Compiling Syntactically Recursive Programs *

Christian Queinnec[†]
École Polytechnique & INRIA–Rocquencourt

Abstract

Lisp programs are traditionally represented by S-expressions. Side-effecting S-expressions allows to easily build cyclic S-expressions which can be turned, through macroexpansion, into *syntactically recursive* programs i.e., programs which do not have a finite DAG-like representation. We do not know of any compiler able to compile such syntactically recursive programs. Yet, interpreters do not have these problems when evaluating these sort of programs.

The paper proposes a program transformation that translates any possibly syntactically recursive program into an equivalent regular program. This result establishes *(i)* that any S-expression can be considered as a program irrespectively of its possibly cyclic structure, *(ii)* the equivalence of compilers and interpreters with respect to any S-expression as well as *(iii)* the possibility to implement a dynamic evaluation facility i.e., the `eval` function, without any interpreter and with the sole use of a compiler.

1 Introduction

Lisp is famous for its *program* = *data* rule. S-expressions i.e., computable values, represent Lisp programs. Macros take advantage of this representation and offer the opportunity to compute on the representation of programs. Macros have a wide range of use from simple abbreviations to more complex program transformations. On another hand, an evaluation facility often exists as a regular Lisp function `eval` which, given a S-expression considered as a program, returns its value. These features are invaluable when implementing new languages yet, they present some drawbacks: the semantics of `eval` violates the tenets of denotational semantics as shown in [MP80], and macros introduce numerous problems accounted in, at least, [KFFD86, DFH88, QP90, CR91a, QP91].

We concentrate on a single problem which seems to almost never receive much attention apart from Stoy [Sto77] who solved it theoretically. S-expressions can be built to contain cycles. This was exploited in the old interpreter-based Lisp folklore to compute the factorial of a number with a *syntactic recursion* as shown below:

```
(let ((n 4))
  #1=(if (= n 1) 1
         (* n ((lambda (n) #1) (- n 1))) ) )          (1)
```

In this example, we use the Common Lisp[Ste90] notation for cyclic S-expressions: `#1=` labels by `1` the expression that follows the equal sign; the expression labeled by `1` can be inserted in all the places marked as `#1#`. Example (1) computes the value of the factorial of `4`. Another example is the transformation of a `prog` (or `tagbody` in Common Lisp parlance) form to replace all inner `go` forms by the associated code. This speeds up the interpretation of these `go` forms.

For now, we will not consider programs using macros or `eval`. Cyclic S-expressions correspond to programs that are said to be *syntactically recursive*. In a compiler-based Lisp system, syntactically recursive programs make compilers loop since these programs do not have a finite DAG-like representation. However syntactically recursive programs do have a semantics. Pragmatically, interpreters can evaluate them and, on a theoretical ground, Stoy proved it [Sto77, pp. 182–189] as briefly recalled below.

*January 20, 1993 at 18:58 — *Revision* : 1.21. Submitted to Lisp Pointers.

[†]Address: Laboratoire d'informatique de l'École Polytechnique (URA 1439), 91128 Palaiseau Cedex, France. Email: `queinnec@poly.polytechnique.fr`, this work has been partially supported by Greco de Programmation.

Let **Program** be the *syntactic domain* of programs. Example (1) can be specified as the least fixed point of the syntactic function Φ defined on **Program** $\rightarrow$ **Program** as:

$\Phi(\pi)$ `= (if (= n 1) 1 (* n ((lambda (n)` π`) (- n 1))))`

This fixed point can be computed as the limit of the suite:

π_0 `=` $\perp$
π_{n+1} `= (if (= n 1) 1 (* n ((lambda (n)` π_n`) (- n 1))))`

Despite its strange aspect this limit exists since it is a general property of domains and particularly of the syntactic domain **Program**. A language is usually defined by means of syntactic equations. Any equation can be turned into a summand that contributes to define the **Program** syntactic domain. For example, among other possibilities, a program can be an assignment or an alternative:

```
<program> ::= (set! <identifier> <program>)
           |  (if <program> <program> <program>)
           |  ...
```

In that case **Program** contains the corresponding summands and looks like:

$$\begin{array}{lcl} \textbf{Program} & = & \textbf{Id} \times \textbf{Program} \\ & + & \textbf{Program} \times \textbf{Program} \times \textbf{Program} \\ & + & \ldots \end{array}$$

This domain contains all tree-like regular programs but also the limits of the syntactic equations i.e.the less known syntactically recursive programs.

