Compiling Laziness by Partial Evaluation

Anders Bondorf *
DIKU, Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
e-mail: anders@diku.dk

December 5, 1990

Abstract

One application of partial evaluation is compilation: specializing an interpreter with respect to a source program yields a target program. And specializing the partial evaluator itself with respect to an interpreter yields a compiler (self-application).

Similix [3] [1] is a self-applicable partial evaluator for a higher order subset of the strict functional language Scheme [16]. It was demonstrated in [1] how Similix can be used to specialize interpreters for non-strict functional languages. Such interpreters implement non-strict argument evaluation by using the well-known suspension technique of wrapping an expression `E` into a lambda: `(lambda () E)`. These suspensions give call-by-*name* reduction in the interpreters and in target programs obtained by partial evaluation.

Using `delay/force` [6], interpreters can also implement call-by-*need* (lazy) reduction. Unfortunately, `delay` is a side effecting operation that uses features not handled by existing self-applicable partial evaluators for higher order functional languages (such as Similix). In this paper we show that Similix is in fact strong enough to handle the limited use of side effects made by `delay` in interpreters. Correct target programs are generated because `delay` is a dynamic operation, because Similix guarantees against computation duplication and discarding, and because Similix preserves evaluation orders of evaluation order dependent expressions.

It is therefore possible to specialize interpreters for lazy languages. Using self-application, compilers (written in Scheme) from lazy languages into Scheme can be generated automatically from interpreters.

1 Introduction

Partial evaluation is a program transformation that *specializes* programs: given a *source* program and a part of its input (the *static* input), a partial evaluator generates a *residual* program. When applied to the remaining input (the *dynamic* input),

*This work was supported by ESPRIT Basic Research Actions project 3124 "Semantique"

the residual program yields the same result as the source program would when applied to all of the input. *Autoprojection* is a synonym for *self-applicable* partial evaluation, that is, specialization of the partial evaluator itself. It was established in the seventies that autoprojection can be used for automatic semantics directed compiler generation: specializing a partial evaluator with static input being (the text of) an *interpreter* for some programming language S yields a *compiler* for S [8] [7] [18]. Specializing the partial evaluator with static input being (the text of) the partial evaluator itself even yields a compiler generator (automatic compiler generator generation!).

The first successfully implemented autoprojector was *Mix* [11] [12]. The language treated by Mix was a subset of statically scoped first order pure Lisp, and Mix was able to generate compilers from interpreters written in this language. Since then, autoprojectors for several languages have been implemented, recently also for higher order functional languages (*Lambda-mix* [10], *Similix-2* [1], and *Schism* [5]).

Similix(-2) [3] [1] is an autoprojector for a higher order subset of Scheme [16]. Both source and residual programs are written in this Scheme subset. Similix handles side effecting operations on global variables, but otherwise it only treats a pure functional Scheme subset. In particular, Similix does not handle programs with `set!` operations on local variables.

It is well-known that call-by-name reduction can be achieved in otherwise strict languages by wrapping expressions into lambda's — in for instance Scheme by the *suspension* `(lambda () E)`. Here `E` is not evaluated until the suspension is "forced", that is, applied to an empty argument list. One application of such suspensions is in interpreters for languages with call-by-name reduction. By partially evaluating such an interpreter, one can compile from a non-strict into a strict language [1]. Notice that the partial evaluator must be able to treat higher order constructs to handle the expression form `(lambda () E)`.

Suspensions of the form `(lambda () E)` implement call-by-*name* reduction. However, lazy functional languages such as described in for instance [15] are based on call-by-*need* evaluation: a function argument is never evaluated more than once. It is possible also to implement call-by-need by using so-called "thunks" [6], local variables that memoize the value of the first application of the suspension. To do this, however, the side effecting operation `set!` is needed. This operation is not handled by existing autoprojectors for higher order functional languages (at least not by the ones we know of [10] [1] [5]), so how can we partially evaluate an interpreter that implements lazy evaluation by `set!`?

