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

Sergei A. Romanenko
sergei.romanenko@supercompilers.ru

Keldysh Institute of Applied Mathematics
Russian Academy of Sciences

Outline

1. Defining a language by an interpreter
   - Interpreters and partial evaluation
   - An example interpreter
   - Representing recursion by cyclic data structures
Outline

1. Defining a language by an interpreter
   - Interpreters and partial evaluation
   - An example interpreter
   - Representing recursion by cyclic data structures

2. Separating binding times
   - What is “binding time”
   - Lifting static subexpressions
   - Liberating control
   - Separating binding times in the interpreter
   - Functionals and the separation of binding times

3. Conclusions
Outline

1. Defining a language by an interpreter
   - Interpreters and partial evaluation
   - An example interpreter
   - Representing recursion by cyclic data structures

2. Separating binding times
   - What is “binding time”
   - Lifting static subexpressions
   - Liberating control
   - Separating binding times in the interpreter
   - Functionals and the separation of binding times

3. Conclusions
Outline

1. Defining a language by an interpreter
   - Interpreters and partial evaluation
   - An example interpreter
   - Representing recursion by cyclic data structures

2. Separating binding times
   - What is “binding time”
   - Lifting static subexpressions
   - Liberating control
   - Separating binding times in the interpreter
   - Functionals and the separation of binding times

3. Conclusions
Suppose, our program is written in Standard ML (a strict functional language). Let us define an “interpreter”, a function `run`, whose type is

\[
\text{val run : prog * input -> result}
\]

Then, somewhere is the program we can write a call

\[
... \text{run (prog, d)} ... 
\]

where

- `run` – an interpreter.
- `prog` – a program in the language implemented by `run`.
- `d` – input data.
Removing the overhead due to interpretation

**Problem**
A naïve interpreter written in a straightforward way is likely to introduce a considerable overhead.

**Solution**
Refactoring = rewriting = “currying” the interpreter.

```ocaml
val run : prog * input -> result
... run (prog, input) ...
```

can be replaced with

```ocaml
val run : prog -> input -> result
... (run prog) input ...
```
1st Futamura projection in the 1st-order world

1st-order world

- A program $p$ is a text, which cannot be applied to an input $d$ directly.
- We need an explicit function $L$ defining the “meaning” of $p$, so that $Lp$ is a function and $Lp d$ is the result of applying $p$ to $d$.

Definition

A specializer is a program $spec$, such that

$L p (s, d) = L (L spec (p, s)) d$

The 1st Futamura projection

$L run (prog, input) = L (L spec(run, prog)) input$
1st Futamura projection in the higher-order world

**Higher-order world**
- We can pretend that a program \( p \) is a function, so that \( p \, d \) is the result of applying \( p \) to \( d \).

**Definition**
- A specializer is a program \( \text{spec} \), such that
  \[
  p \,(s, d) = \text{spec} \,(p, s) \, d
  \]

**The 1st Futamura projection**
- \[
  \text{run} \,(\text{prog}, \text{input}) = \text{spec} \,(\text{run}, \text{prog}) \, \text{input}
  \]

**The 2nd Futamura projection**
- \[
  \text{run} \,(\text{prog}, \text{input}) = \text{spec} \,(\text{spec}, \text{run}) \, \text{prog} \, \text{input}
  \]
Refactoring \textit{run} to \textit{spec(spec, run)} by hand

\textbf{Observation}

\textit{spec(spec, run)} takes as input a program \textit{prog} and returns a function that can be applied to some input data \textit{input}.

\textbf{An idea}

Let try to manually refactor a naïve, straightforward interpreter \textit{run} to a “compiler”, equivalent to \textit{spec(spec, run)}.

\textbf{The sources of inspiration}

A few old papers (1989–1991) about “fuller laziness” and “free theorems”.

\textbf{What is different}

We shall apply the ideas developed for lazy languages to a strict language.
References – “Fuller laziness”

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


References - “Free theorems”


Let us consider an interpreter defined in Standard ML as a function
\( \text{run} \) having type

\[
\text{val run : prog} \to \text{int list} \to \text{int}
\]

We suppose that

- A program \( \text{prog} \) is a list of mutually recursive first-order
  function definitions.
- A function in \( \text{prog} \) accepts a fixed number of integer
  arguments.
- A function in \( \text{prog} \) returns an integer.
- The program execution starts with calling the first function in
  \( \text{prog} \).
Abstract syntax of programs

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;
The factorial function

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

when written in abstract syntax, takes the form

```plaintext
val fact_prog = [
  ("fact", (["x"],
              IF(
                BIN("=" , VAR "x" , INT 0),
                INT 1,
                BIN("*" ,
                     VAR "x" ,
                     CALL("fact" ,
                          [BIN("-" , VAR "x" , INT 1)])))
)) ];
```
First-order interpreter – General structure

fun eval prog ns exp vs =
  case exp of
    | INT i => ...
    | VAR n => ...
    | BIN(name, e1, e2) => ...
    | IF(e0, e1, e2) => ...
    | CALL(fname, es) => ...

