Complete Normalisation in Type Theory

Joey Capper

School of Computer Science, University of Nottingham

FPLab Away Day, Hathersage, 24th of May 2012
Demonstrate a technique for proving completeness of Normalisation by Evaluation (NbE) in Type Theory.

Technique used appears to be quite general (is applicable to some rather complex systems, e.g.: dependent types).

The example system in this paper is Gödel’s system $T$.

1. First, find an appropriate semantic model for system $T$.
2. Demonstrate how the representation can be refined to make the completeness proof really easy!
1. Demonstrate a technique for proving completeness of Normalisation by Evaluation (NbE) in Type Theory.

2. Technique used appears to be quite general (is applicable to some rather complex systems, e.g.: dependent types).

3. The example system in this paper is Gödel’s system T.
   1. First, find an appropriate semantic model for system T.
   2. Demonstrate how the representation can be refined to make the completeness proof really easy!
1. Demonstrate a technique for proving completeness of Normalisation by Evaluation (NbE) in Type Theory.
2. Technique used appears to be quite general (is applicable to some rather complex systems, e.g.: dependent types).
3. The example system in this paper is Gödel’s system T.
   1. First, find an appropriate semantic model for system T.
   2. Demonstrate how the representation can be refined to make the completeness proof really easy!
1. Demonstrate a technique for proving completeness of Normalisation by Evaluation (NbE) in Type Theory.
2. Technique used appears to be quite general (is applicable to some rather complex systems, e.g.: dependent types).
3. The example system in this paper is Gödel’s system T.
   1. First, find an appropriate semantic model for system T.
   2. Demonstrate how the representation can be refined to make the completeness proof really easy!
1. Demonstrate a technique for proving completeness of Normalisation by Evaluation (NbE) in Type Theory.

2. Technique used appears to be quite general (is applicable to some rather complex systems, e.g.: dependent types).

3. The example system in this paper is Gödel’s system $T$.
   1. First, find an appropriate semantic model for system $T$.
   2. Demonstrate how the representation can be refined to make the completeness proof really easy!
Gödel’s System T
Gödel’s System T: Types

**Types**

```latex
\textbf{data} \ Type : \ Set \ \textbf{where}
\begin{align*}
\text{nat} & : \ Type \\
\_ \Rightarrow \_ & : \ Type \rightarrow \ Type \rightarrow \ Type
\end{align*}
```
Gödel’s System T: Terms

\[
\text{Terms}
\]

\[
\text{data } Tm : \ Type \rightarrow Set \ \text{where}
\]

\[
\begin{align*}
\text{ZERO} & : Tm \ nat \\
\text{SUCC} & : Tm (nat \Rightarrow nat) \\
\text{FOLD} & : Tm (\tau \Rightarrow (nat \Rightarrow \tau \Rightarrow \tau) \Rightarrow nat \Rightarrow \tau) \\
S & : Tm ((\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_3) \Rightarrow (\tau_1 \Rightarrow \tau_2) \Rightarrow (\tau_1 \Rightarrow \tau_3)) \\
K & : Tm (\tau_1 \Rightarrow \tau_2 \Rightarrow \tau_1) \\
\_\_ & : Tm (\tau_1 \Rightarrow \tau_2) \rightarrow Tm \ \tau_1 \rightarrow Tm \ \tau_2
\end{align*}
\]
Normalisation by Evaluation: A Whirlwind Introduction

1. Define an interpreter \([\_]\) from terms to objects in a suitable model.
2. **reify** the objects back to the normal forms that they represent.
3. Finally: normalisation = reify \(\circ\) \([\_]\)!
A Model of System \( T \)

Terms and normal forms are given by \( Tm \). But what about the model?

**An obvious (but ill-fated) approach**

\[
\begin{align*}
Model' &: Type \to Set \\
Model' \text{ nat} &= \mathbb{N} \\
Model' (\tau \to \sigma) &= Model' \tau \to Model' \sigma
\end{align*}
\]

**Bad news for reification**

\[
\begin{align*}
\text{reify} &: (\tau : Type) \to Model' \tau \to Tm \tau \\
\text{reify nat} &: n = ... \\
\text{reify} (\tau \to \sigma) f &= \{!!\} \quad -- \text{Impossible.}
\end{align*}
\]
A Model of System $T$

Terms and normal forms are given by $Tm$. But what about the model?