One could of course extend an existing autoprojector for a higher order functional language to be able to handle `set!` in general — but this would be cracking a nut with a sledge hammer. For our purpose, compiling laziness by partial evaluation, `set!` is only needed in a particular context, namely in interpreters to implement thunks. In this paper we show (informally) that Similix in its current form actually already is strong enough to handle this limited use of `set!`.

The consequence of this nice feature is that we can partially evaluate interpreters that implement call-by-need reduction by `set!`. And, using self-application, we can generate compilers (written in Scheme) from lazy functional languages into Scheme.

1.1 Outline

In section 2, we introduce a lazy example language $\frac{\mathcal{LA}}{\mathcal{ZY}}$. $\frac{\mathcal{LA}}{\mathcal{ZY}}$ is implemented by an interpreter written in Scheme. The interpreter uses thunks to implement laziness. In section 3, we argue that Similix in its current form already gives correct treatment of the thunk operation that uses `set!`. We give an example of specialization of the interpreter in section 4; this generates a quite readable target program which is shown in full. The performance is compared to Lazy ML and Miranda[1]. Finally, we conclude in section 5.

1.2 Prerequisites

Some knowledge about partial evaluation is desirable, e.g. as presented in [11] or [12]. Knowledge about Scheme is required (see e.g. [6]).

2 An Example Interpreter for the Language $\frac{\mathcal{LA}}{\mathcal{ZY}}$

To illustrate the use of `delay` we now present a lazy curried named combinator language $\frac{\mathcal{LA}}{\mathcal{ZY}}$ and an interpreter for $\frac{\mathcal{LA}}{\mathcal{ZY}}$ written in the Scheme subset treated by Similix [1]. A $\frac{\mathcal{LA}}{\mathcal{ZY}}$-program is a list of curried function definitions following the (abstract) syntax given in figure 1. An expression is a constant, a variable, a function name, a strict binary operation, a conditional, or a function application. For an example, the program in figure 2 computes a list of the first `n` even numbers; `goal` is the program's goal function.

```
P ∈ Program,  D ∈ Definition,  E ∈ Expression,
C ∈ Constant,  V ∈ Variable,  F ∈ FuncName,  B ∈ Binop

P  ::=  D*
D  ::=  (F V* = E)
E  ::=  C | V | F | (B E1 E2) | (if E1 E2 E3) | (E1 E2)
```

Figure 1: Abstract syntax of $\frac{\mathcal{LA}}{\mathcal{ZY}}$

Some syntactic sugar, expanded by a parser, is used. First, if the body expression `E` of a function definition `D` is compound, the outermost parentheses are omitted. Second, $(E_1\ E_2\ \ldots\ E_n)$ abbreviates a nested application $(\ldots(E_1\ E_2)\ \ldots\ E_n)$.

```
(first-n n l  = if (= n 0)
                  '()
                  (cons (lazy-car l) (first-n (- n 1) (lazy-cdr l))))

(evens-from n = lazy-cons n (evens-from (+ n 2)))

(lazy-cons x y z = z x y)
(lazy-car x      = x 1st)
```

[1]Miranda is a trademark of Research Software Ltd.

```
(lazy-cdr x      = x 2nd)

(1st x y = x)
(2nd x y = y)

(goal input = first-n input (evens-from 0))
```

Figure 2: evens source program written in $\frac{\mathcal{CA}}{\mathcal{ZY}}$

2.1 $\frac{\mathcal{CA}}{\mathcal{ZY}}$-interpreter

The interpreter is given in figure 3. Syntax accessors (`D-V*`, `Ecst-C`, etc.), syntax predicates (`isEcst?`, `isEvar?`, etc.), and `ext` have been defined externally as primitive operations.

The interpreter is written in a compositional (denotational) way. To implement recursion, a recursive function environment is built using the usual applicative order fixed point operator (see function `_P`).

The function `_D*` traverses a list of function definitions. When the list is empty, `_D*` returns the empty function environment. Otherwise, it updates the function environment by binding the function name (`(D-F D)`) to its value (computed by `_V*`), and it recurses over the rest of the definitions.

