A Functional Correspondence between Normalization Functions and Abstract Machines

Olivier Danvy (danvy@brics.dk)

BRICS, University of Aarhus, Denmark

Plan

- The functional correspondence.
- The case of the untyped lambda-calculus.
- From one-step reduction to evaluation and to normalization.
- X by evaluation.
- NBE for what else?

The functional correspondence

evaluator

↓

closure conversion (to make it first order)

CPS transformation (to make it sequential)

defunctionalization (to make it first order)

abstract
machine

Source terms (de Bruijn indices)

```
structure S
= struct
    datatype term = VAR of int
                  | LAM of term
                  | APP of term * term
  end
```

Target terms (de Bruijn levels)

```
structure T
= struct
    datatype nf = LAM of nf
                | VAL of at
         and at = VAR of int
                | APP of at * nf
  end
```

Example: $\lambda x.\lambda k.k\ x$

Source term:

```
S.LAM (S.LAM (S.APP (S.VAR 0,
                     S.VAR 1)))
```

Target term:

```
T.LAM (T.LAM (T.VAL
    (T.APP (T.VAR 1,
            T.VAL (T.VAR 0)))))
```

```
datatype expval = FUN of denval -> expval
                | RES of int -> T.at
withtype denval = unit -> expval
```

```
(*  reify : expval -> int -> T.nf  *)
fun reify (RES r)
    = (fn n => T.VAL (r n))
  | reify (FUN f)
    = (fn n
       => let fun d ()
                  = RES (fn n' => T.VAR n)
          in T.LAM (reify (f d) (n+1))
          end)
```

```
(*  eval : S.term * denval Env.env -> expval  *)
fun eval (S.VAR i, e)
    = List.nth (e, i) ()
  | eval (S.LAM t, e)
    = FUN (fn a => eval (t, a :: e))
```

```
| eval (S.APP (t0, t1), e)
  = (case eval (t0, e)
       of (FUN f)
          => f (fn () => eval (t1, e))
        | (RES r)
          => RES (fn n => T.APP (r n,
                      reify (eval (t1, e)) n)))
```

```
(*  main : S.term -> T.nf  *)
fun main t
    = reify (eval (t, nil)) 0
```

The derivation

- uncurry
- closure convert expressible values
- defunctionalize residual abstract syntax
- CPS transform
- defunctionalize

```
datatype expval = CLO of S.term * expval list
                | RES of target
     and target = VAR of int
                | APP of target * expval
```

```
datatype econt
    = ECONT0
    | ECONT1 of target * econt
    | ECONT2 of S.term * expval list * econt
    | ECONT3 of int * rcont
and rcont
    = RCONT0
    | RCONT1 of T.at * tcont
    | RCONT2 of rcont
and tcont
    = TCONT0 of rcont
    | TCONT1 of expval * int * tcont
```

```
(*  reify : expval * int * rcont -> T.nf  *)
fun reify (RES r, n, k)
    = apply_target (r, n, TCONT0 k)
  | reify (CLO (t, e), n, k)
    = eval (t,
            (RES (VAR n)) :: e,
            ECONT3 (n+1, k))
```

```
(*  eval : S.term * denval Env.env * econt
           -> T.nf  *)
and eval (S.VAR i, e, k)
    = (case List.nth (e, i)
         of (RES r)
            => apply_econt (k, RES r)
          | (CLO (t', e'))
            => eval (t', e', k))
  | eval (S.LAM t, e, k)
    = apply_econt (k, CLO (t, e))
  | eval (S.APP (t0, t1), e, k)
    = eval (t0, e, ECONT2 (t1, e, k))
```

```
(*  apply_target : target * int * tcont
                       -> T.nf  *)
and apply_target (VAR n, n', k)
    = apply_tcont (k, T.VAR n)
  | apply_target (APP (r0, v1), n, k)
    = apply_target (r0, n, TCONT1 (v1, n, k))
```

```
(*  apply_tcont : T.at -> T.nf  *)
and apply_tcont (TCONT0 k, at)
    = apply_rcont (k, T.VAL at)
  | apply_tcont (TCONT1 (v1, n, k), at0)
    = reify (v1, n, RCONT1 (at0, k))
```

```
(*  apply_econt : econt * expval -> T.nf  *)
and apply_econt (ECONT0, v)
    = reify (v, 0, RCONT0)
  | apply_econt (ECONT1 (r0, k), v1)
    = apply_econt (k, RES (APP (r0, v1)))
  | apply_econt (ECONT2 (t1, e, k), CLO (t', e'))
    = eval (t', (CLO (t1, e)) :: e', k)
  | apply_econt (ECONT2 (t1, e, k), RES r0)
    = eval (t1, e, ECONT1 (r0, k))
  | apply_econt (ECONT3 (n, k), v)
    = reify (v, n, RCONT2 k)
```

```
(*  apply_rcont :  rcont * T.nf -> T.nf  *)
and apply_rcont (RCONT0, r)
    = r
  | apply_rcont (RCONT1 (at0, k), r1)
    = apply_tcont (k, T.APP (at0, r1))
  | apply_rcont (RCONT2 k, r)
    = apply_rcont (k, T.LAM r)
```

```
(*  main : S.term * (T.nf -> T.nf) -> T.nf  *)
fun main t
    = eval (t, nil, ECONT0)
```

