Higher-Order Functions as a Substitute for Partial Evaluation (A Tutorial)*

Sergei A. Romanenko

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
4 Miusskaya sq., Moscow, 125047, Russia
romansa@keldysh.ru

Abstract. This tutorial shows how to rewrite an interpreter written in a higher-order functional language, so that it will become more similar to a compiler, thereby eliminating the overhead due to interpretation.

1 Defining a language by means of an interpreter

When writing programs in a functional language, it is fairly easy to "extend" the language by defining an interpreter `run`, which will take a program `prog`, and some input data `d`, and return the result of applying `prog` to `d`:

```
run prog d
```

Hence, in this way the programmer can include in his program pieces written in the language implemented by `run`. `run` is usually said to give an *operational* semantics for the language thus defined.

Unfortunately, an interpreter written in a straightforward way is likely to introduce a considerable overhead.

However, the overhead can be reduced by refactoring a naïve interpreter in such a way that it becomes more similar to a compiler. The refactoring is based on replacing some first-order functions with higher-order ones.

2 An example interpreter

For the user to feel comfortable, `run` should accept programs written in human-oriented form, which can be achieved with the aid of *quotation/antiquotation* mechanism as usually implemented by Standard ML implementations. The techniques of translating programs from the "concrete" syntax into the abstract syntax are well known, and will not be considered in this paper. Hence, for the sake

* Supported by Russian Foundation for Basic Research projects No. 06-01-00574-a and No. 08-07-00280-a and Russian Federal Agency of Science and Innovation project No. 2007-4-1.4-18-02-064.

```
datatype exp =
    INT of int
  | VAR of string
  | BIN of string * exp * exp
  | IF of exp * exp * exp
  | CALL of string * exp list

type prog =
  (string * (string list * exp)) list;
```

Fig. 1. Abstract syntax of programs.

```
val fact_prog =
[
("fact", (["x"],
  IF(
    BIN("=", VAR "x", INT 0),
    INT 1,
    BIN("*",
      VAR "x",
      CALL("fact",
        [BIN("-", VAR "x", INT 1)]))))
  ))
];
```

Fig. 2. A program in abstract syntax.

of simplicity, `run` is supposed to accept programs represented by abstract syntax trees.

As an example, we shall consider a function `run` having type

```
val run : prog -> int list -> int
```

A program will be a list of mutually recursive first-order function definitions, each function accepting a fixed number of integer arguments, and returning an integer. The abstract syntax of programs is shown in Figure 1.

For example, the well-known factorial function

```
fun fact x =
  if x = 0 then 1 else x * fact (x-1)
```

when written in abstract syntax, takes the form shown in Figure 2. Combining the interpreter `run` and the program `fact_prog`, we can define the function `fact` computing factorials of integers:

```
fun eval prog ns exp vs =
  case exp of
    INT i => i
  | VAR n =>
      getVal (findPos ns n) vs
  | BIN(name, e1, e2) =>
      (evalB name) (eval prog ns e1 vs,
                    eval prog ns e2 vs)
  | IF(e0, e1, e2) =>
      if eval prog ns e0 vs <> 0
      then eval prog ns e1 vs
      else eval prog ns e2 vs
  | CALL(fname, es) =>
      let
        val (ns0, body0) =
          lookup prog fname
        val vs0 =
          evalArgs prog ns es vs
      in eval prog ns0 body0 vs0 end

and evalArgs prog ns es vs =
  map (fn e => eval prog ns e vs) es

fun run (prog : prog) vals =
  let val (_, (ns0, body0)) = hd prog
  in eval prog ns0 body0 vals end
```

Fig. 3. First-order interpreter.

```
fun fact x = run fact_prog [x];
fact 4;
```

The interpreter `run` can be defined in a straightforward way (see Figure 3). Some auxiliary declarations used in this interpreter (and further examples) can be found in Figures 4 and 5.

3 Denotational definition

If the program being executed contains a loop, the interpreter may analyze the same fragments of the source program again and again, which slows down the execution. Let us try to eliminate this overhead by rewriting our interpreter in *denotational* stvle.

```
fun findPos ns n =
  let fun loop [] i = raise Fail "findPos"
        | loop (n0::ns) i =
            if n = n0 then i
                      else loop ns (i+1)
  in loop ns 0 end

fun getVal 0 vs = hd vs
  | getVal n vs = getVal (n-1) (tl vs)

fun lookup [] n = raise Fail "lookup"
  | lookup ((k,v) :: rest) n =
      if k=n then v else lookup rest n
```

Fig. 4. Look-up functions.

```
fun evalB "+" = op +
  | evalB "-" = op -
  | evalB "*" = op *
  | evalB "=" =
      (fn(x, y) => if x = y then 1 else 0)
  | evalB _  : int * int -> int =
      (raise Fail "evalB")
```

Fig. 5. Meaning of primitive operators.