An obvious (but ill-fated) approach

\[
\begin{align*}
Model' : & \text{Type } \rightarrow \text{Set} \\
Model' \text{ nat} & = \mathbb{N} \\
Model' (\tau \Rightarrow \sigma) & = Model' \tau \rightarrow Model' \sigma
\end{align*}
\]

Bad news for reification

\[
\begin{align*}
\text{reify} : (\tau : \text{Type}) & \rightarrow Model' \tau \rightarrow Tm \tau \\
\text{reify nat} n & = ... \\
\text{reify} (\tau \Rightarrow \sigma) f & = \{ !! \} \quad -- \text{Impossible.}
\end{align*}
\]
A Model of System T

Terms and normal forms are given by $Tm$. But what about the model?

An obvious (but ill-fated) approach

$$Model' : Type \rightarrow Set$$
$$Model' \text{ nat} = \mathbb{N}$$
$$Model' (\tau \Rightarrow \sigma) = Model' \tau \rightarrow Model' \sigma$$

Bad news for reification

$$reify : (\tau : Type) \rightarrow Model' \tau \rightarrow Tm \tau$$
$$reify \text{ nat} \ n = \ldots$$
$$reify (\tau \Rightarrow \sigma) \ f = \{!!\} \ -- \ Impossible.$$
Dybjør suggests [?] a solution: *glue* the function space to the term representation it represents.

A *glued* model of system $T$

\[
\begin{align*}
\text{Model} &: \ Type \rightarrow Set \\
\text{Model} \ nat &= \mathbb{N} \\
\text{Model} \ (\tau \Rightarrow \sigma) &= Tm \ (\tau \Rightarrow \sigma) \times (\text{Model} \ \tau \rightarrow \text{Model} \ \sigma)
\end{align*}
\]
Reification

Reification is now straightforward:

Reification of the *glued* model

\[
\text{reify} : (\tau : \text{Type}) \to \text{Model} \; \tau \to Tm \; \tau \\
\text{reify nat \; zero} = \text{ZERO} \\
\text{reify nat \; (succ \; n)} = \text{SUCC} \cdot (\text{reify nat} \; n) \\
\text{reify} (\tau \Rightarrow \sigma) \; (F, f) = F
\]
Interpretation of the simple model is pretty straightforward ...

**Interpretation of the simple model**

\[
\begin{align*}
\llbracket\_\rrbracket & : Tm \tau \rightarrow Modell' \tau \\
\llbracket ZERO \rrbracket &= zero \\
\llbracket SUCC \rrbracket &= succ \\
\llbracket FOLD \rrbracket &= fold \\
\llbracket S \rrbracket &= \lambda p q r \rightarrow p r (q r) \\
\llbracket K \rrbracket &= \lambda p q \rightarrow p \\
\llbracket f \cdot x \rrbracket &= \llbracket f \rrbracket [x]
\end{align*}
\]
Interpretation ... with Glue

Interpretation of the **glued** model is also straightforward, albeit messy:

**Interpretation of the **glued** model**

\[
\begin{align*}
\llbracket_\_\rrbracket & : Tm \tau \rightarrow Model \tau \\
\llbracket ZERO \rrbracket & = \text{zero} \\
\llbracket SUCC \rrbracket & = \text{SUCC, succ} \\
\llbracket FOLD \rrbracket & = \text{FOLD, } (\lambda p \rightarrow \ldots \text{ fold } p \ (\lambda n r \rightarrow q \bullet n \bullet r) \ r) \\
\llbracket S \rrbracket & = S, (\lambda p \rightarrow \ldots (\lambda r \rightarrow (p \bullet r \bullet (q \bullet r)))\ldots)) \\
\llbracket K \rrbracket & = K, (\lambda p \rightarrow (K \cdot \text{reify } - p), (\lambda q \rightarrow p)) \\
\llbracket f \cdot x \rrbracket & = \llbracket f \rrbracket \bullet \llbracket x \rrbracket
\end{align*}
\]
Completeness