X by evaluation

Key idea:

Reuse the evaluation mechanism

of the implementation language.

NBE for what else?

$$
\begin{aligned}
\mathcal{M} &= \langle E, \star, \varepsilon \rangle \\
E &= \text{set of elements} \\
\star &: \mathcal{M} \times \mathcal{M} \to \mathcal{M}
\end{aligned}
$$

$$
\begin{aligned}
m &\in \mathcal{M} \\
e &\in E \\
m &::= \varepsilon \mid m_1 \star m_2 \mid e
\end{aligned}
$$

Conversion rules

$$\mathfrak{m}_1 \star (\mathfrak{m}_2 \star \mathfrak{m}_3) \leftrightarrow (\mathfrak{m}_1 \star \mathfrak{m}_2) \star \mathfrak{m}_3$$

$$\mathfrak{m} \star \varepsilon \leftrightarrow \mathfrak{m}$$

$$\varepsilon \star \mathfrak{m} \leftrightarrow \mathfrak{m}$$

Notion of normal form

$$
\begin{array}{rcl}
m & \in & \mathcal{M}_{nf} \\
e & \in & E \\
m & ::= & \varepsilon_{nf} \mid e \star_{nf} m
\end{array}
$$

Reduction-based normalization

1. We orient the conversion rules

   into <u>reduction rules</u>:

$$m_1 \star (m_2 \star m_3) \;\leftarrow\; (m_1 \star m_2) \star m_3$$

$$\varepsilon \star m \;\rightarrow\; m$$

2. We apply the reduction rules

   until a normal form is obtained.

Incidentally

“Reduction rules” is a wonderful title.

Reduction-free normalization (1/2)

Notion of evaluation:

$$eval \; : \; \mathcal{M} \to (\mathcal{M}_{nf} \to \mathcal{M}_{nf})$$

$$\begin{aligned} eval(\varepsilon) &= \lambda m.m \\ eval(m_1 \star m_2) &= (eval(m_1)) \circ (eval(m_2)) \\ eval(e) &= \lambda m.e \star_{nf} m \end{aligned}$$

Reduction-free normalization (2/2)

Notion of reification:

$$\begin{array}{rcl} \mathsf{reify} & : & (\mathcal{M}_{\mathsf{nf}} \to \mathcal{M}_{\mathsf{nf}}) \to \mathcal{M}_{\mathsf{nf}} \\ \mathsf{reify}(f) & = & f(\varepsilon_{\mathsf{nf}}) \end{array}$$

Reduction-free normalization (2/2)

Notion of reification:

$$\begin{aligned} \mathsf{reify} &: (\mathcal{M}_{\mathsf{nf}} \to \mathcal{M}_{\mathsf{nf}}) \to \mathcal{M}_{\mathsf{nf}} \\ \mathsf{reify}(f) &= f(\varepsilon_{\mathsf{nf}}) \end{aligned}$$

Normalization:

$$\begin{aligned} \mathsf{normalize} &: \mathcal{M} \to \mathcal{M}_{\mathsf{nf}} \\ \mathsf{normalize}(m) &= \mathsf{reify}(\mathsf{eval}(m)) \end{aligned}$$

Pragmatic look

From a programming standpoint:

- Reduction-based: iterative flattening
  by reordering.

- Reduction-free: flatten function
  with an accumulator.

Pragmatic look

From a programming standpoint:

- Reduction-based: iterative flattening
  by reordering.

- Reduction-free: flatten function
  with an accumulator.

Also: Correctness issues.

Cool thing: NBE scales

- The λ-calculus.
- The computational λ-calculus.
- Other algebraic structures.

In short

NBE uses the expressive power

of functional programming.

In particular

For the λ-calculus and the computational λ-calculus, i.e., for proof theorists and functional programmers:

- Normalization steps involve substitutions (due to parameter passing).
- Why not reuse 30 years of compiler expertise to carry out these substitutions?