3.1 What is a denotational definition?

A denotational definition is essentially a compiler that maps the source program *prog* into its "meaning" $[\![prog]\!]$, a function that, given the input data, will produce the result of running *prog* with that input.

There is an additional requirement any denotational definition must satisfy: namely, the meaning of each program fragment must be formulated in terms of the meanings of its constituent parts. The interpreter in Figure 3 violates this requirement, because the function `eval` takes as arguments both an expression and the whole program. Hence the meaning of an expression is defined via the meaning of the whole program.

This subtle point can be illustrated by contrasting two definitions of the Pascal construct `while exp do st`.

The semantics of statements can be given via a function `evalS`, which takes as arguments a statement `st` and a store `s`, and returns a new store `evalS st s`.

Figure 6 shows a version of `evalS` that is not denotational, because `evalS` recursively calls itself passing as argument the same fragment of the source

```
fun evalS (WHILE(exp, st)) s =
      if evalE exp s then
        evalS(WHILE(exp, st))
              (evalS st s)
      else s
  | evalS (ASSIGN(id, exp)) s = ...
  ...
```

Fig. 6. An operational definition of the `while` loop.

```
fun evalS (WHILE(exp, st)) s =
      let fun loop s =
        if evalE exp s then
          loop(evalS st s)
        else s
      in loop s end
  | evalS (ASSIGN(id, exp)) s = ...
  ...
```

Fig. 7. A denotational definition of the `while` loop.

program: the whole construct `while`. This definition, however, can be "rectified" by introducing an auxiliary function `loop` (see Figure 7). Now the meaning of `WHILE(exp,st)` is expressed in terms of the meanings of `exp` and `st` !

3.2 Turning the interpreter into a denotational definition

We may turn our interpreter into a denotational definition by replacing the parameter containing the text of the program with a *function environment* ϕ, mapping function names onto their meanings (see Figure 8). Hence, the meaning of an expression depends only upon the meanings of its constituent subexpressions (and is defined with respect to some function environment).

The only problem is how to find the function environment ϕ corresponding to the whole program. If the denotational definition is written in a lazy programming language, ϕ can be given a circular definition

```
val rec phi = ... phi ...
```

in which case `phi` will be found as the "least fixed point" of the above equation. But, if the denotational definition is to be written in a strict language (like SML), the right hand side of a recursive equation must be a λ-abstraction. This restriction will be satisfied, if we rewrite the equation as

```
fun eval phi ns exp vs =
  case exp of
    INT i => i
  | VAR n => getVal(findPos ns n) vs
  | BIN(name, e1, e2) =>
      (evalB name) (eval phi ns e1 vs,
                    eval phi ns e2 vs)
  | IF(e0, e1, e2) =>
      if eval phi ns e0 vs <> 0
      then eval phi ns e1 vs
      else eval phi ns e2 vs
  | CALL(fname, es) =>
      phi fname (evalArgs phi ns es vs)

and evalArgs phi ns es vs =
  map (fn e => eval phi ns e vs) es

fun run (prog : prog) =
  let
    fun phi fname =
      let val (ns, e) = lookup prog fname
      in eval phi ns e end
    val (_, (ns0, e0)) = hd prog
  in eval phi ns0 e0 end
```

Fig. 8. Denotational definition.

```
fun phi fname = ... phi ...
```

See the declaration of `run` in Figure 8 for technical details.

3.3 Representing loops by cyclic data structures

The drawback of the denotational definition in Figure 8 is that, instead of representing the loops appearing in the source program by a cyclic data structure, we, first, replace it with a non-cyclic—but infinite—tree, and then unroll that tree incrementally.

However, we can represent the function environment as a finite graph by making use of some "imperative features" of Standard ML (see Figure 9).

