On Normalisation by Evaluation

Klaus Aehlig

This Talk...

... aims at justifying the thesis

> *Normalisation by Evaluation is about the <u>computational</u> aspects of a formal system, not it's <u>semantical</u> meaning.*

- generic framework for NbE
- two examples on how to use it

... expresses my personal opinions on NbE

... contains...

... joint work with Felix Joachimski.

... work in progress with Andreas Abel and Felix Joachimski.

$\boxed{\textbf{NbE – what's it all about?}}$

- Have some calculus, say the λ-calculus, and want to compute the normal form

$\triangleright$ But want to do "better" than just rewriting

! Crucial observation $[\![(\lambda x.t)s]\!] = [\![t[s/x]]\!]$

…in general: evaluation equates terms with the same denotation

$\therefore$ Could as well imagine that we already evaluated the normal form

$\leadsto$ If we have a way to get back from the semantics…
…we could normalise by just evaluating
especially if our semantics has some operational meaning (programming languages!)

- Benefits: more elegant/modular; no wrongly implemented substitutions, …

General Setting: Syntax and Semantics

- Syntax.
  - Terms
  - Equality relation $\sim$
    (usually induced from some rewrite relation $\rightarrow$)
  - Canonical forms
- Semantics.
  (whatsoever!)
- Back & Forth.
  - Evaluation $[\![\cdot]\!]_\eta$: Syntax $\rightarrow$ Semantics
  - Distinguished valuation η^* : Vars $\rightarrow$ Semantics
  - "Inverse" $\downarrow$: Semantics $\rightarrow$ Syntax

Normalisation by Evaluation

- Back & Forth.
  - Evaluation $[\![\cdot]\!]_\eta : \text{Syntax} \to \text{Semantics}$
  - Distinguished valuation $\eta^* : \text{Vars} \to \text{Semantics}$
  - "Inverse" $\downarrow$: Semantics $\to$ Syntax

- Properties.
  - Soundness: $t \sim t' \Rightarrow [\![t]\!]_\eta = [\![t']\!]_\eta$
  - Reconstruction: for canonical terms $\downarrow [\![t]\!]_{\eta^*} = t$

$\leadsto$ For a term t that has a canonical from $t' \sim t$ we obtain

$$\downarrow [\![t]\!]_{\eta^*} = \downarrow [\![t']\!]_{\eta^*} = t'$$

Contents of the Semantics

- Need some syntactical entities in it – at least for the variables

$$x = \downarrow [\![x]\!]_{\eta^*} = \downarrow \eta^*(x)$$

shows that η^* actually is an embedding.

... but if variables stand for themselves what then is $\eta^*(x)a$?
I.e., how does $\eta^*(x)$ behave as a function?

$\rightsquigarrow$ generalise to neutral terms!

$$\uparrow : \text{Neutr} \to \text{Semantics}$$

... and define

$$\uparrow (x\overrightarrow{t})S = \uparrow (x\overrightarrow{t} \downarrow S)$$

Example: The Untyped λ-Calculus

- Consider the untyped λ-calculus. What more do we need than the embedding of the neutral terms?

$\rightsquigarrow$ In λ-calculus everything is a function

$$D \cong D \to D$$

- We already have to have Neutr
  – why not just add those two things up?

$$D \cong (\mathrm{Neutr} + (D \to D))_\perp$$

  call the other embedding $\underline{\mathrm{abs}}\,(\cdot)$

- Or in `haskell`

```
data LAM = Up TERM | ABS (LAM -> LAM)
```

... use 2-level λ-calculus to analyse.

$\boxed{\textbf{Does this suffice? – First Step}}$

$$D \cong (\mathrm{Neutr} + (D \to D))_\perp$$

(1) Have to provide a semantics for the term calculus

$\rightsquigarrow$ λ is easy: just produce a function:

$$[\![\lambda x.t]\!]_\eta = \underline{\mathrm{abs}}\left(a \mapsto [\![t]\!]_{\eta[a/x]}\right)$$

$\rightsquigarrow$ application is also easy: for Neutr we already had

$$\uparrow (x\,\overrightarrow{t}\,)S = \uparrow (x\,\overrightarrow{t}\,\downarrow S)$$

... and for functions it's obvious

$$\underline{\mathrm{abs}}\,(f)\,a = f(a)$$

This yields the soundness property.

"meta-β holds for meta-functions"

$\boxed{\textbf{Second Step}}$

$$D \cong (\mathrm{Neutr} + (D \to D))_\perp$$

(2) Have to define the "inverse" $\downarrow$

$\leadsto$ For Neutr it's easy: they are already syntactical and normal

$$\downarrow\uparrow t = t$$

$\leadsto$ For functions do the usual trick...

$$\downarrow (\underline{\mathrm{abs}}\,(f)) = \lambda x \downarrow (f(\uparrow x))$$

...and see what it does to a new variable.

This yields the reconstruction property.

(Induction on the buildup of normal terms.)

$\boxed{\textbf{Example}}$

Recall that $S = \lambda xyz.xz(yz)$ and $K = \lambda yx.x$.

$$
\begin{array}{rcl}
\downarrow [\![SKK]\!] & = & \downarrow ((\underline{\mathrm{abs}}\,(x \mapsto \underline{\mathrm{abs}}\,(y \mapsto \underline{\mathrm{abs}}\,(z \mapsto xz(yz)))))\,[\![K]\!][\![K]\!]) \\
& = & \downarrow (\underline{\mathrm{abs}}\,(z \mapsto [\![K]\!]z([\![K]\!]z))) \\
& = & \lambda z.\downarrow ((z \mapsto [\![K]\!]z([\![K]\!]z)) \uparrow z) \\
& = & \lambda z.\downarrow ([\![K]\!](\uparrow z)([\![K]\!](\uparrow z))) \\
& = & \lambda z.\downarrow ((\underline{\mathrm{abs}}\,(x \mapsto \underline{\mathrm{abs}}\,(y \mapsto x)))\,(\uparrow z)([\![K]\!](\uparrow z))) \\
& = & \lambda z.\downarrow (\uparrow z) = \lambda z.z
\end{array}
$$

Side Remark: Evaluation Strategy

- We have *Soundness* and *Reconstruction*

$\leadsto$ So we have achieved a normalisation algorithm

... provided meta-β-equality actually holds!

