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

Meta 2008 – July 3, 2008

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

Outline

Outline

“Extending” a language by means of an interpreter

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

```
val run : prog * input -> result
```

Then, somewhere is the program we can write a call

```
... 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.

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

can be replaced with

```
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 $L\ p$ is a function and $L\ p\ d$ 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 $spec$, such that
$p\ (s, d) = spec\ (p, s)\ d$

The 1st Futamura projection

$run\ (prog, input) = spec(run, prog)\ input$

The 2nd Futamura projection

$run\ (prog, input) = spec(spec, run)\ prog\ input$

Refactoring *run* to *spec*(*spec*, *run*) by hand

Observation

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

An idea

Let try to manually refactor a naïve, straightforward interpreter *run* to a "compiler", equivalent to *spec*(*spec*, *run*).

The sources of inspiration

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

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.
- 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.
- 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.

References - "Free theorems"

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

An interpreter as a function in Standard ML

Let us consider an interpreter defined in Standard ML as a function `run` having type

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

We suppose that

- A program `prog` is a list of mutually recursive first-order function definitions.
- A function in `prog` accepts a fixed number of integer arguments.
- A function in `prog` returns an integer.
- The program execution starts with calling the first function in `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;
```

Example program in abstract syntax

The factorial function

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

when written in abstract syntax, takes the form

```
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
```

First-order interpreter – INT, VAR, BIN, IF

```
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) => ...
```

First-order interpreter – CALL

```
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
```

A problem

Potentially infinite recursive descent

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!

Evaluating a call

"Denotational" approach: a cyclic function environment

Refactoring: replacing prog with a function environment phi

`eval prog ns exp vs` $\longrightarrow$ `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 contentes 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.

eval using a functional environment

```
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

```
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
```

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

“Static” and “dynamic”

In an expression like

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

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

Repeated evaluation of "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`.

Avoiding repeated evaluation by lifting "static" subexpressions

This can be avoided by "lifting" `sin x` 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, 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 `monster` is a function that never terminates, then evaluating

```
val h = fn x => fn y => monster x * cos y
val h' = h 0.1
```

terminates, while the evaluation of

```
val h = fn x =>
  let val monster_x = monster x
  in fn y => monster_x * cos y end
val h' = h 0.1
```

does not terminate.

Lifting a condition

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

By lifting `(p x)` we get

```
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.

An alternative: pushing `fn y =>` into branches

Let us return to the expression

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

Instead of lifting the test `(p x)`, we can push `fn y =>` over `if (p x)` into the branches of the conditional!

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

```
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
```

A subtlety

Evaluating (f x) or (g x) may be still useless, if the function returned by the expression is never called.

Pushing `fn y =>` into branches of a `case`

`fn y =>` can also 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
```

Refactoring `eval`: moving `vs` to the right-hand side

The function `run` is good enough already, and need not be revised.
So let us consider the definition of the function

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

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

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

Refactoring eval: pushing vs to the branches

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.

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

```
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

```
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
```

Separating binding times by removing functionals

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
```

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

```
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 lifted.

Refactoring `evalArgs` without removing `map`

The following subexpression in the definition of `evalArgs`

```
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.

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.