The semantics of Lisp, without macros and `eval` feature, is usually expressed by a valuation function $\mathcal{E}$ mapping programs onto their denotation. Moreover $\mathcal{E}$ being monotonic and continuous, any element of **Program**, and particularly syntactically recursive programs, can be denoted. Syntactic recursions are transformed into semantic recursions by means of:

$$\bigsqcup_{n\to\infty}(\mathcal{E}(\pi_n)) = \mathcal{E}(\bigsqcup_{n\to\infty}(\pi_n)) \quad [1]$$

The goal of this paper is to exhibit a practical method to denote syntactically recursive programs. We propose a program transformation that translates any syntactically recursive program into an equivalent non syntactically recursive program, the denotation of which being easy to compute with usual method i.e., it can be compiled by any regular compiler. This paper establishes *(i)* that any S-expression (computable value) can be considered as a program irrespectively of its possibly cyclic structure, *(ii)* that compilers and interpreters have equal power, *(iii)* that a dynamic evaluation facility i.e., the `eval` function, does not require an interpreter to be implemented and can use a regular compiler as well.

The rest of the paper is organized as follows. Section 2 exposes the possible syntactic recursions in the Scheme language while section 3 presents the translation rules to convert a possibly syntactically recursive program into a regular one. Some examples are given throughout the paper.

2 Syntactical recursions in Scheme

We present our result in the framework of a real and complete language: Scheme [CR91b]. Scheme is an applied λ-calculus with side-effects, assignments and first-class continuations. However these features are irrelevant with respect to our technique. This section defines where syntactic recursions can occur and gives some new examples of syntactically recursive programs.

The places where syntactic recursions may occur are driven by the grammar of Scheme, see figure 1. To simplify the presentation of our program transformation, we exclude dotted variables in <*variables*> as well as all derived syntaxes.

We do not further detail quotations (< *datum* >) that, even cyclic, are relatively easy to deal with. <*variables*> may have different forms in Scheme however, the semantics of Scheme [CR91b], imposes that instances of < *variables* > should not contain a same variable more than once. This excludes syntactic recursions within lists of variables which can only be finite lists of all different identifiers. To add the other forms of <*variables*> allowed by the complete grammar of Scheme does not alter this property.

<*program*>	::=	<*identifier*>
	|	`(if` <*program*> <*program*> <*program*>`)`
	|	`(set!` <*identifier*> <*program*>`)`
	|	`(quote` <*datum*>`)`
	|	`(begin` <*body*>`)`
	|	`(lambda` <*variables*> <*body*>`)`
	|	`(`<*terms*>`)`
<*variables*>	::=	`(`<*identifier*>*`)`
<*body*>	::=	<*program*>
	|	<*program*> <*body*>
<*terms*>	::=	<*program*>
	|	<*program*> <*terms*>

Figure 1: Grammar of Scheme (simplified on <*variables*>)

Observe that we distinguish sequences of expressions composing the body of a sequence (<*body*>) from the arguments of an application (<*terms*>) since they have quite different semantics.

Roughly, a syntactic recursion exists when an instance of <*program*>, <*terms*> or <*body*> appears in itself. In a syntactic recursion, we differentiate the *head*, labeled by `#`*n*`=` in the examples, from the (possibly multiple) *tails*, labeled by `#`*n*`#`. We therefore distinguish three kinds of syntactic recursions through <*program*>, <*terms*> or <*body*>. Note that S-expressions, being mainly made of dotted pairs, offer more radical sharing possibilities i.e. a dotted pair can be simultaneously an instance of <*terms*> and <*body*> as shown in figure 2.

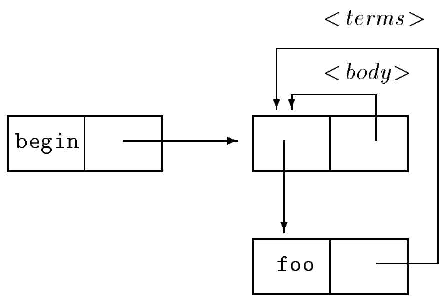

Figure 2: <*terms*> and <*body*> on a single dotted pair: `(begin (foo (foo...) ...) ...)`

The **Program** domain contains all regular programs but many other interesting ones such as:

```
#1=(f #1#)                                          (2)
(begin . #1=((display (eval (read))) . #1#) )       (3)
```

Example (2) mimics the implementation of the `Y` combinator in a graph-reduction context while example (3) implements a toplevel loop using a syntactic recursion on an instance of <*body*>. These are very simple syntactic recursions since their head and tail occur in similar lexical contexts. A more complex example is, for instance:

```
#1=(if (= n 1) 1                                    (4)
       (* p ((lambda (p) (set! n (- n 1)) #1#)
             n )) )
```

Example (4) computes $\mathtt{p}^{\frac{\mathtt{n}!}{2}}$. Observe that, due to the syntactic recursion, the variable `p` that appears as first argument of the multiplication is simultaneously free and bound.

