `create-standard-env` returns the standard environment of R^{4}RS Scheme [CR91b]. Since all levels have their own separate unshared environment, the standard environment should be considered as immutable or freshly copied.

- `enrich` extends an environment with the specified variables; the enriched environment is returned. Depending on the implementation, it may be the same updated environment or a new one incorporating new locations for these new variables. With this latter possibility, environments are immutable objects. If a variable already appears in the environment, it is left unchanged; otherwise it is adjoined with an uninitialized state.

[1] Remember that quotations do not mention the value to be returned *per se* but rather describe it [Mul92]. This is obvious when looking at a Scheme to C compiler where quotations are turned into some C code to regenerate some value. Therefore there is no hidden sharing between levels.

- `env-set!` sets the value of a variable within an environment to some value. Its result is unspecified.
- `run` evaluates some code within some environment and returns its resulting value.

First-class environments [RAM84, 84M84, Que94] facilitate the definition of some of the above procedures. They also often exist, at some low-level, within byte-code interpreters. Otherwise, they can be easily simulated on top of an interpreter written in Scheme.

3.1 The tower

Each level of the tower is materialized by a `level` tuple composed of its global environment (accessed with `level-env` and modified by `set-level-env!`) and its upper level (accessed with `level-next`). New levels are allocated with `make-level`. Using the non standard `define-structure`, levels could be defined as:

```
(define-structure level env next)
```

Level 0 is the ground floor where a denotation is first requested, see figure 1. This organization explicitly separates the different worlds. Remember that the following definitions participate in the implementation of the tower, they do not belong to any level of the tower.

Within each level, there is a pure evaluator: it denotes a program, enriches and updates the global environment of the current level with the free variables contained in this program, then runs the associated code at this level and returns its value.

```
(define (do-pure-eval e level)
  (let ((mn (pure-meaning e)))
    (set-level-env! level (enrich (level-env level)
                                  (meaning-free-variables mn)))
    (run (meaning-code mn) (level-env level))))
```

The usual evaluator, often provided to the users of a real Scheme implementation as the value of the `eval` variable, simply expands expressions into programs before purely evaluating them.

```
(define (do-eval e level)
  (do-pure-eval (do-expand e level) level))
```

Expansion is an evaluation at the upper level but, to avoid an infinite construction, we `force` that upper level to exist before. Note that the `expand` variable must be defined as a unary procedure at the upper level. It is useless for `expand` to exist at level 0 since it is sufficient to display macroexpanded results at level 1. Another more subtle trick to break the infinite regression is to use `do-pure-eval` instead of `do-eval` since the call to `expand` is written in pure Scheme and does not need to be expanded.

```
(define (do-expand e level)
  (do-pure-eval `(expand ',e) (force (level-next level))))
```

We can now detail how levels are built. Initially a level is built out of a copy of the standard Scheme environment and a promise to build the upper level if needed. All levels but the ground floor, which may forbid it to its users, require the presence of an `expand` variable which in turn needs a dynamic `eval` facility. This `eval` procedure must operate within the current level global environment; that's why we enrich the current level with the `eval` variable (the `env-set!` primitive takes care of the compatibility of the invocation protocols that may be different between levels as well as different from the implementation). For simplicity, we assume that every level has initially the same macroexpansion engine; the definitional text of this macroexpansion engine, an S-expression, is contained in the `expand-definition` variable and will be commented upon later.

```
(define expand-definition-meaning (pure-meaning expand-definition))
(define (create-level)
  (make-level
   (create-standard-env)
   (delay
     (let ((nextlevel (create-level)))
       ;;install the free variables of the macroexpansion engine
```

```
      (set-level-env! nextlevel (enrich (level-env nextlevel)
                                        (meaning-free-variables
                                         expand-definition-meaning)))
      ;;install the current eval facility
      (env-set! 'eval (level-env nextlevel)
                (lambda (value)
                  (do-eval value nextlevel)))
      ;;and run the code of the macroexpansion engine
      (run (meaning-code expand-definition-meaning)
           (level-env nextlevel))
      nextlevel))))
```

