The Mechanical Evaluation of Higher-Order Expressions

Olivier Danvy
BRICS
University of Aarhus

MFPS'98 — May 11, 1998.

1

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.

3

Higher-Order Abstract Syntax

Elegant

Higher-Order Abstract Syntax

Elegant

Powerful

5

Higher-Order Abstract Syntax

Elegant

Powerful

Non-trivial

The point

7

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)

9

More generally (2/2)

From first-order to higher-order

Using an environment.

13

From higher-order to first-order

Using a generator of fresh variables.

An alternative to gensym

de Bruijn levels

```
| 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
```

19

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)

21

Expressible values

Functionality

```
eval f : exp -> univ Env.env -> univ
```

```
eval h : exp -> univ
```

23

The first-order evaluator

```
| ...
| 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

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

Type mismatch: exp vs. univ.

29

Indeed

VS.

The point

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

31

The solution

An embedding/projection between semantics and syntax.

The embedding

33

The projection

Thus equipped

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

And we are done.

35

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?

37

A refinement

Yes, we can.

eval : exp -> trivial

39

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)

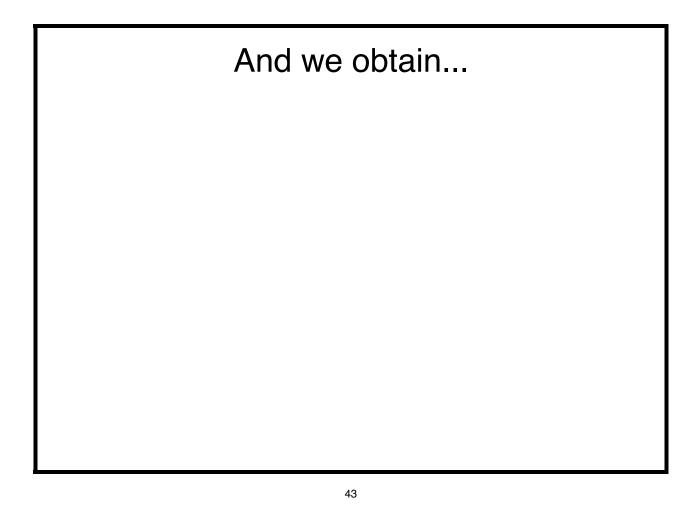
So following Peter Landin's steps has led us somewhere.



41

Part II: Abstract Machines

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



And we obtain...

An SCD machine of course.

(NB: no environment.)

And through a similar development...

It works, about as pleasantly.

45

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!

47

Relative efficiencies

time	
1	
1.15	with univ
1.05	with exp
	1 1.15