New Insights into Partial Evaluation: the SCHISM Experiment

Charles CONSEL

LITP – Université Paris 6
(Couloir 45–55, 2ème étage)
4 place Jussieu, 75252 Paris CEDEX 05, FRANCE
uucp : . . . !mcvax!inria!litp!chac

Université Paris 8
2 rue de la Liberté, 93526 Saint Denis, FRANCE

Abstract

This article describes SCHISM: a self-applicable partial evaluator for a first order subset of Scheme. SCHISM takes place in the framework of mixed computation, and is situated along the line of the MIX project at the University of Copenhagen. The goal is automatically to generate compilers from interpreters by self-application and we have done this with an extensible and directly executable first order subset of Scheme.

SCHISM is an open-ended partial evaluator with a syntactic extension mechanism (macro-functions written in full Scheme). Furthermore, the set of primitives is extensible without any modification of the system.

Partial evaluation of functional languages relies on the treatment of function calls. We have chosen to use annotations for driving SCHISM to eliminate a call (unfold it) or to keep it residual (specialize it). They are local to each function rather than to each function call. This solves the problem of multiple calls to the same function with different patterns of static and dynamic arguments. Usually two pitfalls are possible in such a case: either to make all of these calls residual and specialize the function exponentially; or to eliminate the calls systematically and possibly start an infinite unfolding. Both are avoided by the use of a filter expression attached to functions. These filters drive SCHISM.

In this article we first describe our first order Scheme both with its abstract syntax and informally. Then we analyze the possibilities raised by keeping annotations local to each function. Finally we propose a partial solution to the open problem of reusing the store: the idea is to distinguish compile time and run time in the interpreter itself. In the end some conclusions and issues are proposed.

Keywords

Program transformation, applicative languages, partial evaluation, Scheme, SCHISM, mixed computation, program generation, specialization, unfolding, compiler generation.

Introduction

Partial evaluation [Futamura 82] is a general technique of program transformation. It is based on Kleene's S-m-n theorem [Kleene 52] and in essence consists of the specialization of a program with respect to some known data. To some extent, this encourages to view these known data as static and the unknown data as dynamic. The point is that all the parts in the program which manipulate static data can be processed statically. What remains then is a residual program ready to operate on the dynamic data or to be specialized further.

The idea of specializing programs is first used in [Lombardi 67] to perform incremental compilation. Applied to the triplet

$$< Interpreter, Program, Data >$$

it expresses that specializing an interpreter with respect to a program leads to compiling this program. [Futamura 71] generalizes that application by specializing the partial evaluator itself. This leads to producing a compiler from a partial evaluator and an interpreter, and to producing a compiler generator by specializing the partial evaluator with respect to itself. These applications are now known as the *Futamura projections* and require the partial evaluator to be self-applicable, that is an autoprojector [Ershov 82].

Mix [Jones *et al.* 85] was the first actual self-applicable partial evaluator. It is able to generate stand-alone compilers as well as a compiler generator.

This article presents our self-applicable partial evaluator SCHISM[1]: it is homogeneously specified in Scheme [Rees & Clinger 86] [Consel *et al.* 86] and offers some new insights in the domain of partial evaluation.

SCHISM is built on top of Scheme and written in Schismer: a first order[2] subset of Scheme. Schismer, as the language of an autoprojector, is self-interpretable. It offers a syntactic extension mechanism [Kohlbecker 86] to use high level constructs rather than only a language which sometimes reveals to be a bit too low level. These syntactic extensions are built in full Scheme and they generate Schismer code. We have also built SCHISM to be extensible: one can enrich the initial set of primitives with user defined Scheme functions.

[1] We have called it SCHISM because it operates on data which have been separated into static and dynamic parts.

[2] We have made Schismer first order because it still is an open problem to treat higher order languages, although we hope to offer here a new insight towards that direction.

SCHISM processing consists of specializing a Schismer program: it folds and unfolds function calls, eliminates them or keeps them residual. Annotations are the mean to drive partial evaluation during these transformations by specifying what is static and what is dynamic, that is: what to unfold, what to keep residual, what to specialize. We have taken the choice of keeping annotations local to Schismer functions.