We can now build the ground level and start, for instance, a classical top-level evaluator. With this evaluator, there are no predefined macros at any level but, when created, they persist at their definition level.

```
(define (toplevel)
  (define level0 (create-level))
  (let loop ()
    (display (do-eval (read) level0))
    (loop)))
```

3.2 Expansion

The previous section detailed the tower independently of the macroexpansion mechanism. We now address that topic. We will show that, although very simplistic, its protocol is all we need.

The macroexpansion engine receives an *S-expression* and expands it into a *program* that will be given to a pure denoter. We took some care not to mix these two words in this paper. But it must be observed that the macroexpansion engine works on a syntactically fuzzy and unstructured value; the goal of the macroexpansion engine being to expand that value into another value acceptable by the pure denoter. Therefore a macroexpansion engine must be robust with respect to the S-expressions it has to walk and should not suppose much about their syntax. Conversely one may restrict the input syntax accepted by macros to ensure stronger properties with respect to α-conversion or horizontal captures [Mul94].

The macroexpansion is triggered through the invocation of the unary `expand` procedure with the S-expression to expand as argument. We propose, as a first experiment, to define a small but extensible macroexpansion engine standing on the shoulders of `eval`. Since macroexpansion is the art of weaving computations on more than one level, it is sufficient to let the user access the evaluator of the upper level. More elaborate macroexpansion engines, such as EPS [DFH88], could of course have been directly defined. Note that this definition for `expand` is given as an S-expression to be evaluated at every upper level and therefore is written in a pure macro-less Scheme since it is not expanded.

```
(define bare-expand-definition
  '(define expand                ;Level 1, 2 ...program.
     (lambda (e)
       (if (pair? e)
           (if (eq? (car e) 'eval-in-expansion-world)
               (eval (car (cdr e)))
               e)
           e))))
(define expand-definition bare-expand-definition)
```

This macroexpansion engine behaves as the identity except on S-expressions whose top-level `car` is the keyword `eval-in-expansion-world`. In this case, the expression that follows is considered as a program that must be evaluated at the upper level. The result of this evaluation is returned as the result of the expansion.

Although rather crude as a protocol, it offers the user that ability to define (or `load`) its proper macroexpansion engine replacing this bare one. This might be done with the following level 0 S-expression. Note that this S-expression does not use macros nor derived syntax such as `let` or `and` since it is expanded via the bare expander.

```
(eval-in-expansion-world      ;Level 0, 1, 2 ... programs
 (begin
   (define expand
     (lambda (e)              ;the main function
       (really-expand e global-expand-env)))
   (define macro-env
     (lambda (e)              ;the environment of macros encoded as
       (if (pair? e)          ;a predicate returning an expander.
           (if (eq? (car e) 'eval-in-expansion-world)
               (lambda (e m) (eval (car (cdr e))))
               (if (eq? (car e) 'quote)
                   (lambda (e m) e)
                   #f))
           #f)))
   (define global-expand-env
     (lambda (e)              ;wraps macro-env to always use its last value.
       (macro-env e)))
   (define really-expand
     (lambda (e m)
       ((lambda (expander)
          (if expander
              (expander e m)
              (default-expand e m)))
        (m e))))              ;is there an associated expander?
   (define default-expand
     (lambda (e m)            ;the default expander
       (if (pair? e)
           ((lambda (a)       ;left to right
              (cons a (really-expand (cdr e) m)))
            (really-expand (car e) m))
           e)))
   (define again-izer         ;turn an expander into an equivalent
     (lambda (expander)       ;expander the result of which is reexpanded.
       (lambda (e m)
         (really-expand (expander e m) m))))
   (define form-extend
     (lambda (m key fn)       ;return a new macroenv similar to m
       (lambda (ee)           ;except that it returns fn on a pair
         (if (pair? ee)       ;whose car is key.
             (if (eq? (car ee) key)
                 fn
                 (m ee))
             (m ee)))))
   (define install-macro-form!
     (lambda (name expander) ;extends the global environment of macros with
       (set! macro-env       ;a new macro.
             (form-extend macro-env name expander))
       #f))
   #t))
```