3 Compilation rules for Scheme

According to equation [1], syntactic recursions can be translated into semantic recursions. We adopt a similar method: wherever a syntactic recursion is discovered, it is converted into an appropriate `letrec` form. The general technique is as follows:

1. find all heads and tails of syntactic recursion for every syntactic category: <*program*>, <*body*> and <*terms*>;
2. identify all variables (read or write references) that simultaneously appear free and bound considered from the heads of syntactic recursions;
3. convert any head into a `letrec` form introducing a local recursive function, then translate any associated tail into a call to this local function.

This translation removes all syntactic recursions in favor of simple local recursions. The translated program now has a finite DAG-like representation. The rest of this section details the translation rules.

To detect heads and associated tails of syntactic recursion is straightforward. A simple syntax-driven walk of the program is performed. The list of all visited syntactic nodes (instances of <*program*>, <*body*> and <*terms*>) is maintained so that termination of the walk as well as the detection of heads and tails is easy.

The usual notion of free or bound variable is still decidable in <*program*>, although one has to take care of the possibly cyclic nature of the representations of <*program*>, <*body*> and <*terms*>. A simple way is to memorize the list of the already inspected syntactic nodes to avoid asking a same question more than once. Nonetheless the major difference is that a single occurrence of a variable might be simultaneously free and bound as shown in example (4). Also observe that the freeness and boundness of a variable depends on the viewpoint. Still in example (4), the variable `p` is bound and not free when considered from `#2#` which is a simple unfolding of `#1#`.

```
(if (= n 1) 1                                                    (5)
    (* p (#2=(lambda (p) (set! n (- n 1)) (if (= n 1) 1
                                              (* p (#2# n)) ))
          n )) )
```

The remaining steps are the core of the technique. Of course, the transformation rule depends on the syntactic nature of the syntactic recursion. Let us begin with syntactic recursions on <*program*>.

Rule 1 :

let π be a head of a syntactic recursion,
let $v_1, \ldots, v_n$ be the variables that are simultaneously free and bound in π,
let g not free in π and different from any v_i,
Then rewrite π into `(letrec ((`g `(lambda (`$v_1 \ldots v_n$`)` $\overline{\pi}$`)))` Π`)`
with Π `= (`g `(capture` v_1`) ... (capture` v_n`))`
and with $\overline{\pi} = \mathcal{T}(\pi, \{v_1 \ldots v_n\})$ where $\mathcal{T}$ is defined as:

$$\begin{array}{rcll}
\mathcal{T}(v_i, \{v_1, \ldots v_i, \ldots v_n\}) & \rightarrow & \texttt{(capture-ref } v_i\texttt{)} & \\
\mathcal{T}(\nu, V) & \rightarrow & \nu & \nu \notin V \\
\mathcal{T}(\texttt{(set! } v_i\ \pi\texttt{)}, V) & \rightarrow & \texttt{(capture-set! } v_i\ \mathcal{T}(\pi, V)\texttt{)} & \\
\mathcal{T}(\texttt{(set! } \nu\ \pi\texttt{)}, V) & \rightarrow & \texttt{(set! } \nu\ \mathcal{T}(\pi, V)\texttt{)} & \nu \notin V \\
\mathcal{T}(\texttt{(if } \pi_1\ \pi_2\ \pi_3\texttt{)}, V) & \rightarrow & \texttt{(if } \mathcal{T}(\pi_1, V)\ \mathcal{T}(\pi_2, V)\ \mathcal{T}(\pi_3, V)\texttt{)} & \\
\mathcal{T}(\texttt{(quote } \varepsilon\texttt{)}, V) & \rightarrow & \texttt{(quote } \varepsilon\texttt{)} & \\
\mathcal{T}(\texttt{(begin } \pi_1 \ldots \pi_n\texttt{)}, V) & \rightarrow & \texttt{(begin } \mathcal{T}(\pi_1, V) \ldots \mathcal{T}(\pi_n, V)\texttt{)} & \\
\mathcal{T}(\texttt{(lambda (}\ldots v_i \ldots\texttt{) } \pi\texttt{)}, V) & \rightarrow & \texttt{(lambda (}\ldots v_i \ldots\texttt{) } \mathcal{T}(\pi, V - \{v_i\})\texttt{)} & \\
\mathcal{T}(\texttt{(lambda } \nu^*\ \pi\texttt{)}, V) & \rightarrow & \texttt{(lambda } \nu^*\ \mathcal{T}(\pi, V)\texttt{)} & \nu^* \cap V = \emptyset \\
\mathcal{T}(\texttt{(}\pi_0\ \pi_1 \ldots \pi_n\texttt{)}, V) & \rightarrow & \texttt{(}\mathcal{T}(\pi_0, V)\ \mathcal{T}(\pi_1, V) \ldots \mathcal{T}(\pi_n, V)\texttt{)} & \\
\mathcal{T}(\texttt{\#1\#}, V) & \rightarrow & \Pi &
\end{array}$$

