Normalization by Evaluation for Finitary Typed Lambda Calculus

Tarmo Uustalu

joint work with Thorsten Altenkirch

Teooriapäevad Pedasel, 3.–5.10.2003

THINK. . .

- . . . of simply typed lambda calculus extended with a boolean type Bool (but type variables disallowed).
- The equational theory (defining $=_{\beta\eta}$) is not free of suprises: Define $\mathsf{once} = \lambda^{\mathsf{Bool}\to\mathsf{Bool}} f\ \lambda^{\mathsf{Bool}} x\ f\ x$ and $\mathsf{thrice} = \lambda^{\mathsf{Bool}\to\mathsf{Bool}} f\ \lambda^{\mathsf{Bool}} x\ f\ (f\ (f\ x))$, it holds that

$$\mathsf{once} =_{\beta\eta} \mathsf{thrice}$$

  But: try to derive it (not for the fainthearted).
- But semantically, in sets, where Bool is Bool and function types are function spaces are, this is easy! There are just 4 functions in Bool $\to$ Bool, and for all of these 4 the equality holds rather obviously.

So ... An Idea!

- Could we perhaps conclude $=_{\beta\eta}$ from equality in the set-theoretic semantics?
- Yes..., if we had completeness.
- My message of today: Yes, we have it!

How Do We Get Completeness?

- We show that *evaluation* of typed closed terms into the set-theoretic semantics is *invertible*.
- That is: We can define a function $\mathsf{quote}^\sigma \in [\![\sigma]\!]^{\mathsf{Set}} \to \mathsf{Tm}\ \sigma$ such that

$$t =_{\beta\eta} \mathsf{quote}^\sigma\ [\![t]\!]^{\mathsf{Set}}$$

  for any $t \in \mathsf{Tm}\ \sigma$.
- Consequently, for any $t, t' \in \mathsf{Tm}\ \sigma$,

$$[\![t]\!]^{\mathsf{Set}} = [\![t']\!]^{\mathsf{Set}} \Rightarrow t =_{\beta\eta} t'$$

  (completeness): and, as we obviously have soundness as well,

$$t =_{\beta\eta} t' \iff [\![t]\!]^{\mathsf{Set}} = [\![t']\!]^{\mathsf{Set}}$$

- As everything we do is constructive, quote is computable and hence we get an implementation of normalization $\mathsf{nf}^\sigma\ t = \mathsf{quote}^\sigma\ [\![t]\!]^{\mathsf{Set}}$.

Well, This Is NBE, Isn't It?

- Inverting evaluation to achieve normalization by evaluation (NBE, aka. reduction-free normalization) is not new, but:
  - we give a construction for a standard semantics rather than a nonstandard one,
  - our construction is much simpler than the usual NBE constructions,
  - we give a concrete implementation using Haskell as a poor man's metalanguage (actually one would like to use a language with dependent types).

Outline

- A recap of the calculus
- Implementation of the calculus
- Implementation of `quote`
- A demo (Yes! I can do it. . .)
- Correctness of `quote` and what it gives us
- Conclusions and future work

A recap of the calculus

- Types:

$$\mathsf{Ty} ::= \mathsf{Bool} \mid \mathsf{Ty} \to \mathsf{Ty}$$

- Typed terms:

$$\frac{x : \sigma \vdash t : \tau}{\lambda^\sigma x\ t : \sigma \to \tau} \qquad \frac{t : \sigma \to \tau \quad u : \sigma}{t\ u : \tau}$$

$$\frac{}{\mathsf{true} : \mathsf{Bool}} \qquad \frac{}{\mathsf{false} : \mathsf{Bool}} \qquad \frac{t : \mathsf{Bool} \quad u_0 : \theta \quad u_1 : \theta}{\mathsf{if}\ t\ u_0\ u_1 : \theta}$$

- $\beta\eta$-equality:

$$\begin{aligned}
(\lambda^\sigma x\ t)\ u \quad &=_\beta \quad t[x := u] \\
\lambda^\sigma x\ t\ x \quad &=_\eta \quad t \quad \text{if } x \notin \mathrm{FV}(t) \\
\textsf{if true}\ u_0\ u_1 \quad &=_\beta \quad u_0 \\
\textsf{if false}\ u_0\ u_1 \quad &=_\beta \quad u_1 \\
\textsf{if}\ t\ \textsf{true false} \quad &=_\eta \quad t \\
v\ (\textsf{if}\ t\ u_0\ u_1) \quad &=_\eta \quad \textsf{if}\ t\ (v\ u_0)\ (v\ u_1)
\end{aligned}$$

Implementing the Calculus: Syntax

- Types $\mathsf{Ty} \in \star$, typing contexts $\mathsf{Con} \in \star$ and untyped terms $\mathsf{UTm} \in \star$.

```
data Ty = Bool | Ty :-> Ty
        deriving (Show, Eq)

type Con = [ (String, Ty) ]

data UTm = Var String
        | TTrue | TFalse | If UTm UTm UTm
        | Lam Ty String UTm | App UTm UTm
        deriving (Show, Eq)
```

Cannot do typed terms $\mathsf{Tm} \in \mathsf{Con} \to \mathsf{Ty} \to \star$ (takes inductive families, not available in Haskell). But we can do...

