Vrije Universiteit Brussel
Faculteit Wetenschappen

Continuation-Passing-Style as an Intermediate Representation for Compiling Scheme

Kris De Volder (kdvolder@vnet3.vub.ac.be)

Techreport vub-prog-tr-94-09

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Continuation-Passing-Style as an Intermediate Representation for Compiling Scheme

Kris De Volder
Programming Technology Lab
Computer Science Department
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
email: kdvolder@vnet3.vub.ac.be

Abstract

This paper describes the implementation of a small experimental Scheme compiler, using Continuation Passing Style (CPS) as an intermediate representation for the source code. CPS is a form of code that makes control flow and control information explicit. This, in combination with a solid theoretical basis (lambda-calculus) makes it an excellent medium for representing intermediate code.

0 Introduction:

The main goal of the project was to study the techniques used in transforming a high-level Scheme program into a low-level program, expressed in a typical, assembler like, register oriented language. We did not aim to produce a viable compiler. Our main interest was to explore the possibilities and usefulness of the CPS approach. Using CPS in compilation is a novel approach to compiler writing, first suggested by Steele [16,17]. It has been successfully applied to write compilers for Scheme in other projects [9,16,18]. We have depended very strongly upon this work, especially upon the implementation of Orbit [9].

The VM is implemented in Scheme. By depending on Scheme's garbage collector, and representation for lists, vectors, strings, etc., we only put the least possible effort into nitty-gritty details.

This paper describes our system, starting with an overview of the compiler's layered structure, and then proceeding to describe every individual layer and it's purpose in a separate section. The code generation phase is technically complicated, therefore only the most important aspects will be highlighted. We will show that the resulting compiled code is reasonably efficient. However there still is a lot of room for improvement. Finally we will summarise the advantages and disadvantages of the CPS approach, and discuss related work.

1 Overview of the compiler

Figure 1 shows a schematic view of the compiler's layered architecture. Source code passes through successive layers of transformations. Every layer in the process produces equivalent Scheme code which is in some way or another either simpler, more efficient, or closer to compiled code than the code from the layer above.

The compiler is divided into a front-end and a back-end. The front-end contains a number of pre-processing phases that produce code in CPS-form. The result is then passed on to the back-end which contains the optimisation and code generation phases. Optimisations are performed directly on CPS-code, transforming it into optimised CPS-code which is subsequently used by the code generator to produce executable code. Note that the code generation phase is the only phase that does not produce Scheme code (it produces code in the target language), all other phases are "Scheme to Scheme" transformations.

The idea behind this structure is that the front end layers rewrite expressions of as many types as possible into combinations of more primitive types. Because only a few types of expressions are left in the resulting code only a few simple optimisations should suffice to obtain a high degree of optimisation.

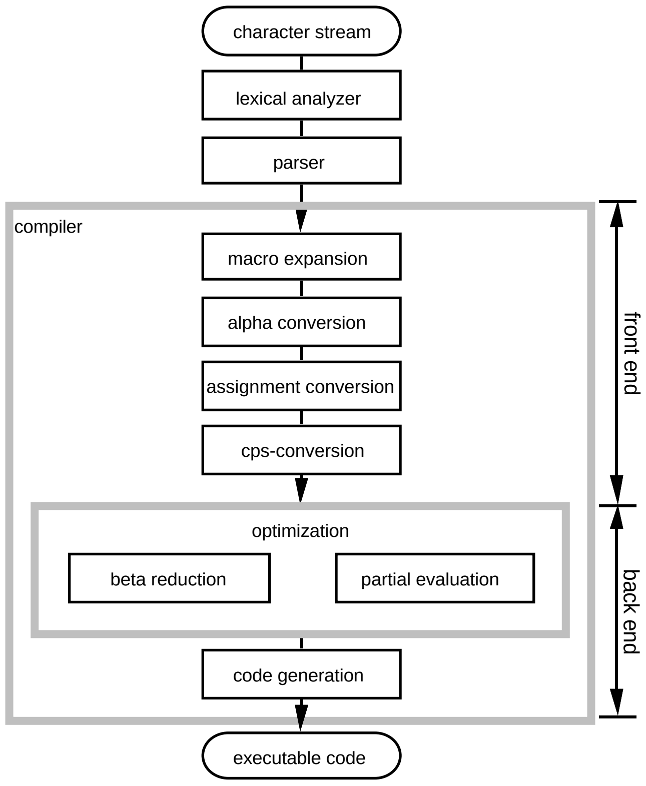

Figure 1: "Schematic view of the Compiler"

2 Macro Expansion

The macro expansion phase transforms derived expression types into primitive expression types. All derived expressions are lists starting with a particular symbol identifying the type of the expression. The macro expander consists of a table of transformation procedures, that associates a particular transformer to a symbol identifying a derived expression type. The expander recursively traverses the expression, invoking the appropriate transformers whenever it encounters a derived expression type.