The constructor `ref` creates "memory locations". When applied to a value v, it creates a new location, v being the initial contents of the location, and returns a reference to the location. The function `!`, when applied to a reference, returns a copy of the contents of the corresponding location. The assignment E_1`:=`E_2 evaluates E_1, which must return a reference to a location, and E_2. Then the contents of the location is replaced with the value returned by E_2.

```
fun eval phi ns exp vs =
  case exp of
    INT i => i
  | VAR n => getVal (findPos ns n) vs
  | BIN(name, e1, e2) =>
      (evalB name) (eval phi ns e1 vs,
                    eval phi ns e2 vs)
  | IF(e0, e1, e2) =>
      if eval phi ns e0 vs <> 0
      then eval phi ns e1 vs
      else eval phi ns e2 vs
  | CALL(fname, es) =>
      let val r = lookup phi fname
      in (!r) (evalArgs phi ns es vs) end

and evalArgs phi ns es vs =
  map (fn e => eval phi ns e vs) es

fun dummyEval (vs : int list) : int =
  raise Fail "dummyEval"

fun app f [] = ()
  | app f (x :: xs) =
      (f x : unit; app f xs)

fun run (prog : prog) =
  let
    val phi =
      map (fn (n,_) => (n, ref dummyEval))
          prog
    val (_, r0) = hd phi
  in
    app (fn (n, (ns, e)) =>
          (lookup phi n) := eval phi ns e)
        prog;
    !r0
  end
```

Fig. 9. Using references to represent cycles in the call graph.

The function `run` builds the environment `phi` by creating a separate location for each function definition and associating the function's name with the location. Then the location is assigned the meaning of the function definition.

4 Separating binding times

4.1 Being denotational is not enough

Theoretically, the denotational definition in Figure 9 transforms a function into its meaning. But, if we examine it more closely, we can easily find out that it can hardly be called a "compiler": the function `eval` does not compute anything, before it has been given parameter `vs`, the values of variables.

One of the consequences is that, if the source program contains loops, the same subexpressions may be analyzed and "compiled" again and again.

We may however improve the definition, by applying a few techniques developed in the framework of lazy programming languages.

4.2 Binding times

When an expression like

```
(fn x => fn y => fn z => e)
```

is applied, `x` is bound before `y`, which again is bound before `z`. According to [Hol90], we call the variables that are bound first *early* and the ones that are bound later *late*. The early variables will be said to be more *static* than the late ones, whereas the late variables will be said to be more *dynamic* than the earlier ones.

4.3 Lifting static subexpressions

Consider the declarations

```
val h = fn x => fn y => sin x * cos y
val h' = h 0.1
val v = h' 1.0 + h' 2.0
```

When `h'` is declared, no real evaluation takes place, because the value of `y` is not known yet. Hence, `sin 0.1` will be evaluated twice, when evaluating the declaration of `v`. This can be avoided if we rewrite the declaration of `h` in the following way:

```
val h = fn x =>
  let val sin_x = sin x
  in fn y => sin_x * cos y end
```

The transformation of that kind (see [Hol90]), when applied to a program in a lazy language is known as transforming the program to a "fully lazy form"[1].

Now by lifting static subexpressions in the denotational definition of the `while` loop (shown in Figure 7), we can obtain an improved definition shown in Figure 10.

[1] Needless to say that in the case of a strict language such transformation may be unsafe, because it may change termination properties of the program. For example, if we replace `sin x` with `monster x`, where `monster` is an ill-behaved function, the evaluation of `monster 0.1` may never terminate!

```
fun evalS (WHILE(exp, st)) s =
      let
        val c1 = evalE exp
        val c2 = evalS st
        fun loop s =
          if c1 s then loop(c2 s)
                  else s
      in loop s end
  | evalS (ASSIGN(id, exp)) s = ...
  ...
```

Fig. 10. The result of lifting static subexpressions in the definition of the `while` loop.

4.4 Liberating control

Consider the expression

```
fn x => fn y =>
  if (p x) then (f x y) else (g x y)
```

If we apply the transformation described above, we can avoid reevaluating `(p x)`:

```
fn x =>
  let val p_x = p x
  in fn y =>
    if p_x then (f x y) else (g x y)
  end
```

The question is whether `f x` and `g x` should be lifted too. If we lift both `f x` and `g x`, this will result in unnecessary computation, because either the value of `f x` or `g x` will be thrown away. If e do not lift them, either `f x` or `g x` will be repeatedly reevaluated.

Another deficiency of the above solution is that the conditional remains inside the inner λ-abstraction. Hence, the choice between the two branches of the conditional is not made, until the value of `y` becomes known. (Despite the fact that the value of the test `p x` is evaluated as soon as the value of `x` has been supplied.)