`_V*` builds a curried Scheme function (possibly with zero arguments) that implements a curried $\frac{\mathcal{CA}}{\mathcal{ZY}}$-function. The environment `r` is initially empty and is updated for each function argument. When all arguments have been supplied, the function body is evaluated by calling `_E`.

`_E` evaluates a constant expression by simply returning its value (which is found by accessing the expression's abstract syntax, by `(Ecst-C E)`). Variables are looked up in the variable environment and are then "forced": to implement lazy evaluation, the values kept in the variable environment are suspensions rather than values. Functions are looked up in the function environment. Binary operations are implemented by the (external) primitive operation `ext`. Conditionals are implemented by Scheme conditionals. Applications are implemented by Scheme applications, but notice that the argument is "delayed" (generating a suspension).

```
;       Basic = Integer + Boolean + Basic* + ...
; v:    Value = Basic + (Suspension → Value)
; s:    Suspension = Unit → Value
; r:    VarEnv = Variable → Suspension
; phi:  FuncEnv = FuncName → Value

; Program × FuncName × Value → Value
(define (_P P F v)
  (((fix (lambda (phi) (_D* P phi))) F) (my-delay v)))

; Definition* × FuncEnv → FuncEnv
(define (_D* D* phi)
  (if (null? D*)
```

```
      (init-env)
      (let ((D (car D*)))
        (upd-env (D-F D)
                 (_V* (D-V* D) (D-E D) (init-env) phi)
                 (_D* (cdr D*) phi)))))

; Variable* × Expression × VarEnv × FuncEnv → Value
(define (_V* V* E r phi)
  (if (null? V*)
      (_E E r phi)
      (lambda (s)
        (_V* (cdr V*) E (upd-env (car V*) s r) phi))))

; Expression × VarEnv × FuncEnv → Value
(define (_E E r phi)
  (cond
   ((isEcst? E)
    (Ecst-C E))
   ((isEvar? E)
    (my-force (r (Evar-V E))))
   ((isEfct? E)
    (phi (Efct-F E)))
   ((isEbinop? E)
    (ext (Ebinop-B E) (_E (Ebinop-E1 E) r phi) (_E (Ebinop-E2 E) r phi)))
   ((isEif? E)
    (if (_E (Eif-E1 E) r phi)
        (_E (Eif-E2 E) r phi)
        (_E (Eif-E3 E) r phi)))
   ((isEapply? E)
    ((_E (Eapply-E1 E) r phi) (my-delay (_E (Eapply-E2 E) r phi))))
   (else
    (error ...))))

; Applicative order fixed point operator:
; (FuncEnv → FuncEnv) → FuncEnv
(define (fix f) (lambda (x) ((f (fix f)) x)))
```

Figure 3: $\frac{CA}{ZY}$-interpreter written in Scheme

Environment initialization and updating are defined as syntactic extensions (macros) following the syntax of [14]:

```
(extend-syntax (init-env) ((init-env) (lambda (name) (error ...))))

(extend-syntax (upd-env)
  ((upd-env name value r)
   (lambda (name1)
     (if (equal? name name1)
         value
```

```
        (r name1)))))
```

Figure 4: Environment initialization and updating

In the interpreter, we have sloppily used the same syntactic extension `init-env` to initialize both variable and function environments. This plays no role in practice: the interpreter works on a parsed and scope checked program in which lookup errors cannot possibly occur. (The `error` call in `_E` is never reached in practice either.)

2.2 Delaying and forcing

`my-delay` builds and `my-force` applies suspensions. They are also defined as syntactic extensions; it is important that `my-delay` is non-strict, so it cannot be defined as a function.

```
(extend-syntax (my-delay)
  ((my-delay value) (save (lambda () value))))

(extend-syntax (my-force) ((my-force delayed-value) (delayed-value)))
```

Here `save` is a primitive operation implemented in Scheme as follows:

```
(lambda (s)
  (let ((thunk s))
    (lambda ()
      (let ((v (thunk)))
        (set! thunk (lambda () v))
        v)))))
```

The first time the saved lambda expression is applied (by `my-force`), the lambda expression is effectively overwritten by a new lambda expression `(lambda () v)`. Here `v` is the value `E` evaluates to. This achieves lazy evaluation: all subsequent "forces" will apply the suspension `(lambda () v)` rather than `(lambda () E)`, so `E` is evaluated at most once.