We will not go into more detail because the expansion of macros[1] is a traditional and well known component of a Scheme implementation and there are not many difficulties in implementing it.

3 Alpha Conversion

This is a simple transformation, that is used frequently in compilers or code transformers that have to deal with code that possibly contains aliased variables. Alpha conversion renames all local variables to unique names, thus avoiding problems with aliassing in subsequent phases of the compiler.

We won't go into further detail because this is a rather trivial code transformation.

[1] This might be a little bit of a misrepresentation. The R3RS report [11] which was used as a specification for our implementation, does not specify any macro facility as a part of the Scheme language. Most R3RS implementations however do provide a macro-system similar to the macro-system of Common Lisp. These kinds of macro-systems are well known things, and are not hard to implement. Recently however there has been a lot of fuss: the opinion is growing that the traditional Common-Lisp notion of macros is inadequate, and attempts are being made to improve upon this, defining a better macros-system for Scheme. The R4RS report [12] specifies an optional macro extension to Scheme, and there are numerous other macro-proposals for Scheme [4,5,6,8]. These systems are much harder to implement, and their implementation and definition is still under study. These topics fall outside of the scope of our work, that's why we chose not to implement a macro-system for user-defined macros and only use a simple, more traditional macro transformation engine to implement derived expression types as pre-defined macros.

4 Assignment Conversion

4.1 What is assignment conversion?

This phase eliminates assignments to local variables. Assignment conversion introduces an extra indirection for side effected variables, replacing them by "cell" data-structures that hold the contents of the variable. References and assignments to these variables are converted into calls to primitive operations that get or set the contents of the cell.

4.2 Dealing with global variables

We also deal with global variables in this phase. Normally this isn't part of assignment conversion as described in [9]. However, we chose to also eliminate side effects to global variables by converting references, definitions and assignments to global variables into calls to the primitives *get-top*, *def-top* and *set-top!* that access the global variables array. This results in an efficient implementation of access and assignment to global variables because the variable names can be replaced by indexes in the table.

Not all global variables are handled this way: a distinction is made between integrable and non-integrable variables. The integrable variables are guaranteed to be side effect-free, and they will not be transformed. In our implementation all global variables containing primitives (like cons, +, <, car, etc.) are integrable and they cannot be assigned[2] to.

4.3 Advantages and disadvantages

Scheme is based on the theory of lambda-calculus, and many optimisations stem directly from the theory, but side effects are not part of lambda calculus. Therefor it is often necessary to insure that no side effect is involved before performing an optimisation. This has the disadvantage of complicating the optimisation phase, and prohibiting a great number of optimisations in cases where it is not certain that a side effect could be involved. Assignment conversion moves side-effects out of the environment, by introducing an extra indirection. This guarantees that environments are immutable, which greatly simplifies the optimisation phases, and additionally has advantages for code generation. It makes it possible to generate code that keeps multiple copies of variables in different places to enhance performance.

The disadvantage of assignment conversion is the extra indirection that is introduced. This makes references and assignments to local variables less efficient. This inefficiency is often argued to be of minor importance because it is bad programming style in Scheme to do side effects to local variables. Often this is true, but one has to be aware of it that in an object-oriented programming style, side effects to local variables are commonly used to model objects with state.