This new engine might have been defined in lieu of the original bare expander. It improves on it since it recursively walks S-expressions; it now allows `eval-in-expansion-world` forms to appear anywhere, it recognizes and ignores quotations. It might itself be improved, for instance to give local variables of **lambda** forms precedence over homonym macros (but this requires **lambda** expressions to have a decipherable list of variables). This macroexpansion engine has been kept simplistic to save space.

The **global-expand-env** and **macro-env** predicates record whether an S-expression has an associated expander. The **install-macro-form!** procedure records macros in the **macro-env** variable and therefore extends the environment of macros, **global-expand-env**, with new global macros. An expander takes an S-expression and an environment of macros and returns a new S-expression. The **again-izer** procedure converts an expander into an expander whose result is re-expanded with the current local environment of macros. This improved engine is still simple but not constraining.

3.3 Local macros

Local macros are quite easy to introduce using the **really-expand** procedure of the improved macroexpansion engine, since the current environment of macros is held in its second variable. This environment of macros will be enriched with some local macros to process the S-expressions in the scope of the local macros. The S-expressions defining the local macros are turned into invocable expanders with the **eval** facility. To save room, the following **let-abbreviation** global macro introduces only one local abbreviation. Note that **let-abbreviation** is defined as a macro at level 0 only; it is of course written with the level 1 language.

```
(eval-in-expansion-world
 (install-macro-form!
  'let-abbreviation  ;(let-abbreviation ((key parameters...) expansion...) scope...)
  (lambda (e m)
    (really-expand
     `(begin ,@(cdr (cdr e)))
     (form-extend m (car (car (car (cdr e))))
                  (again-izer
                   ((lambda (expander)
                      (lambda (ee mm) (apply expander (cdr ee))))
                    (eval `(lambda ,(cdr (car (car (cdr e))))
                             ,@(cdr (car (cdr e))))))))))))
```

For the sake of the following example, we suppose that the improved macroexpansion engine and the **let-abbreviation** macro had been appropriately installed at level 2 with:

```
(eval-in-expansion-world
 (eval-in-expansion-world
  (begin (define expand ...) ...
         (install-macro-form! 'let-abbreviation ...))))
```

Consider the following level 0 example where a local macro maps **progn** onto **begin**:

```
(define foobar
  (lambda (x)
    (let-abbreviation
        ((progn . body)
         (let-abbreviation ((sequence . body) `(begin ,@body))
           (sequence (display "Use begin instead of progn!")
                     `(begin ,@body))))
      (progn x (list x)))))
```

When the local **progn** macro is defined at level 0, its associated expander is evaluated at level 1. During this evaluation, its body is expanded and the **sequence** local macro is defined at level 1 with an associated expander created at level 2. The body of the expander associated with **progn** makes use of the **sequence** macro and is finally expanded into:

```
(lambda body
  (begin (display "Use begin instead of progn!")
         `(begin ,@body)))
```

At the end of the macroexpansion nothing remains at any level since all implied macros were local and disappeared at the end of their associated scope. This example uses a macro to define a macro. `sequence` is invisible except for level 1 while `progn` is invisible except for level 0.

