# Normalization by Evaluation for Untyped Combinatory Logic

Peter Dybjer
Chalmers tekniska högskola

Stockholm-Uppsala Logic Seminar
Stockholm, 2 February 2005

## Untyped normalization by evaluation: previous work

- Mogensen 1992: "Efficient self-interpretation in lambda calculus"
- Aehlig and Joachimski 2004: "Operational aspects of untyped normalization by evaluation"
- Filinski and Rohde 2004: "Denotational aspects of untyped normalization by evaluation"
- Devautour 2004: "Untyped normalization by evaluation" (for combinatory logic)

Related issues appear in Danvy's and Filinski's "Type-directed partial evaluation" for typed languages with general recursion.

# Formalizing typed combinatory logic in Martin-Löf type theory (AgdaLight)

Constructors for `Ty :: Set`:

```
X    :: Ty               -- base type
(=>) :: Ty -> Ty -> Ty -- function types
```

Constructors for `Exp :: Ty -> Set`:

```
K   :: (a,b :: Ty) -> Exp (a => b => a)
S   :: (a,b,c :: Ty) -> Exp ((a => b => c) => (a => b) => a => c)
App :: (a,b :: Ty) -> Exp (a => b) -> Exp a -> Exp b
```

In this way we only generate well-typed terms.

# The glueing model

```
Sem :: Ty -> Set
Sem X          = Exp X
Sem (a => b) = (Exp (a => b), (Sem a) -> (Sem b))
```

The normalization function is obtained by evaluating an expression in the glueing model, and then “reifying” this interpretation

```
nbe :: (a :: Ty) -> Exp a -> Exp a
nbe a e = reify a (eval a e)

eval  :: (a :: Ty) -> Exp a -> Sem a

reify :: (a :: Ty) -> Sem a -> Exp a
```

# Evaluation and reification

Evaluation is defined by induction on `Exp a`, eg

```
eval :: (a :: Ty) -> Exp a -> Sem a
eval (a => b => a) (K a b)
= (K a b, \x -> (App a (b => a) (K a b) (reify a x), \y -> x))
```

Reification is defined by induction on `Ty`, eg

```
reify :: (a :: Ty) -> Sem a -> Exp a
reify (a => b) (e,f) = e
```

It is tempting to "hide" the type information, but note that it is used in the computation.

# A decision procedure for convertibility

Let `e, e' :: Exp a`.

- Prove that `e conv e'` implies `eval a e = eval a e'`!
- It follows that `e conv e'` implies `nbe a e = nbe a e'`
- Prove that `e conv (nbe a e)` using the glueing (reducibility) method!
- Hence `e conv e'` iff `nbe a e = nbe a e'`
- Hence `e conv e'` iff `(nbe a e == nbe a e') = True`

## Formalizing syntax and semantics in Haskell

The Haskell type of untyped combinatory expressions:

```
data Exp = K | S | App Exp Exp
```

(We will later use $e@e'$ for `App e e'`.)

Note that Haskell types contain programs which do not terminate at all or lazily compute infinite values, such as `App K (App K (App K ... )))`.

The untyped glueing model as a Haskell type:

```
data Sem = Gl Exp (Sem -> Sem)
```

A reflexive type!

# The nbe program in Haskell

```
nbe :: Exp -> Exp
nbe e = reify (eval e)

eval :: Exp -> Sem
eval K = Gl K (\x -> Gl (App K (reify x)) (\y -> x))
eval S = Gl S (\x -> Gl (App S (reify x))
               (\y -> Gl (App (App S (reify x)) (reify y)))
                (\z -> appsem (appsem x z) (appsem (y z)))))
eval (App e e') = appsem (eval e) (eval e')

reify :: Sem -> Exp
reify (Gl e f) = e
```

# Application in the model

```
appsem :: Sem -> Sem -> Sem
appsem (Gl e f) x = f x
```

# The nbe program computes the Böhm tree of a term

**Theorem.** (Devautour 2004) `nbe` $e$ computes the combinatory Böhm tree of $e$. In particular, `nbe` $e$ computes the normal form of $e$ iff it exists.