This article is organized as follows. The first section presents the abstract syntax of Schismer and its informal description. The second section describes the partial evaluator; we show an example of a Schismer program in concrete syntax and the residual program produced by SCHISM. Section 3 makes a comparison of our strategy for handling function calls with the Mix approach and illustrates it with some examples. Section 4 describes how residual programs at run time may use data structures other than those available in a partial evaluator.

1 The language Schismer

Schismer is a first order subset of Scheme. Its surface syntax is almost familiar: it is the one from Scheme, enriched with filters. Filters are situated before the body of named functions and lambda-expressions. A Schismer program is basically a set of recursive, statically scoped equations.

Schismer has been conceived to be well-suited for a self-applicable partial evaluator: as shown in the abstract syntax below, the language is simple. However, we wanted to provide a language rich enough to express both a non-trivial autoprojector and a wide variety of interpreters. The idea has been to offer syntactic extensions (macros): they make a program more expressive and concise. Presently, one can either use already existing syntactic extensions or write his own ones in Scheme and with the full power of Scheme. Furthermore, the initial set of primitives is extensible with Scheme user defined functions.

1.1 Abstract syntax

K ∈ Con	constants, including quotations
I ∈ Ide	variables
E ∈ Exp	expressions
F ∈ Fun	functional objects
L ∈ Lam	λ-expressions
D ∈ Def	named definitions
P ∈ Prg	Schismer program

P⟶ (program (I*) (D+) I)

D⟶ (define (I^+) (filter E_0 E_1) E_2)

L⟶ (lambda (I^*) (filter E_0 E_1) E_2)

F⟶ L | I

E ⟶ K | I | (F E^*)
| (if E_0 E_1 E_2)
| (external I E^*)

Figure 4 in section 2 displays a complete Schismer program, performing the catenation of two lists.

1.2 Informal description of Schismer

The constants are the integers, the boolean values `#!true` and `#!false`, the null object `()`, the quoted pairs and the quoted symbols.

A program is divided into three parts:

- A list of syntactic extensions files. They are loaded by the system and used to produce a pure Schismer source program. One can include the system files as well as his own files.
- A list of user defined functions. Named definitions cannot be embedded for the sake of simplicity.
- A variable that is the name of the main function of the program, *i.e.*, the function which starts the application.

A named definition has three parts. The first part is a list of variables, whose head is the function name and whose rest is the parameters list. The second part is the annotation (*filter*) which drives the partial evaluator for treating the function call. The last part is the body of the function.

A λ-expression also has three parts. The first part is the parameters list. The second part is the *filter*. The last part is the body of the λ-expression. It is the body which is partially evaluated.

The if construct is a ternary operator. The first part is evaluated. If it yields a true value then the second part is evaluated and its value is returned. Otherwise, the third part is evaluated and its value is returned.

```
(defschismer-macro (nth n l)
    (list 'car
          (let loop ((n n))
              (if (eq? n 0) l
                  `(cdr ,(loop (-1+ n)))))))
```

Figure 1: A syntactic extension written in Scheme: nth

The external construct allows one to include functions that produce side effects in his programs (this is analogous to the `x-functions` in Mix [Diku 87]).

1.3 Syntactic extensions

The syntactic extension facility provides a powerful language tool for building high level constructs by macro-generation of Schismer code. Using this mechanism, we have implemented a subset of the Common Lisp *structures* [Steele 84]. The syntactic extension defining a given structure generates a set of new syntactic extensions to create the object; to access each field; and to test whether an object is an instance of the given structure. This has proven useful.

Figure 1 displays the syntactic extension `nth` taking as arguments an integer (a constant) and an expression and generating the right combination of `car` and `cdr` to access the n^{th} element of a list.

The syntactic extension mechanism could be viewed as a redundant feature together with a partial evaluator which sometimes performs the same task. However, it yields to constructs that may generate a complex combination of Schismer forms, uneasy to write by hand, such as `cond` and `case`. Moreover, a simple preprocessing phase is more reasonable than using the partial evaluator for what is after all a trivial program manipulation.

1.4 The environment

The initial environment used by SCHISM is built with two sets of functional objects. The *low environment* is the first set; it consists of all the primitives used by the interpreter. The *high environment* is the second set; it consists of all the user defined functions (named definitions).