`my-delay` is a slightly modified version of the well-known `delay` described in [6] under "delayed evaluation". The difference is that we have put the part involving `set!` into the primitive strict operation `save`. A primitive strict operation is a syntactic form already known to the partial evaluator Similix, so by "hiding" `set!` inside a primitive, Similix need not be extended to handle an additional syntactic form. The remaining problem is that `save` involves side effects on the local variable `thunk`; Similix does not in general guarantee to handle such a primitive in a semantically correct way. However, as we shall see in section 3, the `save` primitive is actually handled correctly.

Notice that `save` does not depend on, nor does it change any externally observable state. We say that `save` is *evaluation order independent*. Each application of `save` creates a new local `thunk` variable which is independent of other thunks.

2.3 Memoization

save is a simple *memoization operator* (see for instance [9]) which memoizes a nullary function. Memoizing a pure function has no effect except efficiency: the memoized version *memo(f)* of a function f is *observationally equivalent* to (has the same input/output behavior as) f, but it may compute its result faster. Observationally, **save** is equivalent to the identity when applied to pure functions.

Memoizing an impure function does have an observational effect: only the first application of *memo(f)* may depend on or change a state. Subsequent applications will just return the value returned by the first application. *memo(f)* is thus not observationally equivalent to f if f is impure. Notice, however, that save itself is still evaluation order independent.

2.4 Optimizing the Interpreter

By investigating the interpreter text, it is simple to see that the program piece

```
((isEapply? E)
 ((_E (Eapply-E1 E) r phi) (my-delay (_E (Eapply-E2 E) r phi))))
```

can be optimized. Optimizing the interpreter results in better target code when the interpreter is specialized with respect to source programs (written in $\frac{\mathcal{LA}}{\mathcal{ZV}}$).

First notice that there is no reason to use **save** when the argument of the application is a constant expression. A constant expression is evaluated by a single access **(Ecst-C E2)**, and this operation is cheaper than performing the sequence

```
(let ((v (thunk)))
  (set! thunk (lambda () v))
  v))))
```

Therefore a constant argument can be delayed with the call-by-name delay **my-delay-cbname** defined by

```
(extend-syntax (my-delay-cbname)
  ((my-delay value) (lambda () value)))
```

my-delay-cbname can also replace **my-delay** in the function **_P**:

```
(define (_P P F value)
  (((fix (lambda (phi) (_D* P phi))) F) (my-delay-cbname value)))
```

Second, when the argument is a variable,

```
(my-delay (_E (Eapply-E2 E) r phi))
```

reduces to

```
(my-delay (my-force (r (Evar-V (Eapply-E2 E)))))
```

which can be simplified to `(r (Evar-V (Eapply-E2 E)))`. This is the new simplified piece of code in `_E`:

```
((isEapply? E)
 (let ((E2 (Eapply-E2 E)))
   ((_E (Eapply-E1 E) r phi)
    (cond
     ((isEcst? E2)
      (my-delay-cbname (Ecst-C E2)))
     ((isEvar? E2)
      (r (Evar-V E2)))
     (else
      (my-delay (_E E2 r phi)))))))
```

Figure 5: Optimized piece of interpreter code

3 Partial Evaluation of Programs with Thunks

We now argue that Similix handles **save** correctly.

3.1 Making save operations dynamic

We can simplify the problem of handling **save** by ensuring that **save** expressions always become residual (dynamic), that is, never get reduced at partial evaluation time. This has the consequence that functions memoized by **save** always become dynamic, that is, they are never applied at partial evaluation time.