**Proof.** Following categorical method of Pitts 1993 and Filinski and Rohde 2004 using "invariant relations".

What is the combinatory Böhm tree of an expression? An *operational* notion: the Böhm tree is defined by repeatedly applying the *inductively defined* head normal form relation.

Note that `nbe` gives a *denotational* (*computational*) definition of the Böhm tree of $e$, so the theorem is to relate an operational (inductive) and a denotational (computational) definition.

# Combinatory head normal form

Inductive definition of relation between terms in `Exp`

$$\mathtt{K} \Rightarrow^{\mathrm{h}} \mathtt{K} \qquad \mathtt{S} \Rightarrow^{\mathrm{h}} \mathtt{S}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}}{e@e' \Rightarrow^{\mathrm{h}} \mathtt{K}@e'} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}@e' \qquad e' \Rightarrow^{\mathrm{h}} v}{e@e'' \Rightarrow^{\mathrm{h}} v}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}}{e@e' \Rightarrow^{\mathrm{h}} \mathtt{S}@e'} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}@e'}{e@e'' \Rightarrow^{\mathrm{h}} (\mathtt{S}@e')@e''}$$

$$\frac{e \Rightarrow^{\mathrm{h}} (\mathtt{S}@e')@e'' \qquad (e'@e''')@(e''@e''') \Rightarrow^{\mathrm{h}} v}{e@e''' \Rightarrow^{\mathrm{h}} v}$$

# Formal neighbourhoods

To formalize the notion of combinatory Böhm tree we make use of Martin-Löf 1983 - the domain interpretation of type theory. Notions of

- formal neighbourhood = finite approximation of the canonical form of a program (lazily evaluated); in particular $\Delta$ means no information about the canonical form of a program.
- The denotation of a program is the set of all formal neighbourhoods approximating its canonical form (applied repeatedly to its parts). Two possibilities: *operational neighbourhoods* and *denotational neighbourhoods*. Different because of the *full abstraction problem*, Plotkin 1976.

# Expression neighbourhoods

An expression neighbourhood $U$ is a finite approximation of the canonical form of a program of type `Exp`. Operationally, $U$ is the set of all programs of type `Exp` which approximate the canonical form of the program. Notions of *inclusion* $\supseteq$ and *intersection* $\cap$ of neighbourhoods.

A grammar for expression neighbourhoods:

$$U ::= \Delta \mid \mathtt{K} \mid \mathtt{S} \mid U@U$$

A grammar for the sublanguage of normal form neighbourhoods:

$$U ::= \Delta \mid \mathtt{K} \mid \mathtt{K}@U \mid \mathtt{S} \mid \mathtt{S}@U \mid (\mathtt{S}@U)@U$$

# Combinatory Böhm trees

A (combinatory) Böhm tree is a *filter* of normal form neighbourhoods. A filter is a set $\alpha$ of neighbourhoods satisfying:

- $U \in \alpha$ and $U' \supseteq U$ implies $U' \in \alpha$;
- $\Delta \in \alpha$;
- $U, U' \in \alpha$ implies $U \cap U' \in \alpha$.

# Approximations of head normal forms

$$e \triangleright^{\mathrm{Bt}} \Delta$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}}{e \triangleright^{\mathrm{Bt}} \mathtt{K}} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{K}@e' \qquad e' \triangleright^{\mathrm{Bt}} U'}{e \triangleright^{\mathrm{Bt}} \mathtt{K}@U'}$$

$$\frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}}{e \triangleright^{\mathrm{Bt}} \mathtt{S}} \qquad \frac{e \Rightarrow^{\mathrm{h}} \mathtt{S}@e' \qquad e' \triangleright^{\mathrm{Bt}} U'}{e \triangleright^{\mathrm{Bt}} \mathtt{S}@U'}$$

$$\frac{e \Rightarrow^{\mathrm{h}} (\mathtt{S}@e')@e'' \qquad e' \triangleright^{\mathrm{Bt}} U' \qquad e'' \triangleright^{\mathrm{Bt}} U''}{e \triangleright^{\mathrm{Bt}} (\mathtt{S}@U')@U''}$$

## The Böhm tree of a combinatory expression

The Böhm tree of an expression $e$ in `Exp` is the set

$$\{U \mid e \triangleright^{\mathrm{Bt}} U\}$$

One can prove that it is a filter of normal form neighbourhoods, by induction on the definition of $\triangleright^{\mathrm{Bt}}$. (Note that the head normal form of an expression is unique.)