The low environment is extensible. It is interesting to put unary functions in the low environment rather than defining them. Specializing a unary function with an unknown argument generally behaves like the identity. Conversely, with known argument the primitive will be directly called rather than symbolically processed: its execution time is much faster.

2 SCHISM: the partial evaluator

SCHISM is written in Schismer to be self-applicable. As in Scheme, the integers, the boolean values and the null object do not need any quotation since SCHISM considers the program and the partial evaluation environment as distinct domains. This makes programs more readable. This section focuses on the key point of our system: the treatment of function calls.

2.1 Function calls

For each function call, SCHISM determines whether the operator is a primitive, an **external**, a λ-expression or a named definition. This section presents the way SCHISM treats each of them.

2.1.1 A primitive

Since primitives are written in Scheme, they can be compiled for efficiency (and they are of course). If all the arguments of the primitive are known, the Scheme function is directly called and executed. The return value is used by SCHISM to continue processing. If some of the arguments are unknown, SCHISM substitutes all the known expressions by their values and makes the function call residual.

2.1.2 A λ-expression

Since Schismer offers only one side-effecting construct (`external`), it may be interesting to make an almost systematic β-reduction. However, this approach may generate programs where the same expressions are recomputed several times. It is better to make a selective reduction that avoids recomputation, as described in [Steele 78].

To make a selective β-reduction, a λ-expression includes a filter which drives the SCHISM treatment. Section 2.2 shows that filters for λ-expressions can be generated automatically.

The filter of a λ-expression consists of two expressions. When SCHISM encounters a redex, it evaluates the first expression of the filter (which is a Schismer expression) with the known or unknown value[3] of the arguments. This expression returns the truth value `#!true` if the requirements are fulfilled to β-reduce the λ-expression, *i.e.*, to substitute the parameters by the arguments and eliminate the λ-expression. If the requirements are not fulfilled the first expression returns the truth value `#!false` and SCHISM activates the second expression. As the first one, this expression receives the arguments of the application and returns a list of boolean values by mapping the list of arguments. For each `#!true` value the corresponding parameter is eliminated and the argument is substituted. For each `#!false` value, the parameter and its corresponding argument are kept residual. This treatment gives to SCHISM a particular piece of information for each parameter of the λ-expression.

```
(define (fun a)
   ((lambda (x y)
        (filter #!false
                (list (known? x) (known? y)))
        (cons (f1 x) (f2 y)))
    a 11))
```

Figure 2: An simple example of λ-expression involving a filter

```
(define (fun a)
   ((lambda (x) (cons (f1 x) (f2 11))) a))
```

Figure 3: The effect of the filter to partially evaluate a λ-expression

Figure 2 illustrates with a simple example the use of a λ-expression (the filter of the function has been intentionally omitted). The first expression of this filter is `#!false`. It indicates to SCHISM that this λ-expression should never be β-reduced. The second expression builds a list where the first element is `#!true` if the parameter `x` has a known value. Otherwise, it is `#!false`. The value of the second element of the list (parameter `y`) is determined similarly. A value `#!true` in this list indicates to SCHISM that the corresponding parameter should be eliminated and the argument substituted in the λ-expression body. A value `#!false` keeps the corresponding parameter and argument residual. Figure 3 is the residual program generated by SCHISM if a is dynamic.

[3] A value is known when it is a list whose first element is `quote`. Otherwise the expression is unknown.

```
(program
    (user-syntactic-extensions.h)
    (
        (define (fun l)
            (filter #!false (list l))
            (append '(1 2 3) l))

        (define (append l1 l2)
            (filter (known? l1) '(l1 l2))
            (if (null? l1) l2
                (cons (car l1)
                      (append (cdr l1) l2))))
    )
    fun)
```

Figure 4: A program with filters in named definitions

```
(define (fun-0 l)
    (cons (quote 1)
          (cons (quote 2)
                (cons (quote 3) l))))
```

Figure 5: A residual program where SCHISM has unfolded the function append

2.1.3 A named definition

One generally names a function to call it recursively. For this reason, SCHISM carefully treats named function calls in order to avoid infinite unfolding or infinite specialization. Furthermore, as for λ-expressions, unfolding may generate inefficient code where expressions are recomputed several times.