Joint work with Vincent Balat (GPCE'03)

- Setting: The typed λ-calculus.
- Issue: Type isomorphisms.
- Application: Data conversion, library search, language interoperability.

In particular

The type-theoretical counterpart of Tarski's

high school algebra problem (1950):

Can one prove all arithmetic propositions using the equations taught in high school?

In particular

The type-theoretical counterpart of Tarski's

high school algebra problem (1950):

> Can one prove all arithmetic propositions
>
> using the equations taught in high school?

(Or did they not teach you the secret ones?)

In particular

The type-theoretical counterpart of Tarski’s

high school algebra problem (1950):

Can one prove all arithmetic propositions using the equations taught in high school?

(Or did they not teach you the secret ones?)

cf. Balat, Di Cosmo, Fiore, LICS’02, POPL’04.

The problem

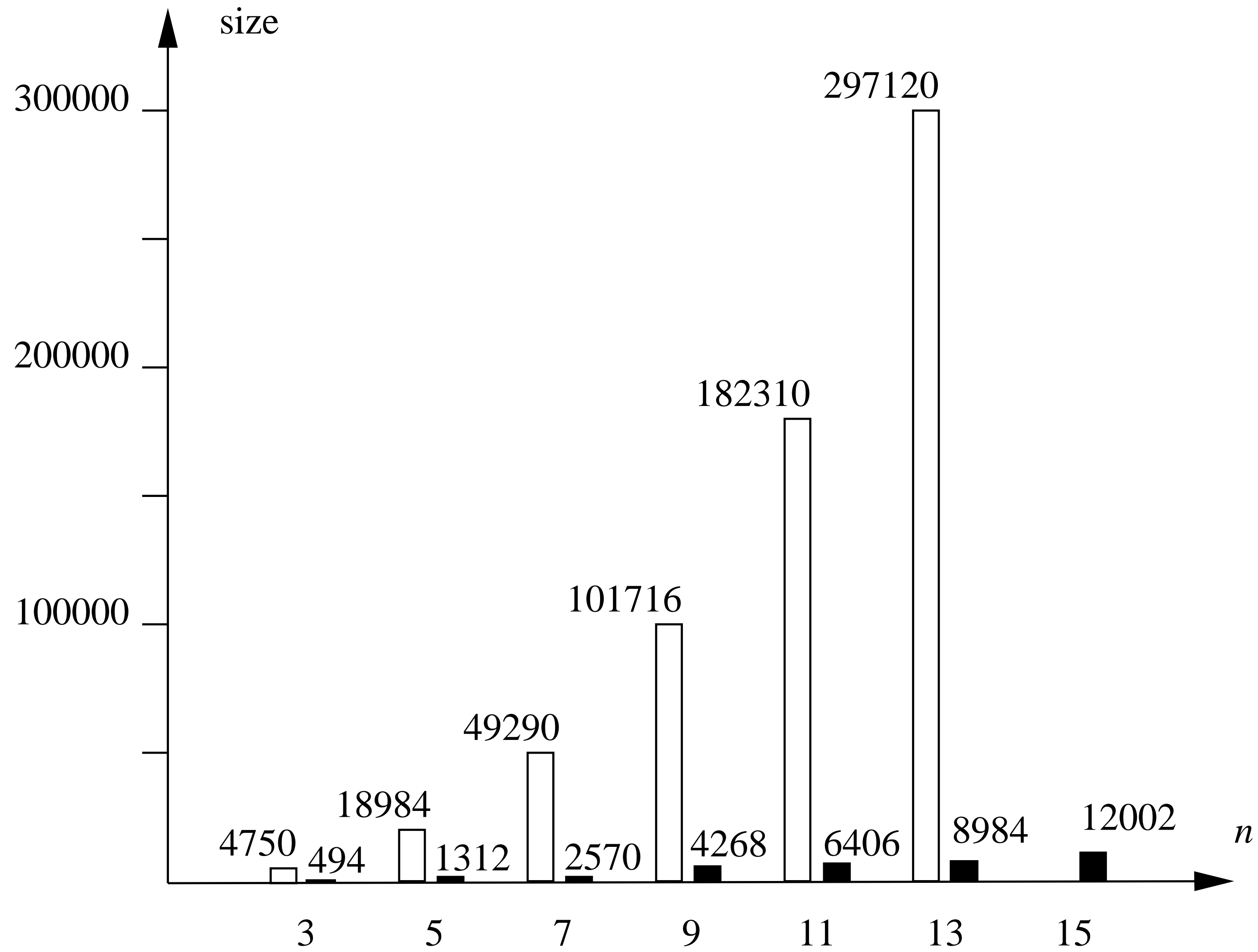
- Verify that two families of functions are inverses of each other, pointwise.
- NBE approach: Write them in ML, compose them, normalize them, and verify textual identity.
- The problem: Huge normalization time, gigantic normal forms.

