

Normalization by Evaluation for $\lambda^{\rightarrow,2}$

Thorsten Altenkirch

Tarmo Uustalu

Workshop on NbE, Tallinn, 17 April 2004

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 surprises: Define `once` $= \lambda^{\text{Bool} \rightarrow \text{Bool}} f \lambda^{\text{Bool}} x f x$ and `thrice` $= \lambda^{\text{Bool} \rightarrow \text{Bool}} f \lambda^{\text{Bool}} x f (f (f x))$, it holds that

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

However: try to derive it...

- 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` \rightarrow `Bool`, and for all of these 4 the equality holds rather obviously.

SO ... AN IDEA!

- Can we perhaps conclude $=_{\beta\eta}$ from equality in the set-theoretic semantics?
- Yes..., provided we have completeness.

HOW TO GET COMPLETENESS?

- We show that *evaluation* of typed closed terms into the set-theoretic semantics is *invertible*.
- That is: We can define a function $\text{quote}^\sigma \in \llbracket \sigma \rrbracket_{\text{set}} \rightarrow \text{Tm } \sigma$ such that

$$t =_{\beta\eta} \text{quote}^\sigma \llbracket t \rrbracket_{\text{set}}$$

for any $t \in \text{Tm } \sigma$.

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

$$\llbracket t \rrbracket_{\text{set}} = \llbracket t' \rrbracket_{\text{set}} \Rightarrow t =_{\beta\eta} t'$$

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

$$t =_{\beta\eta} t' \iff \llbracket t \rrbracket_{\text{set}} = \llbracket t' \rrbracket_{\text{set}}$$

- As everything we do is constructive, `quote` is computable and hence we get an implementation of normalization $\text{nf}^\sigma t = \text{quote}^\sigma \llbracket t \rrbracket_{\text{set}}$.

THIS IS NBE...

- 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

- Implementation of the calculus and quote
- Correctness of quote and what it gives us
- Conclusions and future work

A RECAP OF THE CALCULUS

- Types:

$$\text{Ty} ::= \text{Bool} \mid \text{Ty} \rightarrow \text{Ty}$$

- Typed terms:

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

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

- $\beta\eta$ -equality:

$$(\lambda^\sigma x t) u =_\beta t[x := u]$$

$$\lambda^\sigma x t x =_\eta t \quad \text{if } x \notin \text{FV}(t)$$

$$\text{if true } u_0 u_1 =_\beta u_0$$

$$\text{if false } u_0 u_1 =_\beta u_1$$

$$\text{if } t \text{ true false} =_\eta t$$

$$v (\text{if } t u_0 u_1) =_\eta \text{if } t (v u_0) (v u_1)$$

IMPLEMENTING THE CALCULUS: SYNTAX

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

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

```
type Var = String
```

```
type Con = [ (Var, Ty) ]
```

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

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

TYPE INFERENCE

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

```
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 $\llbracket - \rrbracket : \text{Ty} \rightarrow \star$ in a semantics is also impossible to implement properly just as Tm .

Workaround: coalesce all $\llbracket \sigma \rrbracket$ into one metalanguage type U of untyped semantic elements (just as all $\text{Tm}_\Gamma \sigma$ 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:

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

- (Untyped) term evaluation:

```
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 of the set-theoretic semantics:

```
data UE1 = STrue | SFalse | SLam Ty (UE1 -> UE1)
```

```
instance Eq UE1 where
```

```
  STrue  == STrue  = True
```

```
  SFalse == SFalse = True
```

```
  (SLam sigma f) == (SLam _ f') =
```

```
    and [f d == f' d | d <- flatten (enum sigma)]
```

```
  _ == _ = False
```

- The set-theoretic semantics is a semantics:

```
instance Sem UE1 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 $\text{Tree} \in \text{Ty} \rightarrow \star$ with leaves labelled with decisions, but branching nodes unlabelled (as the trees will be perfectly balanced and the questions along each branch in a tree the same, we prefer to keep these in a separate 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 a type:
 $\text{enum} \in (\sigma \in \text{Ty}) \rightarrow \text{Tree } \llbracket \sigma \rrbracket$ and $\text{questions} \in (\sigma \in \text{Ty}) \rightarrow \llbracket \sigma \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket$:

```
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 $\text{Bool} \rightarrow \text{Bool}$, 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:

$\text{find}^\sigma \in (as \in \llbracket \text{Bool} \rrbracket) \rightarrow (ts \in \text{Tree } \llbracket \sigma \rrbracket) \rightarrow as \langle \rangle ts \rightarrow \llbracket \sigma \rrbracket:$