5 CPS and CPS-conversion

5.1 A few introductory words about CPS

CPS stands for "Continuation Passing Style". CPS-code is a form of Scheme code that has the property that *no function call ever returns*. Control flow like procedure call and return is explicit. Every function call passes a continuation as an extra argument. The continuation represents the program point after the procedure call, the point where the computation is to continue after the called procedure has completed. This continuation is represented by a procedure of one argument, receiving the value that would normally have been returned as a result from the procedure call. Thus in CPS code every transfer of control is represented by a procedure call. This includes return from a procedure, which will be represented by a call to a continuation procedure. Instead of returning a result, procedures call a continuation procedure, thus no procedure call ever returns. This is why procedure call in CPS-code actually behaves more like some kind of *goto* statement. A call in CPS only transfers control in one direction, from caller to callee.

[2] This does not conform to R3RS [11], which does not provide the distinction between integrable and non-integrable variables. All variables should be the same and can all be assigned to. We took this option in our implementation to make things simpler, in a real implementation, an analyses of the program must determine whether or not a global variable is involved in a side effect and can be categorised as integrable.

5.2 The CPS-transformation

The actual transformations we have used, are based upon, and very similar to those used in Orbit [9]. Therefore we won't give a specification, instead we will use an example to give an idea of the spirit of CPS-code to the reader not already familiar with it. The main idea is to convert all calls and lambda expressions, adding an extra continuation argument to the front of the argument list.

```
(define (fac n) (if (= n 0) 1 (* n (fac (- n 1)))))
```

If we put this through the front end of our system this will result in the following CPS-code. All administrative beta-redexes have been removed from this code. Normally this isn't done until later, in the optimisation phase. For clarity we have removed them manually.

```
(def-top *top-cont*
      'fac
      (lambda (cont137 n131)
        (= (lambda (cps-if140)
             (cps-test
               cps-if140
               (lambda () (cont137 '1))
               (lambda ()
                 (get-top (lambda (cps-proc144)
                            (- (lambda (cps-arg145)
                                 (cps-proc144
                                   (lambda (cps-arg143)
                                     (* cont137 n131 cps-arg143))
                                   cps-arg145))
                               n131
                               '1))
                          'fac))))
           n131
           '0)))
```

Notice how every call in this piece of code has an extra first argument, which is usually a lambda of one argument, a continuation. There is one call that is different: `(cont137 '1)` this is a call to a continuation procedure, representing the continuation of the call to factorial. This particular call to *cont137* represents the return of the value 1 to the call-site of the factorial procedure. Another call that differs a bit is the call to the CPS-test primitive, which represents the *if* from the original source code. An *if* is translated into a call to the CPS-test primitive, which takes 2 continuations as arguments, for the *true*-branch and for the *false*-branch respectively.

We can also observe the effect of most of the other front-end phases. Alpha-conversion has renamed all local variables to unique names, numbers are appended to all local variable names to make them unique. The effect of assignment conversion on the global variable *fac* can also be seen here. The definition of *fac* has been converted into a call to the *def-top* primitive, and the reference to it goes via the *get-top* primitive. For clarity the argument to *def-top* and *get-top* is simply the symbol *fac*, normally this symbol would be replaced by an index into the global-variables array. The assignment conversion of side-effected local variables cannot be seen here because the code contains no assignments.

CPS-code only consists of a very limited number of different syntactic constructs: function call, quoted constants and variables. Variables are either local variables, or global variables corresponding to some primitive operation (like +, =, CPS-test, …), all other "non primitive" global variables (in this case *fac*) have been removed by assignment conversion.

6 Optimisations

The optimisation phases are currently very primitive and consist of two co-operating mechanisms. Beta-reduction and a limited form of partial evaluation.

6.1 Beta-reduction

This is a direct derivative of beta-reduction in the lambda-calculus. It serves to eliminate administrative redexes from CPS-conversion and to optimise LET and similar constructs that are represented as lambda calls. In our implementation this phase is absolutely necessary because the CPS phase generates a great number of administrative beta-redexes which if not reduced will cause unacceptably inefficient code to be generated. This is a bit counter-productive: CPS-transformation generates a lot of redundant beta redexes and beta-reduction removes them. The whole process could be made significantly more efficient when CPS wouldn't generate all those administrative redexes in the first place. This can be accomplished with a more complex CPS-transformation algorithm, as described in [7]. Though important in a real system, performance was considered to be of minor importance in our experimental implementation and we choose to keep the transformations simple.