where V is any set of variables.

The head of a syntactic recursion is translated into a `letrec` form which introduces a local recursive function that represents the *body* of the syntactic recursion. Any associated tail is translated into a call to this local function. Variables that only appear free in the syntactic recursion are not problematic, they can be safely read or written from all places from within the syntactic recursion. Variables which are only bound can also be regularly handled. A variable which is both bound and free refers to multiple bindings, the local recursive function that corresponds to a body of a syntactic recursion is therefore parameterized with respect to these variables. The `capture` syntax captures bindings which can be read or written via `capture-ref` and `capture-set!`. The head of the syntactic recursion is then translated into an invocation of the local recursive function on the captured current bindings of these variables which are simultaneously free and bound; tails are similarly translated. Bindings are captured by means of the `capture` syntax:

```
(define-syntax capture
  (syntax-rules ()
    ((capture ?var)
     (lambda (msg . val)
       (case msg ((get) ?var)
                 ((set!) (set! ?var (car val))) ) ) ) ) ) )
(define (capture-ref b) (b 'get))
(define (capture-set! b v) (b 'set! v))
```

The translation of example (4) might ease the reader to grasp this transformation rule.

```
(letrec ((loop111                     ;local function associated to the syntactic recursion
          (lambda (p)
            (if  (= n 1) 1
              (* (capture-ref p)        ; p was simultaneously bound and free
                 ((lambda (p)
                    (begin (set! n (- n 1))
                           (loop111 (capture p)))) ;capture current p
                  n ) ) ) ) ))
        (loop111 (capture p)) )                    ;capture current p
```

The syntactic recursion is associated to function `loop111`, defined and invoked at the head of the syntactic recursion. This function is given the closure representing the current binding of the free variable `p`. The body of `loop111` is translated as explained above so the first argument of the multiplication, a free reference to `p`, is converted into `(capture-ref p)`. A tail is found and converted into a call to `loop111` with the current binding of `p`. The other variable of the example, `n`, is always free and so does not need to be translated.

More precisely, the variables that are to be translated when rewriting a syntactic recursion are those that are free and simultaneously bound between the head and a tail. In example (4), the variable `p` is bound with a scope including `#1#` and therefore must be captured.

Rule 1 takes care of syntactical recursions through <*program*> as well as syntactic recursions through <*body*>, as shown by the translation of example (3):

```
(letrec ((loop132 (lambda ()
                    (begin (display (eval (read)))
                           (loop132) ) )))
        (loop132) )
```

Example (2) is also a syntactic recursion through a <*terms*>. The original `#1=(f #1#)` program never finishes since Scheme is not a lazy language. The translation is:

```
(letrec ((loop131 (lambda ()
                    (f (loop131)) )))
    (loop131) )
```

Syntactic recursion through <*terms*> are somewhat different since they introduce applications with an infinite number of arguments. Consider the following example:

```
(call/cc (lambda (exit)
           (foo . #1=( (if (= n 1) (exit r)                              (5)
```

```
                        (begin (set! r (* r n))
                               (set! n (- n 1)) ) )
                    . #1# )) ))
```

The function `foo` has an infinite number of arguments, all of which must be computed before `foo` can be applied. Moreover the semantics of Scheme does not impose a precise evaluation order. The function `foo` will never be invoked but `exit` will be eventually applied on the value $\texttt{r}(\texttt{n}!)$. The trick is to introduce a local recursive function to compute the arguments that belong to the syntactic recursion and to use the following rule.