3.4 Global macros

The second facility we introduce will allow us to define global macros *i.e.,* macros that persist at the level they were defined. This `define-abbreviation` facility is implemented as a regular macro: it builds a program describing the precise expander that will be triggered when the keyword is recognized. This program is dynamically evaluated into an invocable expander recorded in `macro-env`, the global environment of macros.

```
(eval-in-expansion-world
 (install-macro-form!
  'define-abbreviation  ;(define-abbreviation (key parameters...) expansion...)
  (lambda (e m)
    (install-macro-form!
     (car (car (cdr e)))
     (again-izer
      ((lambda (expander)
          (lambda (ee mm) (apply expander (cdr ee))))
       (eval `(lambda ,(cdr (car (cdr e)))
                ,@(cdr (cdr e))))))))))
```

Here is the previous example now rewritten with `define-abbreviation`. As before, we suppose `define-abbreviation` to be present at level 1. Recall that expansion is performed from left to right to allow the definition of a macro and its subsequent use in the same `begin` form. Also note that `define-abbreviation` requires its result to be re-expanded.

```
(define barfoo
  (lambda (x)
    (define-abbreviation (progn . body)
      (define-abbreviation (sequence . body)
        `(begin ,@body))
      (sequence (display "Use begin instead of progn!")
                `(begin ,@body)))
    (progn (list x))))
```

The `progn` macro is created at level 0 and its associated expander is created at level 1. During this evaluation, its body is expanded at level 1 and defines the `sequence` macro whose expander is created at level 2. After the macroexpansion, the `progn` macro persists at level 0 (and only at that level) while the `sequence` macro persists at level 1. This exhibits a difference between the languages of level 0 (Scheme + `progn`) and level 1 (Scheme + `sequence`). Therefore one may prepare a level to define the language for the definition of a macroexpander that will be used for the definition of another language at the lower level.

4 Perspectives from the tower

Multiple incarnations of Scheme exist; they all have peculiar macro systems. We appreciated that variability when porting the Meroon object system on 12 different Scheme incarnations, but it is not our intention to describe or to compare these systems but only to show that our model may be successfully applied.

Interpreters often mix all levels of our tower into a single one. However, they may adopt our macroexpansion tower *i.e.,* a tower of independent evaluators linked by a macroexpansion relationship. This would have the additional benefit of easing the introduction of interactive modular programming such as [Tun92].

Pure compilers generally offer the first two levels but show their divergences on many points such as macros creating macros, macros loading files, *etc.* They may adopt our macroexpansion tower and this will standardize the effects, scope and extent of macros without constraining the macroexpansion algorithm. This last fact might be appreciated by fans of standardization.

The worst case concerns interpreters/compilers providing a `compile-file` facility. The level 0 of the compiler may or may not be the same as the level 0 of the interpreter therefore blurring the scope and extent of macros defined for one or the other.

4.1 Coalesced levels

Another interesting view is possible where the tower is restricted to its ground level with all upper levels coalesced into a single one. This structure has the advantage that when a macro is defined, it is immediately available at any upper level: this saves the independent management of multiple levels while at the same time retaining the separation of the ground level and all upper levels.

This last variation is easily implemented with `create-level-all-the-same` instead of `create-level`.

```
(define (create-level-all-the-same)
  (make-level
   (create-standard-env)
   (letrec ((nextlevel (make-level
                        (create-standard-env)
                        (delay nextlevel))))
     (set-level-env! nextlevel (enrich (level-env nextlevel)
                                       (meaning-free-variables
                                        expand-definition-meaning)))
     (env-set! 'eval (level-env nextlevel)
               (lambda (value)
                 (do-eval value nextlevel)))
     (run (meaning-code expand-definition-meaning)
          (level-env nextlevel))
     (delay nextlevel))))
```

4.2 Continuations

Nothing prevents level 1 from capturing continuations and this may have interesting properties. A macro may suppose some properties to be true for a certain kind of expansion, it therefore captures the continuation to which it returns the chosen expansion. If it happens, while expanding the rest of the S-expression, that the assumed properties are not true then the macroexpansion engine may decide to change its previous expansion and reinvoke the captured continuation with a more appropriate expansion.