We believe that this simplification is reasonable when considering interpreters: in these, **save** operations are typically used only for traditional run time (dynamic) operations, not for traditional compile time (static) operations such as syntax analysis and environment manipulation. This is at least the case for the $\frac{CA}{ZY}$-interpreter: **save** is only used to memoize the function which suspends the argument to the dynamic (run time) application that implements application in the interpreted language. We do not expect this suspension to be reduced at partial evaluation (compile) time.

In Similix it is possible to specify that a primitive operation should always be considered dynamic [3], even when called with only non-dynamic arguments. By doing this for **save**, we are guaranteed that **save** expressions always become residual. Hence, expressions that apply memoized (saved) functions (for instance `my-force` expressions) always become dynamic too. Potential problems of performing side effects on thunk variables statically are thus eliminated: the code implementing memoization in **save** is never executed at partial evaluation time.

3.2 Pure functions

When applied to pure functions, **save** observationally acts like the identity operator, in source program expressions as well as in residual program expressions. Therefore

observational equivalence between source and residual code is trivially preserved in this case.

Notice, however, that save was introduced for *efficiency*: preserving observational equivalence between source and residual code is *not* sufficient to ensure a "correct" treatment of save. Efficiency is a correctness issue here: if the memoization implemented by save in source programs is lost in residual programs, then we do not consider this a correct treatment of save.

Unfolding let-expressions and function calls is important in partial evaluation: it gives smaller and faster residual code. Unfolding may, however, make residual programs behave differently from the source programs. The reason is that unfolding in general may *duplicate* or *discard* (residual) expressions, and it may change the *evaluation order* of (residual) expressions. Similix has a property which is vital for the correct treatment of save from an efficiency point of view:

- No residual expression is ever duplicated nor discarded [3].

This property ensures that whenever the source program memoizes a function with save (whereby the function also becomes dynamic), the residual program is guaranteed to memoize the residual version of the (dynamic) function (no discarding), and it will do so only once (no duplication).

The important property is the "no duplication" one: duplicating a save expression (for instance by unfolding a let-expression with actual parameter expression being a save expression) would imply that a function was memoized more than once, thus losing efficiency.

3.3 Impure functions

When applied to impure functions, save is not observationally equivalent to the identity (cf. section 2.3). Similix allows impure functions that operate on global variables, so save might be applied to such functions in programs that are partially evaluated. Therefore save must be treated correctly by Similix, also when applied to impure functions.

When partially evaluating programs with impure functions, the "no duplication" and "no discarding" properties are necessary, not only for efficiency, but also to preserve observational equivalence between source and residual programs. In addition to this, preserving evaluation orders is necessary. This is described in detail in [3].

Two kinds of (dynamic) expressions need to be considered in connection to save: (*1) save expressions and (*2) expressions that apply memoized (saved) functions (for instance my-force expressions). Application expressions (*2) may now evaluate the body of an impure function, so they may neither be discarded nor duplicated, and their evaluation order must be preserved. save expressions (*1) may not be duplicated, not only because of efficiency (as in section 3.2), but, perhaps surprisingly, also to preserve observational equivalence.

Let us now address these points in more detail:

- *Discarding*

(*2) When applying an impure (possibly memoized) function, side effects may be performed if this is the first time the memoized function is applied. Such an application must not be discarded, even if the result is not used.

Similix never discards residual expressions.

- *Duplication*

  (*1) The impure operations in a function memoized by save are performed only once in the source program (the first time the memoized function is applied). The residual program must have exactly the same behavior. This requires that save expressions are not duplicated. If they were, we might generate several memoized versions of an impure function in the residual program.

  (*2) Since side effects may be performed when applying an impure (possibly memoized) function, such applications may in general not be duplicated.

  Similix never duplicates residual expressions.

- *Evaluation order*

  (*2) Applications of memoized impure functions are evaluation order dependent. Evaluation order dependent (dynamic) expressions are treated by Similix in the following way: as other dynamic expressions, they are never duplicated nor discarded; in addition to this, the order of evaluation of such expressions is always preserved.