Fortunately, this difficulty can be overcome by means of another trick: instead of lifting the test from within `fn y => ...`, we can push `fn y =>` over `if p x` into the branches of the conditional!

Thus the expression can be rewritten as:

```
fn x =>
  if p x then
    fn v => (f x v)
```

```
    else
      fn y => (g x y)
```

and then as

```
fn x =>
  if p x then
    let val f_x = f x
    in (fn y => f_x y) end
  else
    let val g_x = g x
    in (fn y => g_x y) end
```

which enables us to avoid unnecessary as well as repeated evaluation[2].

Similarly, `fn y =>` can be pushed into other control constructs, containing conditional branches. For example,

```
fn x => fn y =>
  case f x of
      A => g x y
    | B => h x y
```

can be rewritten as

```
fn x =>
  case f x of
      A => fn y => g x y
    | B => fn y => h x y
```

and then as

```
fn x =>
  case f x of
      A => let val g_x = g x
           in fn y => g_x y end
    | B => let val h_x = h x
           in fn y => h_x y end
```

The above transformation is usually applied to programs written in a lazy language to achieve "improved full laziness" [Hol89,Hol90], but can also be applied to programs in a strict language. In the latter case, however, it may not preserve termination properties of the program (which is also true of the transformations performed by some automatic program specializers).

[2] See, however, the previous footnote.

4.5 Separating binding times in the interpreter

Now let's return to the version of the interpreter in Figure 9, and try to separate the static computations, which depend only on the text of the source program, from the dynamic ones, which may also depend on the input data.

The function `run` is good enough already, and need not be revised. So let's consider the definition of the function `eval`. It has the form

```
fun eval phi ns exp vs =
  case exp of
    INT i => i
  ...
```

First of all, let's move `vs` to the right hand side:

```
fun eval phi ns exp =
  fn vs =>
  case exp of
    INT i => i
  ...
```

Now we can push `fn vs =>` into the `case` construct:

```
fun eval phi ns exp =
  case exp of
    INT i => (fn vs => i)
  ...
```

so that the right hand side of each match rule begins with `fn vs =>`, and can be transformed further, independently from the other right hand sides.

The final result of the transformations is shown in Figure 11. In the case of the rules corresponding to `INT`, `BIN`, and `IF`, the transformation is straightforward: we just lift static subexpressions. In the case of `VAR`, the right hand side takes the form

```
fn vs => getVal (findPos ns n) vs
```

and we can perform η-reduction

```
getVal (findPos ns n)
```

Then we have to improve the definition of `getVal`. Again, this can be done by moving `vs` to the right hand sides, and by applying η-reductions and lifting static subexpressions. The revised version of `getVal` is shown in Figure 11 under the name `getVal'`.

By the way, we could also circumvent the explicit lifting of static subexpressions, by formulating the definition of `getVal` in terms of the infix operation `o`, the composition of functions:

```
fun getVal' 0 = hd
  | getVal' n = getVal' (n-1) o tl
```

```
fun getVal' 0 = hd
  | getVal' n =
      let val sel = getVal' (n-1)
      in fn vs => sel (tl vs) end

fun eval phi ns exp =
  case exp of
    INT i => (fn vs => i)
  | VAR n =>
      getVal'(findPos ns n)
  | BIN(name, e1, e2) =>
      let val b = evalB name
          val c1 = eval phi ns e1
          val c2 = eval phi ns e2
      in (fn vs => b (c1 vs, c2 vs)) end
  | IF(e0, e1, e2) =>
      let val c0 = eval phi ns e0
          val c1 = eval phi ns e1
          val c2 = eval phi ns e2
      in fn vs =>
           if c0 vs <> 0 then c1 vs
                         else c2 vs
      end
  | CALL(fname, es) =>
      let
        val r = lookup phi fname
        val c = evalArgs phi ns es
      in fn vs => (!r) (c vs) end

and evalArgs phi ns [] = (fn vs => [])
  | evalArgs phi ns (e :: es) =
      let val c'  = eval phi ns e
          val c'' = evalArgs phi ns es
      in fn vs => c' vs :: c'' vs end
```

Fig. 11. The result of lifting static subexpressions.

This solution appears to be more elegant, but finding it requires more "insight". (Besides, it is less efficient.)

Now let's consider the right hand side of the rule corresponding to `CALL`.

```
fn vs =>
  let val r = lookup phi fname
  in (!r) (evalArgs phi ns es vs) end
```

Here the subexpressions `lookup phi fname` and `evalArgs phi ns es` are static, and we just lift them out of the λ-abstraction.