`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 $\text{quote}^\sigma \in \llbracket \sigma \rrbracket_{\text{set}} \rightarrow \text{Tm } \sigma:$

`quote :: Ty -> UE1 -> 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 $\text{nf} \in (\sigma \in \text{Ty}) \rightarrow \text{Tm } \sigma \rightarrow \text{Tm } \sigma$:

```
nf :: Ty -> UTm -> UTm
```

```
nf sigma t = quote sigma (eval [] t)
```

- A version $\text{nf}' \in \text{UTm} \rightarrow \text{Maybe } ((\sigma \in \text{Ty}) \times \text{Tm } \sigma)$ 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 $R^\sigma \subseteq \text{Tm } \sigma \times \llbracket \sigma \rrbracket_{\text{set}}$ by induction on $\sigma \in \text{Ty}$:
 - if $t =_{\beta\eta} \text{True}$, then $t R^{\text{Bool}} \text{true}$;
 - if $t =_{\beta\eta} \text{False}$, then $t R^{\text{Bool}} \text{false}$;
 - if for all u, d , $u R^\sigma d$ implies $\text{App } t u R^\tau f d$, then $t R^{\sigma \rightarrow \tau} f$.
- **Fund. Thm. of Logical Relations** If $\theta R^\Gamma \rho$ and $t \in \text{Tm}_\Gamma \sigma$, then $\llbracket t \rrbracket \theta R^\sigma \llbracket t \rrbracket_{\text{set}} \rho$.
In particular, if $t \in \text{Tm } \sigma$, then $t R^\sigma \llbracket t \rrbracket_{\text{set}}$.
- **Main Lemma** If $t R^\sigma d$, then $t =_{\beta\eta} \text{quote}^\sigma d$.
Proof: Quite some work.
- **Main Thm.** If $t \in \text{Tm } \sigma$, then $t =_{\beta\eta} \text{quote}^\sigma \llbracket t \rrbracket_{\text{set}}$.
Proof: Immediate from Fund. Thm. and Main Lemma.

WHAT FOLLOWS?

- **Cor. (Completeness)** If $t, t' \in \text{Tm } \sigma$, then $\llbracket t \rrbracket_{\text{set}} = \llbracket t' \rrbracket_{\text{set}}$ implies $t =_{\beta\eta} t'$.

Proof: Immediate from the Main Thm.

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

- **Cor.** If $t, t' \in \text{Tm } \sigma$, then $t =_{\beta\eta} t'$ iff $\text{quote}^\sigma \llbracket t \rrbracket_{\text{set}} = \text{quote}^\sigma \llbracket t' \rrbracket_{\text{set}}$.

Proof: Immediate from soundness and Main Thm.

Consequence: `nf` is good as a normalization function (“Church-Rosser”).

- **Cor.** If $t, t' \in \text{Tm } \sigma$ and $[C] [(x, t)] =_{\beta\eta} [C] [(x, t')]$ for any $C : \text{Tm}_{[(x, \sigma)]} \text{Bool}$, then $t =_{\beta\eta} t'$.

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

$$\text{nf}^{\text{Bool}} (\text{App } (\dots (\text{App } t \ u_1) \ \dots) \ u_n) \neq \text{nf}^{\text{Bool}} (\text{App } (\dots (\text{App } t' \ u_1) \ \dots) \ u_n)$$

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

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

Proof: Immediate from the previous corollary.

PROOF OF MAIN LEMMA

- The proof is by induction on σ . Case `Bool` is trivial, case $\sigma \rightarrow \tau$ is proved easily from two additional lemmata.
- **Cheap Lemma**
 1. $\text{enum}_{\text{syn}}^{\sigma} (\text{Tree } R^{\sigma}) \text{enum}_{\text{set}}^{\sigma}$.
 2. $\text{questions}_{\text{syn}}^{\sigma} [R^{\sigma} \rightarrow R^{\text{Bool}}] \text{questions}_{\text{set}}^{\sigma}$.
- **Technical Lemma** If $t \in \text{Tm } \sigma$, then

$$t =_{\beta\eta} \text{find}_{\text{syn}} [q \ t \mid q \leftarrow \text{questions}_{\text{syn}}^{\sigma}] \text{enum}_{\text{syn}}^{\sigma}$$

CONCLUSIONS

- No radically new ideas, but a very nice combination.
- Inversion of evaluation into the most natural model—the set-theoretic one—, the program and proof simple and elegant.
- As an consequence one gets completeness of the set-theoretic 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) (almost done!) or dependently typed lambda-calculus with $0, 1, \text{Bool}, \Sigma$ and large elim. for `Bool`.
- Try also to extend the method to allow type variables (non-closed types).