Beta reduction works on so called beta-redexes, that is lambda's that are called directly. It replaces formals by actuals in the body of a beta-redex and eliminates lambda calls without arguments. This is summed up in the following 2 transformation rules:

Rule1: Replacing formals by actuals

((lambda (*arg*$_1$ … *arg*$_{i-1}$ *arg*$_i$ *arg*$_{i+1}$ … *arg*$_n$) *body*) *expr*$_1$ … *expr*$_n$)

=>

((lambda (*arg*$_1$ … *arg*$_{i-1}$ *arg*$_{i+1}$ … *arg*$_n$) *subst-body*) *expr*$_1$ … *expr*$_{i-1}$ *expr*$_{i+1}$ … *expr*$_n$)

Subst-body is *body*, where every occurrence of *arg*$_i$ is substituted by *expr*$_i$. To avoid infinite expansion and code duplication this transformation rule is only applicable if either *expr*$_i$ is a constant or variable, or if *arg*$_i$ occurs at most once in *body*.

Rule 2: Removing lambda applications without parameters

((lambda () *body*)) => *body*

6.2 Partial Evaluation of Primitives

A limited form of partial evaluation replaces calls to primitives by their result whenever possible. For example: `(+ <continuation> 3 4)`, would be replaced by `(<continuation> 7)`.

Partial evaluation co-operates intimately with beta-reduction, because sometimes partial evaluation of a primitive makes more beta-reductions possible and vice-versa.

The partial evaluation phase is driven by a table that associates a partial evaluation method with every primitive. Many sophisticated optimisations strategies could be implemented this way. Boolean short-circuiting for example could be accomplished through implementing a sufficiently smart method for the *CPS-test* primitive [9]. Currently our system only handles constant folding for arithmetic operations, but this can easily be extended because of the table driven implementation.

7 Code Generation

This is the final and technically the most complicated phase, transforming optimised CPS code into code for the virtual machine. The crucial point is to generate as efficient code as possible for lambda expressions and calls, since these are the most frequently used constructs (representing every transfer of control). Normally evaluation of a lambda would involve creation of a heap closure. This would be hopelessly inefficient. The solution is classification of lambda's.

Currently we have 2 categories of lambda's: Class 1 are those that are called directly in the same environment as where they are defined (lambda's from LETs and continuations to primitive procedures). Class 2 are all the others. Class 1 lambda's are the majority in CPS code and calls to them are compiled even without a jump instruction and without generating a closure for them. Class 2 are compiled with heap closures. More efficient code could ultimately be generated by further diversifying the classes, but our simple classification already yields reasonable results. Reconsider for example the factorial function. The CPS-code for factorial from section 5.2 is repeated here, but this time all lambda's have been numbered. Class 1 lambda's are underlined and class 2 lambda's are in bold.

```
(def-top *top-cont*
  'fac
  (lambda248 (cont231 n224)
    (=
     (lambda249 (cps-if234)
       (cps-test cps-if234
                 (lambda254 () (cont231 '1))
                 (lambda250 ()
                   (get-top
                    (lambda251 (cps-proc238)
                      (-
                       (lambda252 (cps-arg239)
                         (cps-proc238
                           (lambda253 (cps-arg237)
                            (* cont231 n224 cps-arg237))
                          cps-arg239))
                       n224
                       '1))
                    'fac))))
     n224
     '0)))
```

The majority of the lambda's are class 1. Only two of them are class 2: the lambda representing the factorial procedure itself, and the lambda representing the continuation to the recursive call. The code generated from this CPS-code is shown in figure 2 below.

The rectangle indicates the code that corresponds to the factorial procedure, the surrounding code binds a closure corresponding to the code within the rectangle to the global variable *fac*. The only place within the rectangle where a heap-closure is being generated, is at line 14. This corresponds to *lambda253* in the CPS-code. This heap closure saves (in the heap) the registers that will be needed upon return from the recursive call. Then a jump to *endlambda253* jumps over the continuation's code (that is the code that will be executed upon return from the recursive call, the body of *lambda253*). The recursive call itself is at line 25. The 2 indicates that 2 arguments are passed along in register R0 and R1. The first argument in R0, is the continuation, and the second one is the real argument.

There are a few minor inefficiencies. Some jumps that could have been avoided by rearranging the code, the jump to *endlambda253* is an example. And some unnecessary register to register transfers, for example at line 23, the contents of R1 is "saved away" in R7 because an argument needs to be stored in R1, but R7 is not used anymore in the rest of the code. These inefficiencies are the result of an oversimplification of the code generator. They can be avoided with a more complex code generator. Since we were not trying to build a "real" compiler, we didn't make the extra effort.