Type Inference

- Type inference $\mathsf{infer} \in \mathsf{Con} \to \mathsf{UTm} \to \mathsf{Maybe\ Ty}$ (where $\mathsf{Maybe}\ X \cong 1 + X$):

```
infer :: Con -> UTm -> Maybe Ty
infer gamma (Var x) =
    do sigma <- lookup x gamma
       Just sigma
infer gamma TTrue  = Just Bool
infer gamma TFalse = Just Bool
infer gamma (If t u0 u1) =
    do Bool <- infer gamma t
       sigma0 <- infer gamma u0
       sigma1 <- infer gamma u1
       if sigma0 == sigma1 then Just sigma0 else Nothing
```

```
infer gamma (Lam sigma x t) =
    do tau <- infer ((x, sigma) : gamma) t
       Just (sigma :-> tau)
infer gamma (App t u) =
    do (sigma :-> tau) <- infer gamma t
       sigma' <- infer gamma u
       if sigma == sigma' then Just tau else Nothing
```

Semantics (In General)

- Type evaluation $[\![-]\!] : \mathsf{Ty} \to \star$ in a semantics is also impossible just as Tm. Workaround: coalesce all $[\![\sigma]\!]$ into one metalanguage type U of untyped semantic elements (just as all $\mathsf{Tm}_\Gamma\ \sigma$ appear coalesced in UTm).

```
class Sem u where
    true  :: u
    false :: u
    xif :: u -> u -> u -> u
    lam :: Ty -> (u -> u) -> u
    app :: u -> u -> u
```

- Untyped environments $\mathsf{UEnv}_U \in \star$:

```
type UEnv u = [ (String, u) ]
```

- (Untyped) term evaluation $[\![-]\!] \in \mathsf{UEnv}_U \to \mathsf{UTm} \to U$:

```
eval :: Sem u => UEnv u -> UTm -> u
eval rho (Var x) = d
    where (Just d) = lookup x rho
eval rho TTrue  = true
eval rho TFalse = false
eval rho (If t u0 u1) = xif (eval rho t) (eval rho u0) (eval rho u1)
eval rho (Lam sigma x t) = lam sigma (\ d -> eval ((x, d) : rho) t)
eval rho (App t u) = app (eval rho t) (eval rho u)
```

Set-Theoretic Semantics

- Untyped elements SU $\in \star$ of the set-theoretic semantics:

```
data SU = STrue | SFalse | SLam Ty (SU -> SU)

instance Eq SU where
    STrue  == STrue  = True
    SFalse == SFalse = True
    (SLam sigma f) == (SLam _ f') =
        and [f d == f' d | d <- flatten (enum sigma)]
    _ == _ = False

instance Show SU where
    show STrue  = "STrue"
    show SFalse = "SFalse"
    show (SLam sigma f) =
        "SLam " ++ (show sigma) ++ " " ++
        (show [ (d, f d) | d <- flatten (enum sigma) ])
```

- The set-theoretic semantics is a semantics:

```
instance Sem SU where
    true  = STrue
    false = SFalse
    xif STrue  d _ = d
    xif SFalse _ d = d
    lam = SLam
    app (SLam _ f) d = f d
```

Another Semantics: Free Semantics

- Typed closed terms up to $\beta\eta$ are a semantics too!

```
instance Sem UTm where
    true  = TTrue
    false = TFalse
    xif t TTrue TFalse = t
    xif t u0 u1 = if u0 == u1 then u0 else If t u0 u1
    lam sigma f = Lam sigma "x" (f (Var "x"))
    app = App
```

Note we do λ by cheating (doing it properly would take fresh name generation). But we are sure we will only one bound variable at a time, so cheating is fine!