Figure 4 shows two named functions: they contain a filter consisting of two expressions. As for λ-expressions, for each named function call, the first expression of the filter is evaluated with the values (known or unknown) of the arguments in the application. According to the values received, the first expression returns **#!true** if it wants SCHISM to unfold the call. If the first expression returns **#!false**, the second expression will be activated with the arguments. This second expression specifies how the function has to be specialized, *i.e.*, what are the parameters to be eliminated. A list of values containing a decision for each parameter (as above) is then returned. If a value is known, this constant will replace the corresponding parameter in the function body and the parameter will disappear.

Figure 5 displays the residual program of figure 4. The function **fun** has been renamed **fun-0** to distinguish it from the original version. According to the filter, unfolding has been performed to treat the call to the function **append**, as the induction variable is known.

2.2 Automatic generation of annotations

Experience in writing Schismer programs has shown that a number of program schematas have "obvious" annotations (traversing a list tail-recursively, *etc.*). As a first attempt to automatically generate annotations, we have defined some of them as syntactic extensions. For example we supply a syntactic extension **let** that macro-expands to the corresponding application of a λ-expression: a standard annotation is provided if none is specified. This generic filter produces code that allows SCHISM to eliminate a parameter if it is known or if it is unknown but bound to another variable. Similarly for the named functions, if the filter is not included in the definition, a systematic unfolding filter is inserted. Other cases may be treated. For instance we are currently developing the automatic generation of annotations for self recursive functions by providing some simple syntactic extensions implementing loop structures.

2.3 Reductions

Partial evaluation is based on constant propagation and reduction of expressions. This propagation may be stopped when one or several known data are combined with one or several unknown data. A simple example is **(car (cons 1 (f a)))**: this expression cannot be reduced if the partial evaluator does not know the semantics of **car**. To solve this problem we have enhanced SCHISM with some rules:

(car (**cons** E_0 E_1)) → E_0
(cdr (**cons** E_0 E_1)) → E_1
(null? (**cons** E_0 E_1)) → **#!false**

Similarly, the conditional construct **if** is reduced by SCHISM according to the following rules:

(**if** E_0 E_1 E_1) → E_1
(**if** E_0 **#!true #!false**) → E_0
(**if** (**equal?** E_0 **#!false**) E_1 E_2)
    → (**if** E_0 E_2 E_1)
(**if** (**equal? #!false** E_0) E_1 E_2)
    → (**if** E_0 E_2 E_1)

These rules may appear trivial, and they certainly are. The point here is that a partial evaluator acts as a program specializer and uses some very general program transformation techniques. These simplification rules are not surprising in themselves. What is interesting is to know that they are present in a partial evaluator and intervene here in SCHISM.

Figure 6 displays a source program where an association list is used to represent an environment. This program could be the beginning of an interpreter. The function **make-env** builds the association list with a list of variables and a list of values. The function **lookup** calls the function **assoc** with the association list to find the value of the variable **var**.

Figure 7 shows the residual program when SCHISM knows that **var** is bound to **'c** and that **var*** is bound to **'(a b c d e)**. We can see that the access to the value of the variable c has been totally determined. Program specialization subsumes program simplification.

3 Why keeping the annotations local to the function?

This section compares our approach together with the approach taken in Mix [Jones *et al.* 87]. The goal is to decide for a function call whether it has to be unfolded or suspended.

Mix makes the decision about this for each function call encountered in a program. This implies that the annotation of a function call is made static. Figure 8 points out when this approach could be too conservative. It shows a classical function called twice with two different patterns of static and dynamic arguments. The Mix annotations [Jones *et al.* 85] [Sestoft 86] for the function calls are used: a function call marked `call` will always be unfolded (eliminated); a function call marked `callr` will be residual (specialized).