Finally, we have to transform the definition of `evalArgs`, which can be done in two steps. First, we can replace the call to the higher-order function `map` with explicit recursion:

```
and evalArgs phi ns [] vs = []
  | evalArgs phi ns (e :: es) vs =
      eval phi ns e vs ::
        evalArgs phi ns es vs
```

After that the techniques described above become applicable.

In the end, we come to the definition of `run` in Figure 11, which "compiles" the source program into a composition of λ-abstractions, representing the meaning of the source program. Since the revised `run` examines a fragment of the source program no more than once, it is much closer to a compiler, than to an interpreter.

5 Higher-order functions with separated binding times

When separating binding times in our example interpreter, we had to replace a call to the "general-puprose" functional `map` with explicit recursion. This is evidently against the spirit of the "high-order" programming, for the possibility to use functionals is one of its main attractive features.

5.1 Separating for free!

Holst and Hughes [HH90] suggest that binding times should be separated by applying commutative-like laws, which can be derived from the types of polymorphic functions using the "free-theorem" approach [Wad89].

In our case a suitable law is

```
map (d o s) xs = map d (map s xs)
```

because, if `s` and `xs` are static subexpressions, and `d` a dynamic one, then `map s xs` is a static subexpresion, which can be subsequently lifted out of the dynamic context.

In the interpreter in Figure 9, the expression

```
map (fn e => eval phi ns e vs) es
```

can be transformed into

```
map ((fn c => c vs) o (eval phi ns)) es
```

and then into

```
map (fn c => c vs)
    (map (eval phi ns) es)
```

Now the subexpression

```
(map (eval phi ns) es)
```

is purely static, and can be lifted out.

A drawback of the above solution is that an intermediate list of pre-computed functions has to be generated. Then this list will be repeatedly interpreted by the outer call to `map`. Note that the length of the intermediate list is statically determined by the list `es`, but the outer `map` makes no use of that fact.

5.2 Specialized general-purpose functionals

It seems that the weakness of the "free-theorem" approach is that the solution has to be expressed in terms of the functionals that are already present in the program being transformed. But, as shown by Holst and Gomard [HG91], it is possible to eat the cake and have it too: namely, to let functionals express recursion in the transformed program without introducing intermediate data structures. This can be achieved by introducing transformed versions of functionals.

Let's return to the expression

```
map d (map s xs)
```

The difficulty here is that we can't combine two occurrences of `map` into a single static expression. To achieve that, we need to swap the arguments of the outer `map`. So, let's introduce a new function

```
fun map' xs f = map f xs
```

Now we can rewrite `map d (map s xs)` as `map' (map s xs) d`, and the subexpression `map' (map s xs)` becomes purely static! Unfortunately, our joy is somewhat premature, because `map'` as defined above will not do anything, before it has been given both arguments. We can however develop a better definition for `map'`, that will start to work as soon as it is given only the first argument.

First, let's write down an explicit recursive definition of `map'`:

```
fun map' xs f = []
  | map' (x :: xs) f =
      f x :: map' xs f
```

Now we can apply the standard techniques described in Section 4: `f` should be rearranged to the right hand sides, and the static subexpressions lifted. The result is

```
fun map' xs = (fn f => [])
  | map' (x :: xs) =
      let val c = map' xs
      in fn f => f x :: c f end
```

Now the expression

```
map (fn c => c vs)
    (map (eval phi ns) es)
```

can be transformed into

```
map' (map (eval phi ns) es)
     (fn c => c vs)
```

where `map' (map (eval phi ns) es)` is static.

This solution is not perfect, however: the intermediate list of functions will still be generated by the inner `map` and immediately consumed by the outer `map'`[3].

This is due to the generality of `map'`, which is excessive in our particular case. After all, our goal was to separate binding times in `map (d o s) xs`, and the decision to reduce this expression to `map d (map s xs)` seems to be justified by nothing, except for our "insight" and voluntarism.