The following example defines a `counting` macro that counts the number of occurrences of a given variable in some scope and makes these occurrences expand into forms revealing their order of occurrences.

```
(eval-in-expansion-world
 (install-macro-form!
  'counting
  (lambda (e m)
    ((lambda (number)
       (call/cc
        (lambda (return)
          ((lambda (process)
             (set! process
                   (lambda (i)
                     (really-expand
                      `(begin ,@(cdr (cdr e)))
                      (form-extend
                       m (car (car (cdr e)))
                       (lambda (ee mm)
                         (set! i (- i 1))
                         (if (> i 0)
```

```
                           `(quote (,i / ,(- number 1)))
                           (begin (set! number (+ number 1))
                                  (return (process number)) ) ) ) ) ) ) )
              (process number) )
             'process ) ) ) )
       1 ) ) ) )

(counting (a) (list (a) (a) (list (a))))
≡ (list '(3 / 3) '(2 / 3) (list '(1 / 3)))
```

This is not a property *per se* of the tower but this may be done safely since the relationship between the expander and the denoter are now known and thus manageable. Observe also that the order of expansion is crucial here.

5 Related work

Our macroexpansion tower is orthogonal to any particular macroexpansion algorithms such as Expansion Passing Style (EPS) [DFH88]. It is also unrelated to the concept of hygienic macros [KFFD86, CR91a, DHB93] that provide a good solution to the specific problem of name collision. Nothing prevents the landlord of a macroexpansion tower where the `expand` variable is mutable to install, at any level, the kind of macroexpansion engine he is dreaming of.

It is amusing to see that the old view of macros where `(foo` π_1 π_2 `...)` was considered as a procedure applied on the unevaluated text of its arguments and whose result is evaluated, is somewhat true in our model since the previous macro call is approximately `(eval-in-expansion-world (foo '`π_1 `'`π_2 `...))` (supposing `foo` to hold the appropriate expander) except that the used evaluator is not the one of level 0.

The high-level macroexpansion engine defined by Scheme is based on a pattern language that can be directly interpreted by the engine itself without requiring a dynamic evaluation facility. Unfortunately its power is limited since it cannot handle arithmetic, for instance. This is not inconvenient for a vast majority of macros but this excludes complex macros such as macros defining an object system which must manage a state: a tree of inheritance.

We think that the power of macros should not be restricted and any possible computation on program fragments should be possible. This is one of the preferred ways to implement new language features and the macroexpansion mechanism should not rule it out: this is acknowledged by the low-level macroexpansion engine of Scheme that allows expanders to be defined by unrestricted procedures. Alas, in this latter case, nothing prevents these expanders from using the resources of the programs they are expanding thus mixing run-time and expand-time (and/or compile-time depending on the nature of the evaluator). A possible solution may be to extend the proposed `eval` feature of Scheme to take into account the macroexpansion engine to use as a possible third argument.

To clearly separate the multiple worlds is discussed in [QP91, PNB93, DPS94]. The key is to define modules with static directives telling which macros (from other modules) are imported and which macros (defined in the current module) are exported. Our model is more oriented towards dynamic evaluation but, contrary to [DPS94], allows macros to be defined and used on the fly *i.e.,* in the same module. Our tower is founded on the explicit existence of an `eval` facility whereas the solution of [DPS94] enriches (at link-time) the compiler with the modules providing the imported macros.

6 Conclusion

We proposed a tower of macroexpansion engines providing a clear understanding of the macroexpansion process for dynamic languages *i.e.,* providing unrestricted computation capabilities on the representation of programs. It clearly separates the multiple worlds involved in the expansion and the evaluation of S-expressions. The model may also be used to describe existing Scheme implementations and works the same for other languages. A less abstract version of the tower is presented in [Que94] with an extended construction to capture meanings (thus allowing for hygienic macros) but requiring some cooperation (mutual understanding of some data structures as well as interleaving the two computations) between the expansion engine and the denoter.

7 Acknowledgment

Thanks to Luc Moreau, Pierre Parquier, Shriram Krishnamurthi, Daniel Friedman and the anonymous referees for their stimulating comments.

