Formal Topology and the Correctness of a Haskell Program for Untyped Normalization by Evaluation

Peter Dybjer

Chalmers University, Göteborg
(based on joint work with Denis Kuperberg, ENS Lyon)

Aarhus
24 October, 2007

Normalization and normalization by evaluation

- Computation is a step-wise procedure which is often modelled as a *binary relation* $\mathtt{red}_1$ of reduction in one step
- To prove *strong normalization* is to prove that $\mathtt{red}_1$ is well-founded and to prove *weak normalization* is to prove that all expressions can reach a normal form wrt $\mathtt{red}_1$.
- But $\mathtt{red}_1$ is not itself an *implementation* of the reduction (normalization) procedure! We must write a program to execute it! This is typically done using an abstract machine.
- *Normalization by evaluation* is a new technique for programming this procedure, using an interpretation of expressions in a special kind of model, and then extracting the normal form.
- It is also easier to *prove correctness* of the nbe algorithm than to prove normalization and confluence of reduction.

What is the purpose of normalization?

- Historically, to prove consistency and other metatheoretical properties of logical systems.
- To perform program simplification, cf type-directed partial evaluation.
- To decide convertibility (equality) of expressions, e g in a theorem prover: two expressions are convertible iff they have the same normal form. This follows from weak normalization and Church-Rosser.

To prove correctness of normalization by evaluation

Start instead with conv. An abstract normal form function is a function nbe which picks a canonical representative from each conv-equivalence class:

$$a \ \mathtt{conv} \ a' \leftrightarrow \mathtt{nbe} \ a = \mathtt{nbe} \ a'$$

This property follows from "existence" of normal forms

$$a \ \mathtt{conv} \ (\mathtt{nbe} \ a)$$

and "uniqueness" (cf confluence)

$$a \ \mathtt{conv} \ a' \longrightarrow \mathtt{nbe} \ a = \mathtt{nbe} \ a'$$

Normalization by evaluation in a model

Normalization by "evaluation" in a model. `reify` is a left inverse of `eval` - the "inverse of the evaluation function":

$$\texttt{nbe}\ a = (\texttt{reify}\ (\texttt{eval}\ a))\ \texttt{conv}\ a$$

Strictification:

$$a\ \texttt{conv}\ a'$$

implies

$$\texttt{eval}\ a = \texttt{eval}\ a'$$

implies

$$\texttt{nbe}\ a = \texttt{nbe}\ a'$$

Formalizing typed combinatory logic in Martin-Löf type theory

Constructors for `Ty : Set`:

```
X     : Ty
(=>)  : Ty -> Ty -> Ty
```

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 =
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

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.
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))
```

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)

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

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 $\mathtt{Exp}$

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

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

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

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

Formal neighbourhoods

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

- formal neighbourhood = finite approximation of the canonical form of a program (lazily evaluated); in particular Δ 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 $\texttt{Exp}$. Operationally, U is the set of all programs of type $\texttt{Exp}$ which approximate the canonical form of the program. Notions of *inclusion* $\supseteq$ and *intersection* $\cap$ of neighbourhoods.
A grammar for expression neighbourhoods:

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

A grammar for the sublanguage of normal form neighbourhoods:

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

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 $\mathtt{Exp}$ is the set

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

One can define formal inclusion and formal intersection and prove that α is a *filter* of normal form neighbourhoods:

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

Combinatory conversion

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

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

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

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

One can prove that two convertible expressions have the same Böhm tree, using the Church-Rosser property.

Operational neighbourhoods of nbe

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

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

$\mathtt{Y}$ 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 σ and τ, respectively, then

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

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

Operational and denotational function neighbourhoods

Let f be a program of type $\sigma \to \tau$, then

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

Operationally iff for all $i < n$, $a \in U_i$ implies $f\ a \in V_i$.

Denotationally iff whenever you know for all $i < n$ that a hypothetical input is approximated by U_i, then the output to f is approximated by V_i.

These are different, because of the *full abstraction problem* discovered by Plotkin: there is a formally consistent neighbourhood (parallel or) which is uninhabited by any program in PCF (and Haskell).

Which is the right one??

Neighbourhoods in Sem

- Δ 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.

Denotational semantics of application

Recall that

```
eval (App e e') = appsem (eval e) (eval e')
```

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

Hence

$$\texttt{eval}\,(\texttt{App e e}') \in V$$

iff there exists U such that $\texttt{eval}\, e \in \texttt{Gl}\, \Delta\, [U; V]$ and $\texttt{eval}\, e' \in U$.

Nbe maps convertible terms into equal Böhm trees

Some facts

- `nbe` $e \in U$ implies that U is a normal form neighbourhood, and hence the denotation of `nbe` e is a combinatory Böhm tree.
- `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:

$$e \rhd^{\mathrm{Bt}} U \;\; \text{implies} \;\; \mathtt{nbe}\, e \in U$$

The proof is by induction on the derivation of $e \rhd^{\mathrm{Bt}} U$. Consider eg the case when $e \rhd^{\mathrm{Bt}} \mathtt{K}$ 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 $e \rhd^{\mathrm{Gl}} V$ is defined by induction on V: 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. In the case of an application we use crucially the *denotational* definition of neighbourhoods!
Soundness then follows immediately.

Summary

- Nbe-algorithm for typed combinatory logic generalizes immediately to one for untyped combinatory logic.
- In the typed case it computes normal forms. In the untyped case it computes Böhm trees
- In the typed case the proof falls out naturally in the setting of constructive type theory (a framework for total functions). In the untyped case we need domain theory. In particular we need domain-theoretic (denotational) definition of approximation, rather than the operational one!
- In the typed case we prove correctness by "glueing" - a variant of Tait-reducibility. In the untyped case we need to adapt the glueing method to work on a reflexive domain.

Case $e@e'$.

To prove that $\texttt{eval}\,(e@e') \in V$ implies $(e@e') \rhd^{\mathrm{Gl}} V$ from the induction hypotheses that $\texttt{eval}\,e \in U$ implies $e \rhd^{\mathrm{Gl}} U$ for all U and $\texttt{eval}\,e' \in U'$ implies $e' \rhd^{\mathrm{Gl}} U'$ for all U' we do case analysis on V:

- $V = \Delta$ and we are done.
- Or there exists U such that $\texttt{eval}\,e \in \texttt{Gl}\,\Delta\,[U; V]$ and $\texttt{eval}\,e' \in U$. In this case the induction hypotheses tells us that $e' \rhd^{\mathrm{Gl}} U$ and $e \rhd^{\mathrm{Gl}} \texttt{Gl}\,\Delta\,[U; V]$. But then it follows immediately from the definition of the latter that $(e@e') \rhd^{\mathrm{Gl}} V$.