- Remember: we have diverging terms, like $(\lambda x.xx)(\lambda x.xx)$

$\leadsto$ If we equip our semantics with operational behaviour *(i.e., when implementing it!)* we must appropriately handle them

- Strategy that always finds a normal form, if one exists: standardisation, ...

$\boxed{\textbf{Untyped NbE – Summary}}$

- Provide some semantics consisting of
  - neutral terms *(to host the variables!)*
  - any case distinctions the computation might depend on

  Any such will do!

  If in doubt: term model identifying provably equal terms.
- Define the computations in this semantics.

  In our example: just application.
- Define $\downarrow$ in the natural way

  (guided by the above case distinctions)

As simple as that – let's do another example...

Sum Types — an Untyped Approach

For simplicity of presentation: just take the Booleans.

- two constants T, F
- term former $\underline{\texttt{if}}\,(c, a, b)$

... and of course the (untyped) λ-calculus

- Equalities

$$
\begin{array}{lcl}
\underline{\texttt{if}}\,(T, a, b) & = & a \\
\underline{\texttt{if}}\,(F, a, b) & = & b \\
\underline{\texttt{if}}\,(c, a, b)\, d & = & \underline{\texttt{if}}\,(c, ad, bd) \\
\underline{\texttt{if}}\,(\underline{\texttt{if}}\,(r, s, t), a, b) & = & \underline{\texttt{if}}\,(r, \underline{\texttt{if}}\,(s, a, b), \underline{\texttt{if}}\,(t, a, b))
\end{array}
$$

...and η, if you wish: $\underline{\texttt{if}}\,(\ldots, a, a) = a$ and $\underline{\texttt{if}}\,(t, T, F) = t$

...and the usual β-rules of the λ-calculus

Booleans — the Case Distinctions

Recall Equations:

$$
\begin{array}{lcl}
\underline{\mathtt{if}}\,(T, a, b) & = & a \\
\underline{\mathtt{if}}\,(F, a, b) & = & b \\
\underline{\mathtt{if}}\,(c, a, b)\, d & = & \underline{\mathtt{if}}\,(c, ad, bd) \\
\underline{\mathtt{if}}\,(\underline{\mathtt{if}}\,(r, s, t), a, b) & = & \underline{\mathtt{if}}\,(r, \underline{\mathtt{if}}\,(s, a, b), \underline{\mathtt{if}}\,(t, a, b))
\end{array}
$$

What values do we have to distinguish?

- Syntax: neutral terms
- Booleans: T, F
- $\underline{\mathtt{if}}\,(\cdot, \cdot, \cdot)$-expressions: *permutations!*
- functions

$$D \cong (\mathrm{Neutr} + 2 + (D \times D \times D) + (D \to D))_\perp$$

The implementation...

$$D \cong (\mathrm{Neutr} + 2 + (D \times D \times D) + (D \to D))_\perp$$

Disallowing terms like $\underline{\texttt{if}}\,(\lambda x.t, \ldots, \ldots)$ we could optimise to

$$D \cong (\mathrm{Neutr} + 2 + (\mathrm{Neutr} \times D \times D) + (D \to D))_\perp$$

```
data LAM = Up TERM                                  -- neutral terms
         | IF LAM LAM LAM                           -- IF neut norm norm
         | ABS (LAM -> LAM) | TRUE | FALSE -- values
```

Application is not too hard...

```
app :: LAM -> LAM -> LAM
app (ABS f) r    = f r
app (Up r) s     = Up (apply r (down s))
app (IF a b c) s = IF a (b `app` s) (c `app` s)
```

. . . more implementation. . .

Of course, all the computational power comes from the evaluation. . .

```
eval :: Term -> Valuation -> LAM
...
eval T          xi = TRUE
eval F          xi = FALSE
eval (If r s t) xi = evif (eval r xi) (eval s xi) (eval t xi)
```

. . . where the semantical "if" does the job.

```
evif :: LAM -> LAM -> LAM -> LAM
evif TRUE  a b = a
evif FALSE a b = b
evif (IF r s t) a b = IF r (evif s a b)
                           (evif t a b)
evif (Up r) a b = IF (Up r) a b
```

What about η-rules?

- Still missing: the η-rule $\underline{\texttt{if}}\,(\ldots, a, a) = a$ and $\underline{\texttt{if}}\,(t, T, F) = t$
- However η-rules are never a problem

  No new redexes generated!

$\ldots$ since we have enough permutations.

$\leadsto$ Handle η on the fly at $\downarrow(\underline{\texttt{if}}\,(\cdot, \cdot, \cdot))$

Conclusions and Further Directions

Summary.

- NbE is a generic, flexible framework
- Just add necessary case distinctions and computation rules
  Don't worry about types!

Maybe Martin-Löf Type Theory as case study?

Metamathematical remark:

This method seems to extend to arbitrary complex systems...

Q: But these systems get stronger and stronger!

A: Never mind – our system is inconsistent anyway.