In figure 8, the function **append** has an induction variable `l1` [Aho *et al.* 86]. If `l1` is known, unfolding can be performed safely. If this variable is unknown, unfolding cannot take place and the only possible operation is specializing this function with respect to `l2`. A problem occurs if the function **append** is called once with the known induction variable, and a second time with the unknown induction variable. Since the recursive call in **append** is annotated to be residual both cases cannot be treated in an optimal way and the result is far too conservative.

```
(program
    (user-syntactic-extensions.h)
    (
        (define (lookup var var* val*)
            (filter #!false
                    (list var var* val*))
            (cdr (assoc var
                        (make-env var* val*))))

        (define (make-env var* val*)
            (filter (known? var*) 'void)
            (if (null? var*)
                '()
                (cons
                    (cons (car var*) (car val*))
                    (make-env (cdr var*)
                              (cdr val*)))))

        (define (assoc key alist)
            (filter (known? alist) 'void)
            (cond
                ((null? alist)
                    #!false)
                ((equal? (car (car alist))
                        key)
                    (car alist))
                (else
                    (assoc key (cdr alist)))))
    )
    lookup)
```

Figure 6: A program representing an environment with an association list

```
(define (lookup-0 val*)
    (car (cdr (cdr val*))))
```

Figure 7: Effects of reduction rules

One may annotate the function **append** to make a systematic specialization: this is safe, but the residual program is huge, as each recursive call to **append** produces a residual function.

On the other hand, a strategy based on a systematic unfolding produces infinite loops at partial evaluation time. If in figure 8 the recursive call to append is annotated to be unfolded when the induction variable is unknown, the function will be unfolded infinitely.

In figure 9 (the Schismer version), the function **fun** is (locally) annotated to be always unfolded[4].

[4] Since **fun** is never to be specialized, the second part of the

```
(define (fun l)
    (cons (call append l '(1 2 3))
          (call append '(a b c) l)))

(define (append l1 l2)
    (if (null? l1)
        l2
        (cons (car l1)
              (callr append (cdr l1) l2))))
```

Figure 8: A too conservative annotation using MIX notations

```
(define (fun l)
    (filter #!true 'void)
    (cons (append l '(x y z))
          (append '(a b c) l)))

(define (append l1 l2)
    (filter (known? l1) (list 'l1 l2))
    (if (null? l1)
        l2
        (cons (car l1)
              (append (cdr l1) l2))))
```

Figure 9: The equivalent program in Schismer

The filter of **append** makes a call unfolded when its first argument is known. If not, the second part of the filter drives the specialization of the call with respect to l2.

Our strategy allows the annotations to drive SCHISM according to quantitative criteria, which is strictly more powerful than a boolean annotation. Figure 10 presents the same function **append** as figure 4 but with a new filter. It indicates that a call to **append** should be unfolded first if the parameter l1 is known *and* second when the length of the list is not greater than 20. Otherwise, a call to **append** is kept residual and *only* specialized with respect to l2 (if known). This last example shows that keeping annotations local to each function makes it possible to tune SCHISM precisely.

filter will not be activated. We note it as void for readability because this second part is to be ignored.

```
(define (append l1 l2)
    (filter
        (and (known? l1)
             (<= (length l1) 20))
        (list 'l1 l2))
    (if (null? l1)
        l2
        (cons (car l1)
              (append (cdr l1) l2))))
```

Figure 10: The equivalent program in Schismer

4 Extra data structures in residual programs

To be self-applicable a partial evaluator must be expressed with the same objects that it treats. Presently they are lists: one represents objects such as the environment in an interpreter with lists. In particular, an assignment in the interpreted language is commonly implemented by rebuilding the environment. The reason is that the interpreter is written without assignment. This is a problem because the naive specialization of an interpreter with respect to a target program with assignments leads to a program that rebuilds entire pieces of the interpretation environment. Then it may happen that the specialized program is not as efficient as could be expected.

We propose an approach for designing interpreters that makes it possible to generate residual programs where only the allocations of the program remain and *not* the allocations required by the interpreter.