Bibliography

[84M84] *Mit Scheme Manual, Seventh Edition.* Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Mass., September 1984.

[BR88] Alan Bawden and Jonathan Rees. Syntactic closures. In *Proceedings of the 1988 ACM Symposium on LISP and Functional Programming*, Salt Lake City, Utah., July 1988.

[CR91a] William Clinger and Jonathan Rees. Macros that work. In *POPL '91 – Eighteenth Annual ACM symposium on Principles of Programming Languages*, pages 155–162, Orlando, (Florida USA), January 1991.

[CR91b] William Clinger and Jonathan A Rees. The revised^4 report on the algorithmic language Scheme. *Lisp Pointer*, 4(3), 1991.

[DFH88] R. Kent Dybvig, Daniel P. Friedman, and Christopher T. Haynes. Expansion-passing style: a general macro mechanism. *International journal on Lisp and Symbolic Computation*, 1(1):53–76, June 1988.

[DHB93] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *International journal on Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[dM95] Antoine Dumesnil de Maricourt. *Macro-expansion en Lisp, sémantique et réalisation.* Thèse d'université, Université Paris 7, Paris (France), June 1995.

[DPS94] Harley Davis, Pierre Parquier, and Nitsan Séniak. Talking about modules and delivery. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programmming*, pages 113–120, Orlando (Florida USA), June 1994. ACM Press.

[dR87] Jim des Rivières. Control-related meta-level facilities in Lisp. In P. Maes and D. Nardi, editors, *Workshop on Meta-Level Architecture and Reflection*, Alghiero, Sardinia (Italy), October 1987. North Holland.

[dRS84] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pages 331–347, Austin, Texas, August 1984. ACM Press.

[Gra93] Paul Graham. *On Lisp, Advanced Techniques for Common Lisp.* Prentice-Hall, 1993.

[KFFD86] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. *Symposium on LISP and Functional Programming*, pages 151–161, August 1986.

[Mul92] R. Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14, No. 4:589–615, 1992.

[Mul94] R. Muller. A staging calculus and its application to the verification of translators. In *POPL '94 – Twenty-first Annual ACM symposium on Principles of Programming Languages*, pages 389–396, 1994.

[PNB93] Padget, J.A., Nuyens, G., and Bretthauer, H. An overview of EuLisp. *Lisp and Symbolic Computation*, 6(1/2):9–98, 1993.

[QP91] Christian Queinnec and Julian Padget. Modules, macros and Lisp. In *Eleventh International Conference of the Chilean Computer Science Society*, pages 111–123, Santiago (Chile), October 1991. Plenum Publishing Corporation, New York NY (USA).

[Que94] Christian Queinnec. *Les langages Lisp.* InterÉditions, Paris (France), 1994. ISBN 2 7296 0549 5, 61 2448 1, English version soon available from Cambridge University Press.

[RAM84] Jonathan A. Rees, Norman I. Adams, and James R. Meehan. *The T Manual, Fourth Edition.* Yale University Computer Science Department, January 1984.

[SG93] Guy L. Steele, Jr. and Richard P Gabriel. The evolution of Lisp. In *The Second ACM SIGPLAN History of Proramming Languages Conference (HOPL-II)*, pages 231–270, Cambridge (Massachusetts, USA), April 1993. ACM SIGPLAN Notices 8, 3.

[Ste90] Guy L. Steele, Jr. *Common Lisp, the Language.* Digital Press, Burlington MA (USA), 2nd edition, 1990.

[Tun92] Sho-Huan Simon Tung. Interactive modular programming in Scheme. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 86–95, San Francisco, USA, June 1992.

[WF88] Mitchell Wand and Daniel P. Friedman. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation*, 1:11–37, 1988. Reprinted in *Meta-Level Architectures and Reflection* (P. Maes and D. Nardi, eds.) North-Holland, Amsterdam, 1988, pp. 111–134.