A more serious inefficiency is that registers which are used across the recursive call are saved on the heap. A traditional compiler could probably have saved them on the stack[3], which is more efficient. This problem is more difficult to solve, we will elaborate on it in section 8.2.

---

[3] This is certainly true for compilers for (C, PASCAL, etc.) languages that do not have Scheme's powerful call-with-current-continuation mechanism. Call/cc makes things much more complicated because it is no longer guaranteed that continuations are always allocated and deallocated in a stack like manner, usually however they are allocated in a stack like manner because people do not tend to use call/cc very often.

```
     R0 := fac
     R1 := closure lambda248
     JUMP endlambda248
 ::lambda248
     ARGS 2
     R2 := 0
     R3 := =(R1 R2 )
     IF NOT R3 THEN JUMP else255
     R4 := R0
     R0 := 1
JUMP-PROC 1 R4
 ::else255
     R4 := get-top(R2 )
     R5 := -(R1 R2 )
     R6 := R0
     R0 := closure lambda253 R0 R1
     JUMP endlambda253
 ::lambda253
     ARGS 1
     R1 := Lev:0 Ofs:1
     R2 := R0
     R0 := *(R1 R0)
     R3 := Lev:0 Ofs:0
     JUMP-PROC 1 R3
 ::endlambda253
     R7 := R1
     R1 := R5
     JUMP-PROC 2 R4
 ::endlambda248
     R2 := R0
     R0 := def-top(R0 R1 )
     R3 := (%closure #f #f #f '*top-cont*)
     JUMP-PROC 1 R3
```

Figure 2: Code generated for "factorial"

8 Conclusion

8.1 Why use CPS as an intermediate representation

There are several reasons why CPS is an interesting representation for compiling programs.

It makes control flow and control information explicit. Pieces of control information, that is for example return addresses for procedure calls, are explicitly represented by continuation procedures. CPS-code has a very sequential nature, mostly because of the "goto-like" behaviour of procedure call. CPS has a low-level "almost compiled" look and feel to it, which makes it easier to transform into the target language than "normal" Scheme code.

The explicitness of control information has the advantage that in CPS-code, there is no distinction between user data and control information. Both are explicit in the code, both are stored in variables, and the variables are not distinguishable from one another. When performing optimisations on, or generating efficient code for variable lookup, these optimisations or code generation techniques equally apply to both user variables and control variables. Traditional compilers usually treat control information and user information differently, implementing separate mechanisms for both of them. For example, suppose we implement a strategy to analyse the CPS-code, deciding to keep some variables in registers while storing others on the stack and yet others in the heap. This would automatically apply to both user data (variables from the source program) and compiler data (for example return addresses) alike.

CPS combines both a low-level "close to compiled code" nature, with a solid theoretical foundation, the lambda calculus. This is especially advantageous for languages like Scheme, which are closely related to the lambda calculus. The theoretical nature of the code makes it a good medium for optimisations, while the low-level sequential nature of the code accommodates the code generation phase.

8.2 Complications with the CPS approach

As previously pointed out in section 7, we are having some trouble with using the stack. A good compiler should be able to allocate some variables on the stack, while our compiler will always allocate variables that cannot be kept in registers on the heap. The problem is that it is unclear when storage on the stack can be released (i.e. when to "pop" the stack). Normally this would be done upon return from a procedure call. In CPS it is not clearly distinguishable which calls are "real calls" and which calls are "procedure returns". Continuations are not theoretically distinguishable from "normal" procedures. This makes compilation towards a stack oriented target language more complicated to implement. What would solve the problem is a sufficiently powerful analysis (this is certainly not trivial!) that can categorise lambda's into 3 categories in stead of 2: register allocatable (currently class 1), heap-allocatable (class 2), and stack-allocatable.

