The Mechanical Evaluation of Higher-Order Expressions

Olivier Danvy
BRICS
University of Aarhus

MFPS'98 — May 11, 1998.

The Mechanical Evaluation of Expressions (1964)

Outlined much of

the functional-programming research

done in the 80's.

Our point

To revisit

"The Mechanical Evaluation of Expressions"

with *higher-order abstract syntax*.

Higher-Order Abstract Syntax

Elegant

Higher-Order Abstract Syntax

Elegant

Powerful

Higher-Order Abstract Syntax

Elegant

Powerful

Non-trivial

The point

To model bindings

uniformly

with λ-abstractions.

Example

First-order abstract syntax:

```
LAM("x", ADD(VAR "x", LIT 1))
```

Higher-order abstract syntax:

```
LAM(fn x => ADD(x, LIT 1))
```

More generally (1/2)

```
structure Foas
= struct
    datatype exp
      = LIT of int
      | VAR of string
      | LAM of string * exp
      | APP of exp * exp
      | ADD of exp * exp
  end
```


More generally (2/2)

```
structure Hoas
= struct
    datatype exp
      = LIT of int
      | VAR of string (* free var. *)
      | LAM of exp -> exp
      | APP of exp * exp
      | ADD of exp * exp
  end
```

From first-order to higher-order

```
fun f2h e
    = f2h' e Env.empty
(* f2h : Foas.exp -> Hoas.exp *)
```

Using an environment.


```
fun f2h' (Foas.LIT i) r
    = Hoas.LIT i
  | f2h' (Foas.APP (e0, e1)) r
    = Hoas.APP (f2h' e0 r, f2h' e1 r)
  | f2h' (Foas.ADD (e1, e2)) r
    = Hoas.ADD (f2h' e1 r, f2h' e2 r)
  | ...
```

```
  | ...
  | f2h' (Foas.LAM (x, e)) r
    = Hoas.LAM
        (fn v
          => f2h' e (Env.ext (x, v, r)))
  | f2h' (Foas.VAR x) r
    = Env.lookup (x, r)
```


From higher-order to first-order

```
fun h2f e
    = ...
(* h2f : Hoas.exp -> Foas.exp *)
```

Using a generator of fresh variables.

```
fun h2f (Hoas.LIT i)
    = Foas.LIT i
  | h2f (Hoas.APP (e0, e1))
    = Foas.APP (h2f e0, h2f e1)
  | h2f (Hoas.ADD (e1, e2))
    = Foas.ADD (h2f e1, h2f e2)
  | ...
```



```
  | ...
  | h2f (Hoas.LAM f)
    = let val x = Gensym.gensym "x"
      in Foas.LAM
           (x, h2f (f (Hoas.VAR x)))
      end
  | h2f (Hoas.VAR x)
    = Foas.VAR x
```

An alternative to gensym

de Bruijn levels

```
fun h2f e
    = h2f’ e 0
(* h2f : Hoas.exp -> Foas.exp *)
```



```
fun h2f’ (Hoas.LIT i) c
    = Foas.LIT i
  | h2f’ (Hoas.APP (e0, e1)) c
    = Foas.APP (h2f’ e0 c, h2f’ e1 c)
  | h2f’ (Hoas.ADD (e1, e2)) c
    = Foas.ADD (h2f’ e1 c, h2f’ e2 c)
  | ...
```

```
| ...
| h2f'(Hoas.LAM f) c
  = let val x = "x"^(makestring c)
    in Foas.LAM
       (x,
        h2f'(f (Hoas.VAR x)) (c+1))
    end
| h2f'(Hoas.VAR x) c
  = Foas.VAR x
```


Plan

I. An evaluator
for higher-order expressions.

II. An abstract machine
for higher-order expressions.

Plan

I. An evaluator
for higher-order expressions.

II. An abstract machine
for higher-order expressions.

(using call-by-value)