  In binding time analysis, potentially evaluation order dependent dynamic expressions get a binding time value "X" which is greater (more conservative) in the lattice of binding time values than "D" (dynamic). Applications of impure functions get binding time value X. Applications of impure functions memoized by save also get binding time value X: applying a primitive (such as save) never lowers a binding time value in Similix. This means that the evaluation order of applications of memoized impure functions is preserved in residual programs.

To conclude, Similix preserves the semantics of save correctly, even when save is applied to impure functions. It does so because of its "no duplication", "no discarding", and evaluation order preserving properties.

4 Specializing the $\frac{\mathcal{LA}}{\mathcal{ZY}}$-Interpreter

Using Similix to specialize the $\frac{\mathcal{LA}}{\mathcal{ZY}}$-interpreter with respect to the *evens* program yields an efficient target program. Computing the list of the first 20 even numbers by the target program is around 14 times faster than by interpreting the source program using the interpreter. The text of the target program is given below (figure 6); function and variable names have been renamed by hand for readability, but otherwise the program was generated completely automatically. Notice that `(save (lambda () E))` corresponds to `(my-delay E)`, and that some applications `(E)` correspond to `(my-force E)`. `my-delay` and `my-force` are macros which are expanded before partial evaluation; therefore they do not occur in specialized programs.

```
(define (goal input)
   (((first-n) (lambda () input))
    (save (lambda () ((evens-from) (lambda () 0))))))

(define (evens-from)
   (lambda (n)
      (let ((result
               (save (lambda ()
                        ((evens-from) (save (lambda ()
                                              (ext '+ (n) 2))))))))
         (lambda (z) (((z) n) result)))))

(define (first-n)
   (lambda (n)
      (lambda (l)
         (if (ext '= (n) 0)
             '()
             (ext 'cons
                  ((l)
                   (save (lambda () (lambda (x) (lambda (y) (x))))))
                  (((first-n) (save (lambda () (ext '- (n) 1))))
                   (save (lambda ()
                           ((l)
                            (save (lambda ()
                                    (lambda (x)
                                      (lambda (y) (y))))))))))))))
```

Figure 6: Machine produced evens target program written in Scheme

The program text corresponds closely to the source program (written in $\frac{\mathcal{CA}}{\mathcal{ZY}}$), the main difference being the many delays and forces (nullary applications).

By self-applying the partial evaluator, a stand-alone compiler is generated. Generating the target program using the stand-alone compiler is around 7 times faster than by applying the specializer to the interpreter. The size of the compiler is 10.6 Kbytes.

4.1 Comparing to other implementations

To get some idea of the efficiency of the target programs we generate, we have compared with Lazy ML (version 0.97) and Miranda (version 2.014). The Scheme system used is Chez Scheme (version 3.2). All runs are done on SPARC stations.

We have used a test program based on the *evens* program. Instead of returning a list of even numbers, the test program returns the sum of the first 1000 even numbers. This is done to avoid having the printing time being an important part of the run time.

Computing the sum 10 times takes around 0.9 seconds using Lazy ML, around 9 seconds using Miranda, and around 7 seconds using our Scheme target program. The Scheme program is thus faster than Miranda, but much slower than Lazy ML.

Notice, however, that the Scheme target program is far from being optimal. Firstly, arguments (for instance to `first-n`) may be *tupled* to avoid unnecessary currying. This can be done either by postprocessing target programs or by preprocessing $\frac{\mathcal{CA}}{\mathcal{ZY}}$-source programs to find tupled applications; the latter approach requires extending the interpreter to implement tupled applications.

Secondly, *strictness analysis* (see for instance [15]) may be used to avoid delaying and forcing arguments when this is not necessary. For instance, `first-n` is strict in its first argument `n`, so delaying and forcing this argument could be avoided. We could apply strictness analysis to source programs and then extend the interpreter to handle strict arguments differently from non-strict ones. This would then result in better, more reduced, target programs.