Implementing quote: Decision Trees

- Decision trees $\mathsf{Tree} \in \mathsf{Ty} \rightarrow \star$ with leaves labelled with decisions, but branching nodes unlabelled (as the trees will be balanced and the questions along each branch in a tree the same, we prefer to keep these in a list):

```
data Tree u = Val u | Choice (Tree u) (Tree u) deriving (Show, Eq)

instance Monad Tree where
    return = Val
    (Val d) >>= h = h d
    (Choice l r) >>= h = Choice (l >>= h) (r >>= h)

instance Functor Tree where
    fmap h ds = ds >>= return . h

flatten :: Tree u -> [ u ]
flatten (Val d) = [ d ]
flatten (Choice l r) = (flatten l) ++ (flatten r)
```

enum AND questions

- Calculating the decision tree and the questions to identify an element of type: enum $\in (\sigma \in \mathsf{Ty}) \rightarrow \mathsf{Tree}\ [\![\sigma]\!]$ and questions $\in (\sigma \in \mathsf{Ty}) \rightarrow [[\![\sigma]\!] \rightarrow [\![\mathtt{Bool}]\!]]$:

```
enum :: Sem u => Ty -> Tree u
questions :: Sem u => Ty -> [ u -> u ]

enum Bool = Choice (Val true) (Val false)
questions Bool = [ \ b -> b ]
```

```
enum (sigma :-> tau) =
    fmap (lam sigma) (mkEnum (questions sigma) (enum tau))

mkEnum :: Sem u => [ u -> u ] -> Tree u -> Tree (u -> u)
mkEnum [] es = fmap (\ e -> \ d -> e) es
mkEnum (q : qs) es = (mkEnum qs es) >>= \ f1 ->
                     (mkEnum qs es) >>= \ f2 ->
                     return (\ d -> xif (q d) (f1 d) (f2 d))

questions (sigma :-> tau) =
    [ \ f -> q (app f d) | d <- flatten (enum sigma),
                           q <- questions tau ]
```

- Example of the tree and the questions for an arrow type: for `Bool` $\rightarrow$ `Bool`, these are

```
Choice
   (Choice
       (Val (lam Bool (\ d -> xif d true  true)))
       (Val (lam Bool (\ d -> xif d true  false))))
   (Choice
       (Val (lam Bool (\ d -> xif d false true )))
       (Val (lam Bool (\ d -> xif d false false))))
```

resp.

```
(\ f -> app f true :
   (\ f -> app f false :
       []))
```

quote AND nf

- Answers and a tree give a decision: $\mathsf{find}^\sigma \in [\llbracket \mathtt{Bool} \rrbracket] \to \mathsf{Tree}\,\llbracket\sigma\rrbracket \to \llbracket\sigma\rrbracket$:

```
find :: Sem u => [ u ] -> Tree u -> u
find [] (Val t) = t
find (a : as) (Choice l r) = xif a (find as l) (find as r)
```

- Inverted evaluation $\mathsf{quote}^\sigma \in \llbracket\sigma\rrbracket^{\mathsf{Set}} \to \mathsf{Tm}\ \sigma$:

```
quote :: Ty -> SU -> UTm
quote Bool STrue  = TTrue
quote Bool SFalse = TFalse
quote (sigma :-> tau) (SLam _ f) =
    lam sigma (\ t -> find [ q t | q <- questions sigma ]
                           (fmap (quote tau . f) (enum sigma)))
```

Haskell infers that we mean the enum of the set-theoretic semantics and the questions and find of the free semantics.

- Normalization $\mathsf{nf} \in (\sigma \in \mathsf{Ty}) \rightarrow \mathsf{Tm}\ \sigma \rightarrow \mathsf{Tm}\ \sigma$:

```
nf :: Ty -> UTm -> UTm
nf sigma t = quote sigma (eval [] t)
```

- A version $\mathsf{nf}' \in \mathsf{UTm} \rightarrow \mathsf{Maybe}\ ((\sigma \in \mathsf{Ty}) \times \mathsf{Tm}\ \sigma)$ exploiting type inference:

```
nf' :: UTm -> Maybe (Ty, UTm)
nf' t = do sigma <- infer [] t
           Just (sigma, nf sigma t)
```

CORRECTNESS OF quote

- **Def. (Logical Relations)** Define a family of relations $\mathsf{R}^\sigma \subseteq \mathsf{Tm}\ \sigma \times [\![\sigma]\!]^{\mathsf{Set}}$ by induction on $\sigma \in \mathsf{Ty}$:
  - if $t =_{\beta\eta}$ `True`, then $t\mathsf{R}^{\mathtt{Bool}}\mathrm{true}$;
  - if $t =_{\beta\eta}$ `False`, then $t\mathsf{R}^{\mathtt{Bool}}\mathrm{false}$;
  - if for all u, d, $u\mathsf{R}^\sigma d$ implies `App` $t\ u\mathsf{R}^\tau f\ d$, then $t\mathsf{R}^{\sigma\to\tau} f$.
- **Fund. Thm. of Logical Relations** If $\theta\mathsf{R}^\Gamma\rho$ and $t \in \mathsf{Tm}_\Gamma\ \sigma$, then $t[\theta]\ \mathsf{R}^\sigma [\![t]\!]^{\mathsf{Set}}_\rho$.

  In particular, if $t \in \mathsf{Tm}\ \sigma$, then $t\mathsf{R}^\sigma[\![t]\!]^{\mathsf{Set}}$.
- **Main Lemma** If $t\mathsf{R}^\sigma d$, then $t =_{\beta\eta} \mathsf{quote}^\sigma\ d$.