Rule 2 :

$\forall \pi, \alpha, \beta \ldots \in$ **Program**
$(\phi\ \alpha\ \beta\ \ldots) \equiv (\texttt{apply}\ \phi\ (\texttt{cons}\ \alpha\ (\texttt{cons}\ \beta\ \ldots)))$
Functions `cons` and `apply` are assumed to be globally visible.

Rule 2 replaces a static application by a dynamic one. This avoids the problem of a static infinite number of arguments in favor of a possibly run-time infinite list of arguments. Combining rule 1 and 2 allows to translate example (5) into:

```
(call/cc (lambda (exit)
           (apply foo
                  (letrec ((loop145
                            (lambda ()
                              (cons (if (= n 1) (exit r)
                                        (begin (set! r (* r n))
                                               (set! n (- n 1)) ) )
                                    (loop145) ) ) ))
                           (loop145) ) ) ))
```

Observe that `loop145` was introduced to compute the infinite list of the arguments to be submitted to `foo`. Also note that the order of evaluation is still undefined and only imposed by the evaluation order of the arguments of `cons` in `loop145`. Rule 2 converts a static application into a dynamic application and this might introduce some questions as: if the arity of `foo` was known to be finite, then example (5) is statically erroneous while its translation is not.

4 Related work and conclusion

Stoy is, to our knowledge, the first to pose the problem of syntactic recursions and to show that they can be denoted. The proof is only theoretical and non constructive. Related investigations on the `Y` combinator has been inquired by [Fel87], [Dil88, page 84] and [FRW90]. Our work contrasts with theirs since it deals with a real language and proposes an effective program transformation that transforms syntactically recursive programs into regular programs with finite DAG-like representation. On the other hand, we do not present proofs here.

It seems obvious from our result that syntactically recursive programs can always be avoided since equivalent non syntactically recursive programs can be written directly by the user. On the other hand, we show that any computable value even cyclic can be turned into a program which can be compiled with a regular compiler. It is therefore possible to use this transformation:

after macro-expansion to prevent a compiler from looping[1] when analyzing a cyclic program,

before the `eval` function so that it can accept any kind of dynamically computed value and still be able to compile it as a program that can be dynamically executed.

[1] Of course there are many other reasons which can make a compiler loops.

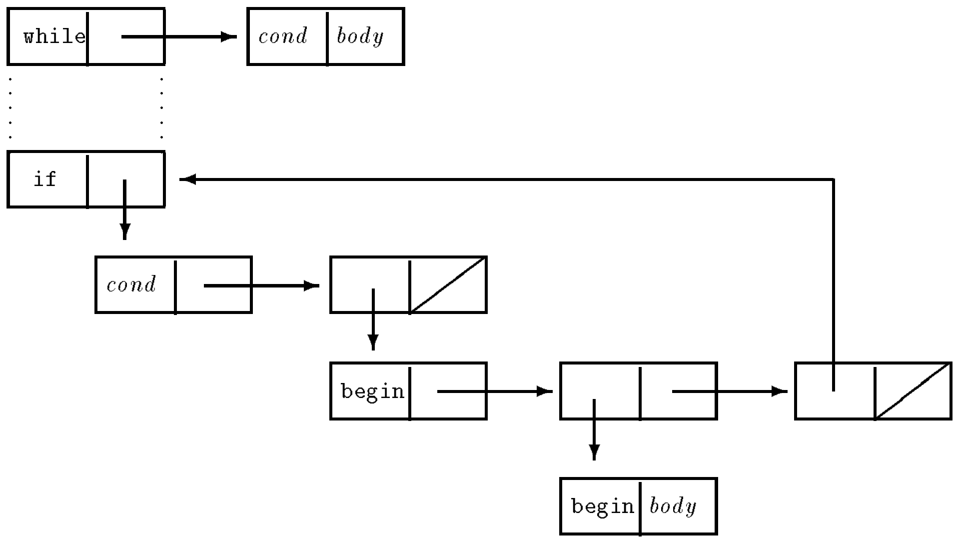

Figure 3: A displacing WHILE macro

Interpreters can then be completely avoided in favor of compiler-based systems; self-modifying program (see [Que90] for an adaptation of CoreWar [Dew84] to Scheme), dynamic macros that are expanded again and again and are not required to deliver similar results for equivalent invocations still need other computation models.