A more straightforward approach is to introduce a specialized functional

```
fun map_dos s xs d = map (d o s) xs
```

whose direct definition is

```
fun map_dos s [] d = []
  | map_dos s (x :: xs) d =
      d (s x) :: map_dos s xs d
```

which, upon separating binding times, takes the form

```
fun map_dos s [] = (fn d => [])
  | map_dos s (x :: xs) =
      let val x1 = s x
          val x2 = map_dos s xs
      in fn d => d x1 :: x2 d end
```

Now the expression

```
map ((fn c => c vs) o (eval phi ns)) es
```

can be rewritten as

```
map_dos (eval phi ns) es (fn c => c vs)
```

A minor deficiency of that definition is that a strange auxiliary function `fn c => c vs` has had to be introduced. This can be rectified, if we return to the initial expression

```
map (fn e => eval phi ns e vs) es
```

which is a special case of

[3] Unlike the previous solution, this list will be consumed only once, at "compile-time", rather than each time the value of `vs` is supplied.

```
map (fn x => s x d) xs
```

Thus let's introduce the functional

```
fun map_sxd s xs d =
  map (fn x => s x d) xs
```

Defining it in terms of explicit recursion

```
fun map_sxd s [] d = []
  | map_sxd s (x :: xs) d =
      s x d :: map_sxd s xs d
```

and separating binding times, we obtain

```
fun map_sxd s [] = (fn d => [])
  | map_sxd s (x :: xs) =
      let val c1 = s x
          val c2 = map_sxd s xs
      in fn d => c1 d :: c2 d end
```

Now the expression

```
map (fn e => eval phi ns e vs) es
```

can be rewritten as

```
map_sxd (eval phi ns) es vs
```

5.3 Static values under dynamic control

Consider the expression

```
fn s => fn d => s (if d then 1 else 2)
```

in which the test in the conditional is dynamic, whereas both its branches are static. Hence, the choice between the two branches cannot be made until the value of `d` becomes known, for which reason the application of `s` gets delayed too.

Nevertheless, if we push `s` into the conditional

```
fn s => fn d => if d then s 1 else s 2
```

the applications of `s` become static, so that they can be lifted out of `fn d =>`:

```
fn s =>
  let val x1 = s 1 and x2 = s 2
  in fn d => if d then x1 else x2 end
```

This works fine, if a static function `s` is immediately applied to a dynamic conditional, but `s` may be applied to a function call `s (f d)`, where the body of the definition of `f` is known to contain dynamic conditionals with static branches. In this case we need a trick to propagate the application of `s` to the static values.

This trick may consist in rewriting the function `f` in *continuation-passing style*, or CPS [HG91]. Namely, `f` is replaced with `f'`, its version in CPS, such that `s(f d) = f' s d`.

For example, let's consider the function `lookup` in Figure 4, and the expression

```
s (lookup kvs d)
```

where `s` and `kvs` are static, and `d` dynamic. Since the definition of `lookup` contains a conditional with a dynamic test

```
if k=n then v else lookup rest n
```

the result of the function is dynamic too. However, if we rewrite the definition of `lookup` in CPS

```
fun lookup' c [] n =
      c (raise Fail "lookup")
  | lookup' c ((k,v) :: rest) n =
      if k=n then c v
            else lookup' c rest n
```

and separate binding times

```
fun lookup' c [] =
      fn n => c (raise Fail "lookup")
  | lookup' c ((k,v) :: rest) =
      let val x1 = c v
          val x2 = lookup' c rest
      in
        fn n =>
          if k=n then x1 else x2 n
      end
```

the expression `s (lookup kvs d)` can be rewritten as `lookup' s kvs d`, where the subexpression `lookup' s kvs` is purely static.

6 Conclusions

If we write language definitions in a first-order language, we badly need a partial evaluator in order to remove the overhead introduced by the interpretation. But, if our language provides functions as first-class values, an interpreter can be relatively easily rewritten in such a way that it becomes more similar to a compiler, rather than to an interpreter.

The language in which the interpreters are written need not be a lazy one, but, if the language is strict, some attention should be paid by the programmer to preserving termination properties of the program being transformed.

References

[Hol89] Carsten Kehler Holst. Syntactic currying: yet another approach to partial evaluation. Student report 89-7-6, DIKU, University of Copenhagen, Denmark, July 1989.

[Hol90] Carsten Kehler Holst. Improving full laziness. In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional programming*, Ullapool, Scotland, 1990, Springer-Verlag.

[HH90] Carsten Kehler Holst and John Hughes. Towards improving binding times for free! In Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional programming*, Ullapool, Scotland, 1990, Springer-Verlag.

[HG91] Carsten Kehler Holst and Carsten Krogh Gomard. Partial evaluation is fuller laziness. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no.9, September 1991)*, pages 223–233, ACM, 1991.

[Wad89] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architectures, pages 347–359*, London, September 1989. ACM.