Expressible values

```
datatype univ = INT of int
              | FUN of univ -> univ
```

Functionality

```
eval_f : exp -> univ Env.env -> univ
```

```
eval_h : exp -> univ
```


The first-order evaluator

```
fun eval (LIT i) r
    = INT i
  | eval (ADD (e1, e2)) r
    = (case (eval e1 r, eval e2 r) of
           (INT i1, INT i2)
             => INT (i1 + i2)
         | _ => raise TypeError)
  | ...
```

```
| ...
| eval (APP (e0, e1)) r
  = (case (eval e0 r) of
         (FUN f) => f (eval e1 r)
       | _ => raise TypeError)
| ...
```



```
| ...
| eval (LAM (x, e)) r
  = FUN
    (fn v
     => eval e (Env.ext (x, v, r)))
| eval (VAR x) r
  = Env.lookup (x, r)
```

The higher-order evaluator

```
fun eval (LIT i)
    = INT i
  | eval (ADD (e1, e2))
    = (case (eval e1, eval e2) of
           (INT i1, INT i2)
             => INT (i1 + i2)
         | _ => raise TypeError)
  | ...
```



```
  | ...
  | eval (APP (e0, e1))
    = (case (eval e0) of
           (FUN f) => f (eval e1)
         | _ => raise TypeError)
  | ...
```

```
| ...
| eval (LAM f)
  = FUN (fn v
          => eval (f ???))
```

Type mismatch: `exp` vs. `univ`.

Indeed

```
datatype exp = ...
             | LAM of exp -> exp
             | ...
```

vs.

```
datatype univ = INT of int
              | FUN of univ -> univ
```

The point

```
| eval (LAM f)  (* f : exp -> exp *)
  = FUN (fn v   (* v : univ *)
           => eval (f ???))
```


The solution

An embedding/projection
between semantics and syntax.

The embedding

```
fun u2e (INT i)
    = LIT i
  | u2e (FUN f)
    = LAM (fn e => u2e (f (e2u e)))
```


The projection

```
and h2u (LIT i)
    = INT i
  | h2u (LAM f)
    = FUN (fn v => h2u (f (u2h v)))
  | h2u _
    = raise Not_A_Value
```

Thus equipped

```
| eval (LAM f)  (* f : exp -> exp *)
  = FUN (fn v   (* v : univ *)
           => eval (f (u2e v))
```

And we are done.

Preliminary assessment

Higher-order abstract syntax can make sense

- in a typed setting; and
- independently of higher-order unification and rewriting.

Simplify, simplify, simplify.

Do we really need the embedding/projection?

Could we rather use the syntactic domain *directly* as our semantic domain?

A refinement

```
datatype exp = trivial
             | APP of exp * exp
 and trivial = INT of int
             | LAM of trivial -> exp
```

Yes, we can.

```
eval : exp -> trivial
```


Assessment

- Using higher-order abstract syntax, the evaluation is based on *substitutions*.
- This is essentially how Frank Pfenning encodes natural semantics in Elf.
- “Values are essentially syntactic in operational semantics.” (Peter Mosses, MFPS’98)

Assessment

- Using higher-order abstract syntax, the evaluation is based on *substitutions*.
- This is essentially how Frank Pfenning encodes natural semantics in Elf.
- "Values are essentially syntactic in operational semantics." (Peter Mosses, MFPS'98)

So following Peter Landin's steps has led us somewhere.

Part II: Abstract Machines

Idea: revisit Peter Landin's SECD machine with higher-order abstract syntax.

And we obtain...

And we obtain...

An SCD machine of course.

(NB: no environment.)

And through a similar development...

It works, about as pleasantly.

Conclusion

Now I believe that I understand higher-order abstract syntax better.

Conclusion

Now I believe that I understand higher-order abstract syntax better.

Thank you, Peter!

Relative efficiencies

	time
foas:	1
hoas:	1.15 with `univ`
hoas:	1.05 with `exp`