A benefit of our technique is that it is particularly useful when programs are automatically (macro-)generated and where some sharing in the representation may occur without being noticed. The simplest example (but pedagogically very difficult to explain) is probably the `WHILE` macro, often written by naïve students as:

```
(define (cyclic-while-expander call)
  (let ((condition (cadr call))
        (body (cddr call)) )
    `(if ,condition (begin (begin . ,body) ,call)) ) )
(macro WHILE cyclic-while-expander)
```

If `WHILE` is a displacing[2] macro, it creates a cyclic program (thus making the compiler to loop) otherwise it makes the macro-expansion to never finish. Our technique rehabilitates displacing macros. If macro-expansion terminates then its result can be denoted. To be denoted does not imply to compute useful results: syntactically recursive programs that do not loop are rare.

Another related paper is [Bak92] where Baker proposed a semantics for the inlining of recursive functions. He introduced a directive `(declare (inline-`k f`))` where f is the name of the function to inline and, k the integer up to which calls to f should be unfolded. We can provide an alternate semantics where a to-be-inlined invocation is open-coded with recursive calls, if any, cyclically represented. Suppose, for example, that we want to inline a call to the regular (and recursive) factorial function such as:

```
(with-inlining-of (fact)
  (+ 1 (fact p)) )
```

This excerpt can be translated (paying attention to hygien i.e. α-renaming and access to global variables) into:

```
(+ 1 ( #1=(lambda (n) (if (= n 1) 1 (* n (#1# (- n 1))))) p))
```

Eventually our transformation rewrites this fragment into:

[2] A displacing macro stores the resulting expansion in the dotted pair representing the original macro call, see figure 3.

```
(+ 1 ((letrec ((loop138 (lambda ()
                            (lambda (n)
                              (if (= n 1) 1 (* n ((loop138) (- n 1)))) ) )))
          (loop138) )
      p ) )
```

Standard techniques such as those mentioned in [Roz92] can be used to efficiently compile these pieces of code especially if good properties such as tail recursion appear in them.

Independently of its possible use, this paper describes a program transformation technique which takes programs without macros but with possibly cyclic syntactic representation and converts them into regular DAG-like finite representations. The transformation identifies all the places where occur syntactic recursions, computes the free and bound variables that might be confused and transform them into local `letrec` forms, coding appropriate variables with closures. This technique can be adapted to other languages as well; the syntactic representation does not need to use dotted pairs, side-effects are not mandatory and only lexical scoping, local recursion and first-class closures are required for the transformation.

Acknowledgements

I am pleased to thank Matthias Felleisen and the numerous anonymous referees whose encouragements and suggestions improved this paper.

Bibliography

[Bak92] Henry G Baker. Inlining semantics for subroutines which are recursive. *SIGPLAN Notices*, 27(12):39–46, December 1992.

[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.

[Dew84] A K Dewdney. Core war. *Scientific American*, May 1984.

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

[Dil88] Antoni Diller. *Compiling Functional Languages*. John Wiley and sons, 1988.

[Fel87] Matthias Felleisen. *The Calculi of lambda-ν-CS conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[FRW90] William M. Farmer, John D. Ramsdell, and Ronald J. Watro. A Correctness Proof for Combinator Reduction with Cycles. *ACM Transaction on Programming Languages and Systems*, 12(1):123–134, January 1990.

[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.

[MP80] Steven S. Muchnick and Uwe F. Pleban. A semantic comparison of lisp and scheme. In *Conference Record of the 1980 Lisp Conference*, pages 56–65. The Lisp Conference, P.O. Box 487, Redwood Estates CA., 1980.

[QP90] Christian Queinnec and Julian Padget. A deterministic Model for Modules and Macros. Bath Computing Group Technical Report 90-36, University of Bath, Bath (UK), 1990.

[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).

[Que90] Christian Queinnec. Struggle, The First Denotational Game. In *EuroPal '90 – European Conference on Lisp and its Practical Applications*, pages 351–361, Cambridge (UK), March 1990.

[Roz92] Guillermo Juan Rozas. Taming the y operator. In *LFP '92 – ACM Symposium on Lisp and Functional Programming*, pages 226–234, San Francisco (California USA), June 1992. ACM Press. Lisp Pointers V(1).

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

[Sto77] Joseph E Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge Massachussetts USA, 1977.