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
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) => ...
fun eval prog ns exp vs =
    case exp of
    | INT i => ...
    | VAR n => ...
    | BIN(name, e1, e2) => ...
    | IF(e0, e1, e2) => ...
    | CALL(fname, es) =>
        let
            val (ns0, body0) = lookup prog fname
            val vs0 = evalArgs prog ns es vs
        in eval prog ns0 body0 vs0 end
Formally, the present version of `run` is “curried”, i.e. the evaluation of `run prog` returns a function. But, in reality, the evaluation starts only when `run` is given 2 arguments:

```
run prog vals
```

### A problem

For the most part, `eval` recursively descends from the current expression to its subexpressions. But, when evaluating a function call, it replaces the current expression with a new one, taken from the whole program `prog`. Thus, if we tried to evaluate `eval` with respect to `exp`, this might result in an infinite unfolding!
"Denotational" approach: a cyclic function environment

Refactoring: replacing `prog` with a function environment `phi`

```
eval prog ns exp vs → eval phi ns exp vs
```

`phi` should map function names to their "meanings", i.e. functions.

A problem

- Recursive calls in `prog` lead to a cyclic functional environment `phi`.
- Standard ML is a strict language, for which reason we cannot directly represent `phi` as an infinite tree.

A solution

Standard ML allows us to use "imperative features": locations, references and destructive updating.
Imperative features of Standard ML

- `ref v` creates a new location, initializes it with `v`, and returns a reference to the new location.
- `! r` returns the contents of the location referenced to by `r`. The contents of the location remains unchanged.
- `r := v` replaces the contents of the location referenced by `r` with a new value `v`.

An idea

- `phi fname` should return a reference to the “meaning” of the function `fname`.
- We can easily create `phi fname` with locations initialized with dummy values and update the locations with correct values at a later time.
fun eval phi ns exp vs =
  case exp of
  | INT i => ... |
  | VAR n => ... |
  | BIN(name, e1, e2) => ... |
  | IF(e0, e1, e2) => ... |
  | 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
Initializing \( \phi \)

\[
\text{fun dummyEval (vs : int list) : int =}
\]
\[
\quad \text{raise Fail "dummyEval"}
\]

\[
\text{fun app f [] = ()}
\]
\[
\quad | \text{app f (x :: xs) = (f x : unit; app f xs)}
\]

\[
\text{fun run (prog : prog) =}
\]
\[
\quad \text{let val phi = map (fn (n,_) => (n,ref dummyEval)) prog}
\]
\[
\quad \quad \text{val (_, r0) = hd phi}
\]
\[
\quad \text{in app (fn (n, (ns, e)) =>}
\]
\[
\quad \quad \quad (\text{lookup phi n) := eval phi ns e)}
\]
\[
\quad \quad \text{prog;}
\]
\[
\quad \text{!r0}
\]
\[
\text{end}
\]
Outline

1. Defining a language by an interpreter
   - Interpreters and partial evaluation
   - An example interpreter
   - Representing recursion by cyclic data structures

2. Separating binding times
   - What is “binding time”
   - Lifting static subexpressions
   - Liberating control
   - Separating binding times in the interpreter
   - Functionals and the separation of binding times

3. Conclusions
In an expression like

\[(\text{fn } x \Rightarrow \text{fn } y \Rightarrow \text{fn } z \Rightarrow e)\]

- \(x\) is bound \textit{before} \(y\), \(y\) is bound \textit{before} \(z\).
- The variables that are bound first are called \textit{early}, and the ones that are bound later are called \textit{late} (Holst, 1990).
- The early variables are said to be more \textit{static} than the late ones, whereas the late variables are said to be more \textit{dynamic} than the earlier ones.
Consider the declarations

\[
\text{val } h = \text{fn } x \Rightarrow \\
\quad \text{fn } y \Rightarrow \\
\quad \quad \sin x \times \cos y \\
\text{val } h' = h \ 0.1 \\
\text{val } v = h' \ 1.0 + h' \ 2.0
\]

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 by “lifting” \( \sin x \) in the following way:

\[
\text{val } h = \text{fn } x => \\
\quad \text{let val } \sin_x = \sin x \\
\quad \text{in fn } y => \sin_x * \cos y \text{ end}
\]

The transformation of that kind, when applied to a program in a lazy language, is known as transforming the program to a “fully lazy form” (Holst 1990).
Lifting may be unsafe

A danger

In the case of a strict language, the lifting of subexpressions may change termination properties of the program!