This approach is based on splitting the bindings of identifiers to values [Jones *et al.* 87]. We use the same strategy as in denotational semantics, where the values of some variables are not given until run time. This creates frozen expressions [Gordon 79] [Schmidt 86]. The primitives that manipulate the store are changed according to the data type used to implement the store. Then compilation phase and run time phase are totally separated. As an example (see Appendix A) we have adapted the MP interpreter described in [Sestoft 85]. Unlike the residual program produced by Mix with respect to the `reverse` program, SCHISM has generated a residual program (see Appendix B) where the primitive **cons** is only used where it is needed in the program and not because it is needed in the interpreter (see figure 11). This is a first contribution to the open problem of reusing the store.

```
;;; variable l = offset 1
;;; variable res = offset 0

(define (execute-mp-0 input store)
    (store-ref
        (mp-while-1
            (store-set! store (quote 1) input))
        (quote 0)))

(define (mp-while-1 store)
    (if (null? (store-ref store (quote 1)))
        store
        (mp-while-1
            (mp-block-2
                (store-set!
                    store
                    (quote 0)
                    (cons
                        (car (store-ref
                                store
                                (quote 1)))
                        (store-ref
                            store
                            (quote 0))))))))

(define (mp-block-2 store)
    (store-set!
        store
        (quote 1)
        (cdr (store-ref store (quote 1)))))
```

Figure 11: A residual program reusing the store

5 Conclusions and Issues

We have built a partial evaluator operating homogeneously on a first order subset of Scheme. We believe that it offers some new insights into partial evaluation engineering: the whole system is open-ended; annotations can partly be generated automatically; the set of primitives is extensible; local annotations allow to drive SCHISM with a high precision.

After this article has been written, we have achieved complete self-application. SCHISM generates small sized and readable compilers, and is currently experimented both at LITP and at DIKU.

Next stage in our work is to process a fully imperative language with SCHISM. We are now elaborating a new methodology that describes an imperative language together with its interpreter. The idea is to make the interpreter ready to be specialized. The variety of concepts is already raising problems and this experience is already enriching SCHISM.

Acknowledgements

Thanks to Anders Bondorf, Neil Jones, Torben Mogensen and Peter Sestoft for their welcome at DIKU and their close interaction during the workshop on Partial Evaluation and Mixed Computation. Special thanks to Olivier Danvy for many suggestions and discussions about my work and this paper.

Bibliography

Aho, A. V., Sethi, R. and Ullman J. D.
Compilers: Principles, Techniques and Tools,
Addison-Wesley [1986]

Bondorf A.
Towards a Self-Applicable Partial Evaluator for Term Rewriting Systems,
North Holland Publ. proceedings of the Workshop on Partial Evaluation and Mixed Computation, Denmark [1987]

Consel C., Deutsch A., Dumeur R. and Fekete J-D.
Skim Reference Manual,
Rapport Technique 86/09 Université de Paris 8, France [1986]

Diku, University of Copenhagen
The Mix System User's Guide Version 3.0
Diku internal report, University of Copenhagen, Denmark [1987]

Ershov, A. P.
Mixed Computation: Potential Applications and Problems for Study,
Theoretical Computer Science 18 (41-67) [1982]

Emanuelson, P. and Haraldsson A.
On Compiling Embedded Languages in Lisp,
Lisp Conference, Standford, California, (208-215) [1980]

Futamura, Y.
Partial Evaluation of Computation Process - an Approach to a Compiler-Compiler,
Systems, Computers, Controls 2, 5 (45-50) [1971]

Futamura, Y.
Partial Computation of Programs,
In E. Goto et al (eds.): RIMS Symposia on Software Science and Engineering, Kyoto, Japan.
Lecture Notes in Computer Science 147, 1983, (1-35) [1982]

Gordon, M. J. C.
The Denotational Description of Programming Languages,
Springer-Verlag [1979]

Jones, N. D., P. Sestoft, and H. Søndergaard
An Experiment in Partial Evaluation: the Generation of a Compiler Generator,
Rewriting Techniques and Applications, Dijon, France.
Lecture Notes in Computer Science 202, (124-140) Springer-Verlag [1985]

Jones, N. D., P. Sestoft, and H. Søndergaard
Mix: a Self-Applicable Partial Evaluator for Experiments in Compiler Generation,
Diku Report 87/08, University of Copenhagen, Denmark [1987]

Kleene, S. C.
Introduction to Metamathematics,
Van Nostrand [1952]

Kohlbecker, E. E.
Syntactic Extensions in the Programming Language Lisp,
PH. D. thesis, Technical Report No 199, Indiana University, Bloomington, Indiana [1986]

Lombardi, L. A.
Incremental Computation,
Advances in Computers 8 (ed. F. L. Alt and Rubinoff), Academic Press, (247-333) [1967]

Rees, J. and W. Clinger (eds.)
Revised[3] Report on the Algorithmic Language Scheme,
SIGPLAN Notices 21, 12, (37-79) [1986]

Schmidt, D. A.
Denotational Semantics: a Methodology for Language Development,
Allyn and Bacon, Inc. [1986]

Sestoft, P.
The Structure of a Self-Applicable Partial Evaluator,
Diku report 85/11, University of Copenhagen, Denmark. [1985].