One can also prove that two convertible expressions have the same Böhm tree.

# Combinatory conversion

Conversion is inductively generated by the rules of reflexivity, symmetry, and transitivity, together with:

$$(\mathtt{K}@e)@e' \ \mathtt{conv}\ e$$

$$((\mathtt{S}@e)@e')@e'' \ \mathtt{conv}\ (e@e')@(e@e'')$$

$$\frac{e_0 \ \mathtt{conv}\ e_1 \qquad e_0' \ \mathtt{conv}\ e_1'}{e_0@e_0' \ \mathtt{conv}\ e_1@e_1'}$$

# Operational neighbourhoods of nbe

$\texttt{nbe}\,e \in U$ iff $U$ is a finite approximation of the canonical form of $\texttt{nbe}\,e$ when evaluated lazily. For example,

- $\texttt{nbe}\,e \in \Delta$, for all $e$
- $\texttt{nbe}\,\texttt{K} \in \texttt{K}$
- $\texttt{nbe}\,(\texttt{Y@K}) \in \texttt{K@}\Delta$
- $\texttt{nbe}\,(\texttt{Y@K}) \in \texttt{K@}(\texttt{K@}\Delta)$, etc

Y is a fixed point combinator.

# Definition of the operational neighbourhood relation

Is this operational semantics or denotational semantics?

The definition of the operational neighbourhood relation follows the computation rules (operational semantics) of a program. So to define the relation `nbe` $e \in U$, we must first define the relations `eval` $e \in V$ and `reify` $x \in U$. Here $V$ is a neighbourhood of the reflexive type

```
data Sem = Gl Exp (Sem -> Sem)
```

We need to consider *function neighbourhoods*.

## Function neighbourhoods

If $(U_i)_{i<n}$ and $(V_i)_{i<n}$ are families of neighbourhoods of types $\sigma$ and $\tau$, respectively, then

$$\bigcap_{i<n} [U_i; V_i]$$

is a function neighbourhood of the type $\sigma \to \tau$. We write $\Delta = \bigcap_{i<0}[U_i; V_i]$.

If $f$ is a program of type $\sigma \to \tau$, then

$$f \in \bigcap_{i<n} [U_i; V_i]$$

iff for all $i < n$, $a \in U_i$ implies $f\,a \in V_i$. In addition to inclusion and meet we consider *consistency (inhabitedness)* of function neighbourhoods.

## Neighbourhoods in Sem

- $\Delta$ is a Sem-neighbourhood.
- If $U$ is an Exp-neighbourhood and $(V_i)_{i<n}$ and $(W_i)_{i<n}$ are families of Sem-neighbourhoods, then

$$\mathtt{Gl}\, U\, (\bigcap_{i<n} [V_i; W_i])$$

is a Sem-neighbourhood.

# Operational neighbourhoods of $\texttt{eval}\, e$

$\texttt{eval}\, e \in \Delta$, as always.

For $e = \texttt{K}$ we have the equation

$$\texttt{eval}\,\texttt{K} = \texttt{Gl}\,\texttt{K}\,(\lambda x.\texttt{Gl}\,(\texttt{K@}(\texttt{reify}\,x))\,(\lambda y.x))$$

Hence,

$$\texttt{eval}\,\texttt{K} \in \texttt{Gl}\,U\,(\bigcap_i [V_i; W_i])$$

iff $\texttt{K} \in U$ and for all $i$ and for all $x \in V_i$, we have $\texttt{Gl}\,(\texttt{K@}(\texttt{reify}\,x))\,(\lambda y.x) \in W_i$. This is the case iff either $W_i = \Delta$ or $W_i = \texttt{Gl}\,U_i\,(\bigcap_j [V_{ij}; W_{ij}])$ and $\texttt{K@}(\texttt{reify}\,x) \in U_i$ and $x \in W_{ij}$ for all $j$.