For example, if \texttt{monster} is a function that never terminates, then evaluating
\begin{verbatim}
val h = fn x => fn y => monster x * cos y
val h' = h 0.1
\end{verbatim}
terminates, while the evaluation of
\begin{verbatim}
val h = fn x =>
    let val monster_x = monster x
    in fn y => monster_x * cos y end
val h' = h 0.1
\end{verbatim}
does not terminate.
Lifting a condition

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

By lifting \( (p \ x) \) we get

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

The result is not as good as we’d like

- Lifting the condition \( (p \ x) \) does not remove the conditional.
- We still cannot lift \( (f \ x) \) and \( (g \ x) \), because this would result in unnecessary computation.
Let us return to the expression

\[
\text{fn } x \Rightarrow \\
\text{fn } y \Rightarrow \text{if } (p \ x) \text{ then } (f \ x \ y) \text{ else } (g \ x \ y)
\]

Instead of lifting the test \((p \ x)\), we can push \(\text{fn } y \Rightarrow\) over \(\text{if } (p \ x)\) into the branches of the conditional!

\[
\text{fn } x \Rightarrow \\
\text{if } (p \ x) \text{ then} \\
\text{fn } y \Rightarrow (f \ x \ y) \\
\text{else} \\
\text{fn } y \Rightarrow (g \ x \ y)
\]
Safely lifting static subexpression inside each branch

Finally, \((f \, x)\) and \((g \, x)\) can be lifted, because this will not necessary lead to unnecessary computation.

\[
\text{fn } x \Rightarrow \\
\text{ if (p \, x) then } \\
\text{ let val f\_x = (f \, x) } \\
\text{ in (fn y => f\_x \, y) end } \\
\text{ else } \\
\text{ let val g\_x = (g \, x) } \\
\text{ in (fn y => g\_x \, y) end }
\]

A subtlety

Evaluating \((f \, x)\) or \((g \, x)\) may be still useless, if the function returned by the expression is never called.
Pushing \( \text{fn } y => \) into branches of a case

\( \text{fn } y => \) can also be pushed into other control constructs, containing conditional branches. For example,

\[
\text{fn } x => \\
\text{fn } y => \\
\text{case } f \ x \ \text{of} \\
\text{A} => g \ x \ y \\
\text{B} => h \ x \ y
\]

can be rewritten as

\[
\text{fn } x => \\
\text{case } f \ x \ \text{of} \\
\text{A} => \text{fn } y => g \ x \ y \\
\text{B} => \text{fn } y => h \ x \ y
\]
The function `run` is good enough already, and need not be revised. So let us consider the definition of the function

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

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

```haskell
fun eval phi ns exp =
  fn vs =>
    case exp of
      INT i => i
  ...
```
Now we can push \( \text{fn vs =>} \) into the \text{case} construct:

\[
\text{fun eval phi ns exp = case exp of} \\
\text{INT i => (fn vs => i)} \\
\ldots
\]

so that the right hand side of each match rule begins with \( \text{fn vs =>} \), and can be transformed further, independently from the other right hand sides.
Refactoring eval: final result for INT, VAR, BIN

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) => ...
    | CALL(fname, es) => ...

and evalArgs phi ns [] = ...
Refactoring eval: final result for IF

fun eval phi ns exp =
  case exp of
    INT i => ...
  | VAR n => ...
  | BIN(name, e1, e2) => ...
  | 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) => ...

and evalArgs phi ns [] = ...
Refactoring `eval`: final result for `CALL`

```haskell
fun eval phi ns exp =
  case exp of
    INT i => ...
  | VAR n => ...
  | BIN(name, e1, e2) => ...
  | IF(e0, e1, e2) => ...
  | 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 [] = ...
```
Refactoring `eval`: final result for `getVal'` and `evalArgs`

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

fun eval phi ns exp = ... 

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
```
We do not know how to lift static subexpressions appearing in the arguments of higher-order functions:

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

A straightforward solution consists in replacing functionals with explicit recursion:

```
  and evalArgs phi ns [] vs = []
  | evalArgs phi ns (e :: es) vs =
      eval phi ns e vs ::
      evalArgs phi ns es vs
```
Functionals and the separation of binding times

Separating binding times without removing functionals

A suggestion by Holst and Hughes (1990)

Binding times can be separated by applying commutative-like laws, which can be derived from the types of polymorphic functions using the “free-theorem” approach (Wadler 1989).

For example, for the function `map` a useful law is

\[
\text{map} \ (d \circ s) \ \text{xs} = \text{map} \ d \ (\text{map} \ s \ \text{xs})
\]

because, if \( s \) and \( \text{xs} \) are static subexpressions, and \( d \) a dynamic one, then `map s xs` is a static subexpression, which can be lifted.
Refactoring evalArgs without removing map

The following subexpression in the definition of \texttt{evalArgs}

\[
\text{map (fn e => eval phi ns e vs) es}
\]

can be transformed into

\[
\text{map ((fn c => c vs) o (eval phi ns)) es}
\]

and then into

\[
\text{map (fn c => c vs)}
\]
\[
\quad \text{(map (eval phi ns) es)}
\]

Now the subexpression

\[
\text{(map (eval phi ns) es)}
\]

is purely static, and can be lifted out.
Outline

1. Defining a language by an interpreter
   - Interpreters and partial evaluation
   - An example interpreter
   - Representing recursion by cyclic data structures

2. Separating binding times
   - What is “binding time”
   - Lifting static subexpressions
   - Liberating control
   - Separating binding times in the interpreter
   - Functionals and the separation of binding times

3. 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.

If the 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.