The root of the problem

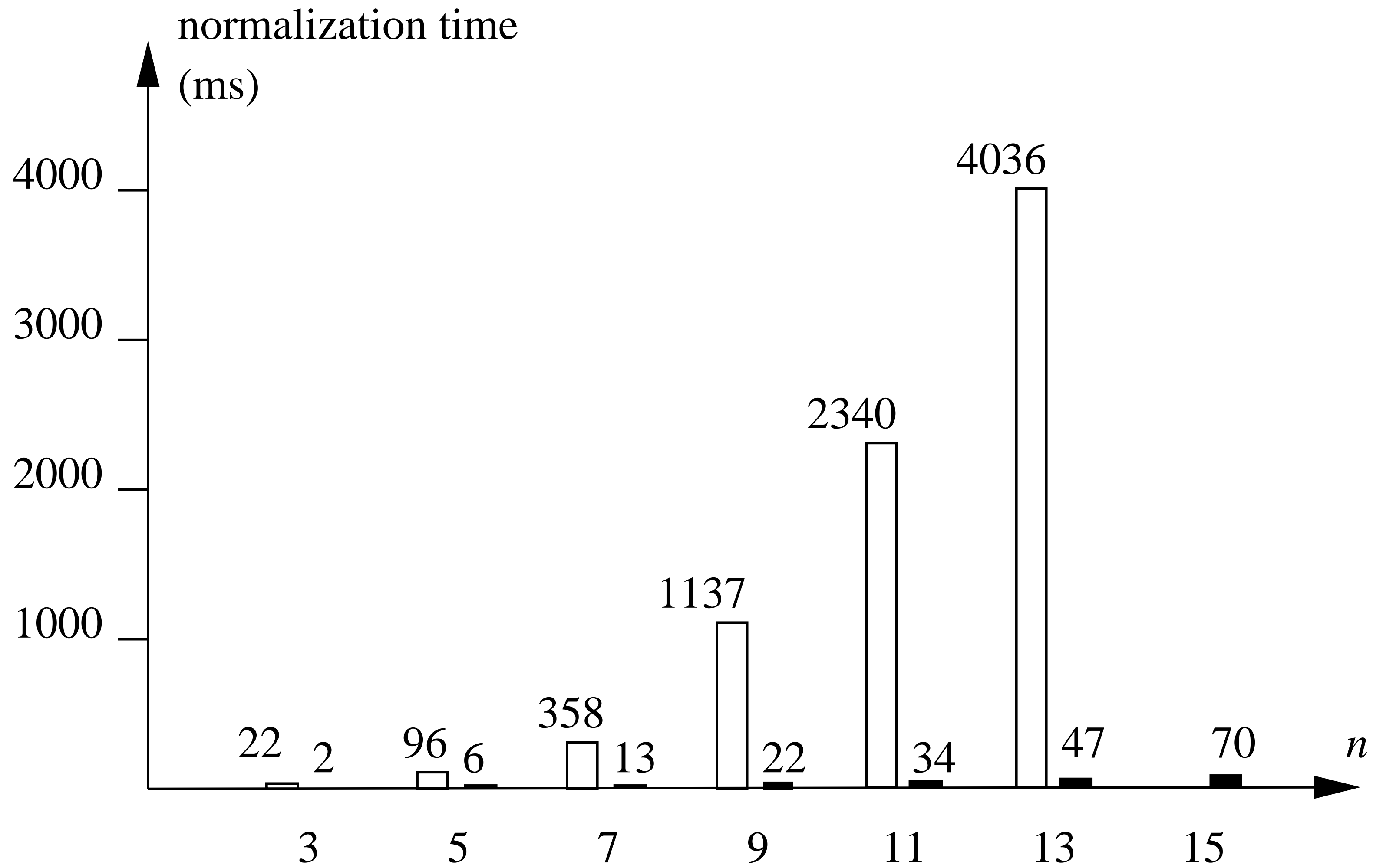
Duplication of work across conditional expressions due to use of continuations.

Our (pragmatic) solution: Dynamic memoization of intermediate results.

Results: Size of normalized programs



Results: Normalization time



Mission accomplished

NBE is now usable

for more and bigger examples

of type isomorphisms.

And furthermore

Using set/cupto instead of shift/reset,

Balat met Filinski's challenge of identifying

once and thrice (POPL'04).