# Operational neighbourhoods of $\texttt{eval}\,(e@e')$

Recursion equations

$$\begin{aligned}\texttt{eval}\,(e@e') &= \texttt{appsem}\,(\texttt{eval}\,e)\,(\texttt{eval}\,e') \\ \texttt{appsem}\,(\texttt{Gl}\,e\,f)\,x &= f\,x\end{aligned}$$

One can prove that $\texttt{eval}\,(e@e') \in W$ iff either $W = \Delta$ or there exist $U$ and $V$ such that $\texttt{eval}\,e \in \texttt{Gl}\,U\,[V;W]$ and $\texttt{eval}\,e' \in V$

# Operational neighbourhoods of nbe

Equations:

$$\begin{aligned} \texttt{nbe}\, e &= \texttt{reify}\,(\texttt{eval}\, e) \\ \texttt{reify}\,(\texttt{Gl}\, e\, f) &= e \end{aligned}$$

Thus, $\texttt{nbe}\, e \in U$ iff $U = \Delta$ or $\texttt{eval}\, e \in \texttt{Gl}\, U\, \Delta$.

## Nbe maps convertible terms into equal Böhm trees

We can prove that $\texttt{nbe}\, e \in U$ implies that $U$ is a normal form neighbourhood, and hence the denotation of $\texttt{nbe}\, e$ is a Böhm tree.

We can also prove that if $e$ $\texttt{conv}$ $e'$ and $\texttt{nbe}\, e \in U$, then $\texttt{nbe}\, e' \in U$, that is, `nbe` maps convertible terms to equal Böhm trees (cf "uniqueness of normal forms"). As in the typed case this follows by induction on the definition of convertibility, using a lemma that `eval` maps convertible terms into equal denotations.

# Completeness of nbe

Any finite part of the Böhm tree is returned:

$$e \triangleright^{\mathrm{Bt}} U \quad \text{implies} \quad \mathtt{nbe}\, e \in U$$

The proof is by induction on the derivation of $e \triangleright^{\mathrm{Bt}} U$.

Consider eg the case when $e \triangleright^{\mathrm{Bt}} \mathtt{K}$ comes from $e \Rightarrow^{\mathrm{h}} \mathtt{K}$. Since $\mathtt{nbe}\, \mathtt{K} \in \mathtt{K}$ and convertible terms have equal Böhm trees it follows that $\mathtt{nbe}\, e \in \mathtt{K}$.

# Soundness of nbe

Only approximations of the Böhm tree are returned by nbe:

$$\texttt{nbe}\, e \in U \quad \text{implies} \quad e \rhd^{\mathrm{Bt}} U$$

We need a lemma (cf reducibility/glueing method)

$$\texttt{eval}\, e \in V \quad \text{implies} \quad e \rhd^{\mathrm{Gl}} V$$

where $e \rhd^{\mathrm{Gl}} V$ iff either $V = \Delta$ or $V = \texttt{Gl}\, U\, (\bigcap_i [V_i; W_i])$ where $e \rhd^{\mathrm{Bt}} U$ and for all $i$ and $e'$, $e' \rhd^{\mathrm{Gl}} V_i$ implies $e@e' \rhd^{\mathrm{Gl}} W_i$.

This lemma is proved by induction on $e$. Soundness then follows immediately.

## Definition of $e \triangleright^{\mathrm{Gl}} U$

The property on the previous page is not directly acceptable as an inductive definition because of negative occurrence of $e \triangleright^{\mathrm{Gl}} U$.

Instead we define it as the union of an infinite sequence of approximations: $e \triangleright^{\mathrm{Gl}} V$ iff there exists an $n$ such that $e \triangleright^{\mathrm{Gl}}_n V$, where

$e \triangleright^{\mathrm{Gl}}_0 V$ iff $V = \Delta$.