  Proof: Quite some work.
- **Main Thm.** If $t \in \mathsf{Tm}\ \sigma$, then $t =_{\beta\eta} \mathsf{quote}^\sigma\ [\![t]\!]^{\mathsf{Set}}$.

  Proof: Immediate from Fund. Thm. and Main Lemma.

What Follows?

- **Cor. (Completeness)** If $t, t' \in \mathsf{Tm}\ \sigma$, then $[\![t]\!]^{\mathsf{Set}} = [\![t']\!]^{\mathsf{Set}}$ implies $t =_{\beta\eta} t'$.

  Proof: Immediate from the Main Thm.

  Consequence from this together with soundness: $=_{\beta\eta}$ is decidable.

- **Cor.** If $t, t' \in \mathsf{Tm}\ \sigma$, then $t =_{\beta\eta} t'$ iff $\mathsf{quote}^{\sigma}\ [\![t]\!]^{\mathsf{Set}} = \mathsf{quote}^{\sigma}\ [\![t']\!]^{\mathsf{Set}}$.

  Proof: Immediate from soundness and Main Thm.

  Consequence: nf is good as a normalization function ("Church-Rosser").

- **Cor.** If $t, t' \in \mathsf{Tm}\ \sigma$ and $C\ t =_{\beta\eta} C\ t'$ for any $C : \mathsf{Tm}\ \sigma \to \mathsf{Tm}\ \mathtt{Bool}$, then $t =_{\beta\eta} t'$.

  Or, contrapositively, and more concretely, if $t, t' \in \mathsf{Tm}\ (\sigma_1 \to \ldots \to \sigma_n \to \mathtt{Bool})$ and $t \neq_{\beta\eta} t'$, then there exist $u_1 \in \mathsf{Tm}\ \sigma_1, \ldots u_n \in \mathsf{Tm}\ \sigma_n$ such that

  $$\mathsf{nf}^{\mathtt{Bool}}\ (\mathtt{App}\ (\ldots\ (\mathtt{App}\ t\ u_1)\ \ldots)\ u_n) \neq \mathsf{nf}^{\mathtt{Bool}}\ (\mathtt{App}\ (\ldots\ (\mathtt{App}\ t'\ u_1)\ \ldots)\ u_n)$$

  Proof: Can be read out from the proof of Main Thm.

- **Cor. (Maximal Consistency)** If $t, t' \in \mathsf{Tm}\ \sigma$ and $t \neq_{\beta\eta} t'$, then from the equation $t = t'$ as an additional axiom one would derive $\mathtt{True} = \mathtt{False}$.

  Proof: Immediate from the previous corollary.

Proof of Main Lemma

- The proof is by induction on σ. Case `Bool` is trivial, case $\sigma \to \tau$ is proved easily from two additional lemmata.
- **Cheap Lemma**
  1. $\mathsf{tenum}^\sigma\ (\mathsf{Tree}\ \mathsf{R}^\sigma)\ \mathsf{senum}^\sigma$.
  2. $\mathsf{tquestions}^\sigma\ [\mathsf{R}^\sigma\ \to \mathsf{R}^{\mathtt{Bool}}]\ \mathsf{squestions}^\sigma$.
- **Technical Lemma** Define a relation $< \subseteq \mathsf{UTm}\ \times [\mathsf{UTm}\ \to \mathsf{UTm}] \times \mathsf{Tree}\ \mathsf{UTm}$ by

$$t < (qs, ts) \text{ iff } t =_{\beta\eta} \mathsf{tfind}\ [q\ t \mid q \leftarrow qs]\ ts$$

If $t \in \mathsf{Tm}\ \sigma$, then $t < (\mathsf{tquestions}^\sigma, \mathsf{tenum}^\sigma)$, i.e.,

$$t =_{\beta\eta} \mathsf{tfind}\ [q\ t \mid q \leftarrow \mathsf{tquestions}^\sigma]\ \mathsf{tenum}^\sigma$$

Conclusions

- No radically new ideas, but a very nice combination.
- Inversion of evaluation into the simplest semantics—the set-theoretic one—, the program and the proof simple and elegant.
- As an extra one gets completeness of the set-theoretic semantics (a natural semantics) rather than completeness of some artificial semantics only invented to do NBE.

FUTURE WORK

- Do BDDs instead of decision trees, gives normalization into term graphs (=lambda calculus extended with `let`, or explicit substitutions).
- Extend from simply typed lambda-calculus with Bool to simply typed lambda-calculus with $0, +, 1, \times$ (intuitionistic prop. logic) or dependently typed lambda-calculus with $0, 1, \mathsf{Bool}, \Sigma$ and large elim. for Bool.
- Try also to extend the method to allow type variables (non-closed types).