9 Related work

Our whole system is modelled after the techniques employed in Orbit [9]. One difference is in the handling of Scheme's LETREC construct to create locally recursive procedures. In Orbit LETREC constructs are regarded as primitive expression types in order to be able to implement them efficiently. In our opinion this is contradictory to the reasoning of transforming source code into a form with as few types of expression as possible. Thus we did not regard LETREC as primitive since it can be expressed in terms of SET! and LAMBDA. The disadvantage of this is of course that one will have to make sure that suitable optimisation techniques will combine to optimise the code resulting from LETRECs efficiently. Although we did not succeed in doing this, because of lack of time, we do feel confident this should be possible. Very probably the effort will prove to be worthwhile because the extra optimisations introduced will not only optimise LETREC but will also be useful on their own and combine to optimise various other constructs.

Another difference is in the handling of continuation lambda's (see also section 8.2). Orbit treats continuation lambda's as special cases. This allows for allocating closures for such lambda's on the stack, thus gaining efficiency (less garbage collection). We chose not to do this because there is no theoretical ground to distinguish them from other lambda's. In fact Orbit employs the heuristic that most of the time these closures can be allocated and deallocated in a stack-like manner. Sometimes however (when call/cc is used) this is not true. Therefore Orbit has to provide a more complicated and more inefficient implementation of call/cc, involving copying the entire runtime stack. It is our opinion that it is possible to devise a strategy for categorising some lambda's as being stack-allocatable, without treating continuations as special cases.

10 References

[1] Abelson, H, Sussman, G.J. and Sussman, J.; Structure and Interpretation of *Computer Programs*; MIT Press; 1985.

[2] Aho, A.V., Sethi, R., Ullman J.D.; *Compilers Principle, Techniques and Tools*; Addison-Wessley; 1986.

[3] Bartley, D.H., Jensen J.C.; *The Implementation of PC Scheme*; ACM Conf. on Lisp and Functional programming; 1986.

[4] Clinger, W.; *Hygienic Macros Through Explicit Renaming*, Lisp Pointers IV(4), 17-23, December 1991.

[5] Clinger, W.; *Macros in Scheme*; Lisp Pointers IV(4), 25-28, December 1991.

[6] Hieb, Dybvig and Bruggeman; *Syntactic Abstraction in Scheme*, Computer Science Department, Indiana University, June 1992 [revised 7/3/92].

[7] Danvy o. and Filinski A.; *Representing Control: A study of the CPS transformation*, CIS-91-2, Department of Computing and Information Sciences, Kansas State University, February 1991, Revised June 1992. To appear in the journal Mathematical Structures for Computer Science.

[8] Dybvig K.; *Writing Hygienic Macros in Scheme with Syntax-Case*, Computer Science Department, Indiana University, June 1992.

[9] Kranz, D., *et al.* ; *ORBIT: An Optimizing Compiler for Scheme*; Proceedings of SIGPLAN '86 Symposium on Compiler Construction; June 1986.

[10] Norvig, P.; *Paradigms of Artificial Intelligence Programming* (chapter 22: "*Scheme an Uncommon Lisp*"; chapter 23: "*Compiling Lisp*"); Morgan Kaufmann Publishers, Inc. ; 1992.

[11] Rees, J. and Clinger W. (editors); *Revised^3 Report on the Algorithmic Language Scheme*; ACM SIGPLAN Notices 12(7); 1986.

[12] Clinger W. and Rees J. (editors), J. ,*et al.* ; *Revised^4 Report on the Algorithmic Language Scheme*, LISP Pointers, IV(3):1-55, July-September 1991.

[13] Sabry & Felleisen; *Reasoning about Programs in Continuation-Passing Style;* Rice University, May 1992.

[14] Shivers, O.; *Control Flow Analysis in Scheme*; Proceedings of the SIGPLAN conf. on Programming Language Design and Implementation; 1988.

[15] Springer, G., Friedman D.P.; *Scheme and the Art of Programming*; MIT Press; 1989.

[16] Steele, G.L.Jr.; *Rabbit: a Compiler for Scheme*; MIT AI Memo 474; Cambridge, Mass; May 1978

[17] Steele, G.L.Jr.; *Compiler Optimisation Based on Viewing LAMBDA as RENAME + GOTO*; in *AI: An MIT Perspective*; Winston, P.H. and Brown, R.H. (editors); Cambridge, Mass. ,1980.

[18] Teodosiu, D.; *HARE: An Optimising Portable Compiler for Scheme*, ACM SIGPLAN Notices Vol. 26 Nr. 1; Jan. 1991.