Steele G. L. Jr.
Rabbit: a Compiler for Scheme,
MIT AIL TR 474, Cambridge, Mass. [1978]

Steele G. L. Jr.
Common Lisp,
Digital Press [1984]

Appendix A: The MP Interpreter in Schismer

```
;;; This MP-int is almost the same as the Mix version
;;; Activation: (program parameter locals block)

(program
    (mp.h)
    (
        (define (execute-mp program input store)
            (filter #!false (list program input))
            (let ((var-env (make-var-env (nth 2 program) (nth 1 program))))
                (let ((newstore (update-env (car (nth 1 program)) input var-env store)))
                    (filter #!false (list (known? newstore)))
                    (mp-block (nth 3 program) var-env newstore))))

        (define (make-var-env local-name* par-name*)
            (filter #!true 'void)
            (if (null? local-name*)
                par-name*
                (cons (car local-name*)
                      (make-var-env (cdr local-name*) par-name*))))

        (define (run-mp expr var-env store)
            (filter #!true 'void)
            (cond
                ((and (pair? expr)
                      (or (equal? (car expr) ':=)
                          (equal? (car expr) 'while)))
                    (run-command expr var-env store))
                (else
                    (run-expression expr var-env store))))

        (define (run-command expr var-env store)
            (filter #!true 'void)
            (case (car expr)
                ((:=)
                    (update-env (nth 1 expr)
                                (run-mp (nth 2 expr) var-env store)
                                var-env
                                store))
                (else
                    (mp-while (nth 1 expr) (nth 2 expr) var-env store))))

        (define (run-expression expr var-env store)
            (filter #!true 'void)
            (cond
                ((not (pair? expr))
                    (fetch expr var-env store))
                (else
                    (case (car expr)
                        ((cons)
                            (cons (run-mp (nth 1 expr) var-env store)
                                  (run-mp (nth 2 expr) var-env store)))
                        ((car)
                            (car (run-mp (nth 1 expr) var-env store)))
                        ((cdr)
                            (cdr (run-mp (nth 1 expr) var-env store))
```

```
                    ((equal?)
                        (equal? (run-mp (nth 1 expr) var-env store)
                                (run-mp (nth 2 expr) var-env store)))
                    ((quote)
                        (nth 1 expr))
                    ((if)
                        (if (not (null? (run-mp (nth 1 expr) var-env store)))
                            (run-mp (nth 2 expr) var-env store)
                            (run-mp (nth 3 expr) var-env store)))
                    (else
                        '|unknown form|)))))

        (define (mp-block expr* var-env store)
            (filter #!true 'void)
            (if (null? (cdr expr*))
                (run-mp (car expr*) var-env store)
                (mp-block (cdr expr*) var-env (run-mp (car expr*) var-env store))))

        (define (mp-while condition body var-env store)
            (filter #!false (list condition body var-env 'store))
            (if (not (null? (run-mp condition var-env store)))
                (mp-while condition body var-env (mp-block body var-env store))
                store))

        (define (fetch var var-env store)
            (filter #!true 'void)
            (store-ref store (give-offset var var-env)))

        (define (update-env var val var-env store)
            (filter #!true 'void)
            (external store-set! store (give-offset var var-env) val))

        (define (give-offset var var-env)
            (filter #!true 'void)
            (cond
                ((null? var-env)
                    '|undefined variable|)
                ((equal? var (car var-env))
                    0)
                (else
                    (+ 1 (give-offset var (cdr var-env))))))
    )
    execute-mp)
```

Appendix B: reverse written in MP

```
(program
    (l)
    (res)
    (
        (while l (
            (:= res (cons (car l) res))
            (:= l (cdr l)) )  )
        res
    ) )
```