It should be noted that the test program cannot be typed by Lazy ML and Miranda (the problem is the recursive `evens-from` definition). To make the program run in Lazy ML, we simply switched off the type checking (this is possible to do in Lazy ML). This is not as bad as it sounds: we have not addressed compiling typed languages, so using an "untyped Lazy ML" seems reasonable for comparison. To get the program through the Miranda type checker, we redefined `lazy-cons`, `lazy-car`, and `lazy-cdr` to the built-in Miranda list operations `:`, `hd`, and `tl`. This makes the comparison less fair since Miranda presumably has efficient implementations of its built-in operations — and yet Miranda is still the slowest of the three.

5 Conclusion

We have demonstrated how to handle *dynamic* laziness: all (side effecting) computations involving `my-force`/`my-delay` were suspended till run time. We have not addressed *static* laziness; to handle this, it would be necessary to extend Similix to be able to execute `set!` operations statically.

The method presented in this paper could easily be used in other, more complex interpreters. For instance, one could write a self-interpreter for Scheme. This interpreter could then be changed to implement lazy evaluation; by specializing the lazy interpreter, one would compile from lazy Scheme into (standard) Scheme.

The idea can also be used to implement other lazy languages by partial evaluation into Scheme. Similix has already been used to compile a subset of an Orwell-like language into Scheme [13].

Acknowledgements

I would like to thank members of the "TOPPS/Semantics" group at DIKU, in particular Lars Ole Andersen, Jesper Jørgensen, Torben Mogensen, and Peter Sestoft. Also thanks to John Launchbury and John Hughes for their useful comments.

References

[1] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 1991. Accepted for publication, to appear. Journal version of [2].

[2] Anders Bondorf. Automatic autoprojection of higher order recursive equations. In Neil D. Jones, editor, *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark. Lecture Notes in Computer Science 432*, pages 70–87, Springer-Verlag, May 1990.

[3] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 1991. Accepted for publication, to appear. Journal version of [4].

[4] Anders Bondorf and Olivier Danvy. *Automatic autoprojection of recursive equations with global variables and abstract data types.* Technical Report 90-4, DIKU, University of Copenhagen, Denmark, 1990.

[5] Charles Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Languages. Nice, France*, pages 264–272, June 1990.

[6] R. Kent Dybvig. *The SCHEME Programming Language.* Prentice-Hall, New Jersey, 1987.

[7] Andrei P. Ershov. On the partial computation principle. *Information Processing Letters*, 6(2):38–41, April 1977.

[8] Yoshihiko Futamura. Partial evaluation of computing process — an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[9] John Hughes. Lazy memo-functions. In Jean-Pierre Jouannaud, editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy, France. Lecture Notes in Computer Science 201*, pages 129–146, Springer-Verlag, 1985.

[10] Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Æ. Mogensen. A self-applicable partial evaluator for the lambda calculus. In *IEEE Computer Society 1990 International Conference on Computer Languages*, pages 49–58, IEEE, March 1990.

[11] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In Jean-Pierre Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202*, pages 124–140, Springer-Verlag, 1985.

[12] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: a self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[13] Jesper Jørgensen and Lars Mathiesen. *Generating a compiler for a lazy functional language.* Student Report 90-5-16, DIKU, University of Copenhagen, Denmark, November 1990.

[14] Eugene E. Kohlbecker. *Syntactic Extensions in the Programming Language Lisp.* PhD thesis, Indiana University, Bloomington, 1986.

[15] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages. Computer Science*, Prentice-Hall, 1987.

[16] Jonathan Rees and William Clinger. Revised report[3] on the algorithmic language Scheme. *Sigplan Notices*, 21(12):37–79, December 1986.

[17] David A. Schmidt. *Denotational Semantics, a Methodology for Language Development.* Allyn and Bacon, Boston, 1986.

[18] Valentin F. Turchin. Semantic definitions in Refal and the automatic production of compilers. In Neil D. Jones, editor, *Workshop on Semantics-Directed Compiler Generation, Århus, Denmark. Lecture Notes in Computer Science 94*, pages 441–474, Springer-Verlag, January 1980.