$e \triangleright^{\mathrm{Gl}}_{n+1} V$ iff either $V = \Delta$ or $V = \mathtt{Gl}\, U\, (\bigcap_i [V_i; W_i])$ where $e \triangleright^{\mathrm{Bt}} U$ and for all $i$ and $e'$, $e' \triangleright^{\mathrm{Gl}}_n V_i$ implies $e@e' \triangleright^{\mathrm{Gl}}_n W_i$.

The set $\{V |\ e \triangleright^{\mathrm{Gl}} V\}$ is a filter of Sem-neighbourhoods, and is invariant under convertibility.

## Case K: $\texttt{eval}\,\texttt{K} \in V$ implies $\texttt{K} \rhd^{\mathrm{Gl}} V$

Proof by analyzing the neighbourhoods of $\texttt{eval}\,\texttt{K}$.

Case $V = \Delta$ is immediate.

Case $V = \texttt{Gl}\,U\,(\bigcap_i [V_i; W_i])$, where $\texttt{K} \in U$ and for all $i$ and $x \in V_i$, we have $\texttt{Gl}\,(\texttt{K}@(\texttt{reify}\,x))\,(\lambda y.x) \in W_i$. We need to prove two things:

- $\texttt{K} \rhd^{\mathrm{Bt}} U$. This follows from $\texttt{K} \in U$.
- For all $i$, $e' \rhd^{\mathrm{Gl}} V_i$ implies $\texttt{K}@e' \rhd^{\mathrm{Gl}} W_i$.

  Case $W_i = \Delta$, and we are done.

  Case $W_i = \texttt{Gl}\,U_i\,(\bigcap_j [V_{ij}; W_{ij}])$, where $\texttt{K}@(\texttt{reify}\,x) \in U_i$ and $x \in W_{ij}$ for all $j$. We need to show two things:

– $\texttt{K}@e' \rhd^{\mathrm{Bt}} U_i$.
Case $V_i = \Delta$. It follows that $U_i \supseteq \texttt{K}@\Delta$ and hence $\texttt{K}@e' \rhd^{\mathrm{Bt}} U_i$.
Case $V_i = \texttt{Gl}\, U'_i\, (\bigcap_j [V'_{ij}; W'_{ij}])$. It follows that $U_i \supseteq \texttt{K}@U'_i$. We know $e' \rhd^{\mathrm{Bt}} U'_i$ and hence $\texttt{K}@e' \rhd^{\mathrm{Bt}} U_i$.
– For all $j$, $e'' \rhd^{\mathrm{Gl}} V_{ij}$ implies $(\texttt{K}@e')@e'' \rhd^{\mathrm{Gl}} W_{ij}$. Because of closure of convertibility it suffices to prove $e' \rhd^{\mathrm{Gl}} W_{ij}$. But this follows from $W_{ij} \supseteq V_i$ and upward closure of $\rhd^{\mathrm{Gl}}$ in the right argument, since we know $e' \rhd^{\mathrm{Gl}} V_i$.

# Case $e@e'$:

Prove that $\texttt{eval}\,(e@e') \in W$ implies $(e@e') \rhd^{\mathrm{Gl}} W$ from the induction hypotheses that $\texttt{eval}\, e \in W$ implies $e \rhd^{\mathrm{Gl}} W$ and $\texttt{eval}\, e' \in W'$ implies $e' \rhd^{\mathrm{Gl}} W'$.

Either $W = \Delta$ and we are done.

Or there exist $U$ and $V$ such that $\texttt{eval}\, e \in \texttt{Gl}\, U\, [V; W]$ and $\texttt{eval}\, e' \in V$. We can now use the induction hypotheses to conclude that $e \rhd^{\mathrm{Gl}} \texttt{Gl}\, U\, [V; W]$ and $e' \rhd^{\mathrm{Gl}} V$. Hence it follows by the second property of $\rhd^{\mathrm{Gl}}$ that $(e@e') \rhd^{\mathrm{Gl}} W$.

# Conclusion

The proof could presumably be carried out in a similar way using denotational neighbourhoods. Can we isolate the abstract properties of function neighbourhoods which are needed for the proof?