Completeness
Completeness

The completeness property

\[ t \sim_\beta \text{ normalise } t \]

The relation \( \sim_\beta \) is the *equational theory*, it is:

- An equivalence relation (reflexive, symmetric, and transitive)
- A relation expressing the \( \beta \)-rules of the language, e.g.:

Reduction rules for \( \text{FOLD} \)

\[
\begin{align*}
\text{FOLD} \cdot s \cdot t \cdot \text{ZERO} & \sim_\beta s \\
\text{FOLD} \cdot s \cdot t \cdot (\text{SUCC} \cdot n) & \sim_\beta t \cdot n \cdot (\text{FOLD} \cdot s \cdot t \cdot n)
\end{align*}
\]
Completeness

The completeness property

\[ t \sim_\beta \text{ normalise } t \]

The relation \( \sim_\beta \) is the *equational theory*, it is:

- An equivalence relation (reflexive, symmetric, and transitive)
- A relation expressing the \( \beta \)-rules of the language, e.g.:

Reduction rules for FOLD

\[
\begin{align*}
\text{FOLD} \cdot s \cdot t \cdot \text{ZERO} & \sim_\beta s \\
\text{FOLD} \cdot s \cdot t \cdot (\text{SUCC} \cdot n) & \sim_\beta t \cdot n \cdot (\text{FOLD} \cdot s \cdot t \cdot n)
\end{align*}
\]
Completeness

The completeness property

\[ t \sim_\beta \text{ normalise } t \]

The relation \( \sim_\beta \) is the *equational theory*, it is:

- An equivalence relation (reflexive, symmetric, and transitive)
- A relation expressing the \( \beta \)-rules of the language, e.g.:

**Reduction rules for FOLD**

\[
\begin{align*}
\text{FOLD} \cdot s \cdot t \cdot \text{ZERO} & \sim_\beta s \\
\text{FOLD} \cdot s \cdot t \cdot (\text{SUCC} \cdot n) & \sim_\beta t \cdot n \cdot (\text{FOLD} \cdot s \cdot t \cdot n)
\end{align*}
\]
Completeness

The completeness property

\[ t \sim_\beta \text{ normalise } t \]

The relation \( \sim_\beta \) is the *equational theory*, it is:

- An equivalence relation (reflexive, symmetric, and transitive)
- A relation expressing the \( \beta \)-rules of the language, e.g.:

**Reduction rules for FOLD**

\[
\begin{align*}
\text{FOLD} \cdot s \cdot t \cdot \text{ZERO} & \sim_\beta s \\
\text{FOLD} \cdot s \cdot t \cdot (\text{SUCC} \cdot n) & \sim_\beta t \cdot n \cdot (\text{FOLD} \cdot s \cdot t \cdot n)
\end{align*}
\]
The main idea behind the technique is to index the codomain of the normalisation function on its domain.

Indexing

\[
norm : Tm \tau \rightarrow Tm \tau
\]
\[
norm = ...\]
The Technique: Indexing

The main idea behind the technique is to index the codomain of the normalisation function on its domain.

**Indexing**

\[
norm : (t : Tm \tau) \rightarrow Tm' t
\]
\[
norm = \ldots
\]
The Technique: Indexing

The indexing is fairly obvious, with only one interesting new constructor:

```
data Tm' : Tm τ → Set where
    ZERO : Tm' ZERO
    SUCC : Tm' SUCC
    REC : Tm' REC
    S    : Tm' S
    K    : Tm' K
    _ _  : Tm' s → Tm' t → Tm' (s · t)
    _ :: _ : s \sim_{\beta} t → Tm' s → Tm' t
```
The indexing is also extended to the Model:

Term indexed models

\[
\begin{align*}
\text{Model}' : \{ \tau : \text{Type} \} & \rightarrow Tm \tau \rightarrow \text{Set} \\
\text{Model}' \{ \text{nat} \} & \quad t = \mathbb{N}' t \\
\text{Model}' \{ \tau \Rightarrow \sigma \} & \quad t = Tm' t \times (\text{Model}' s \rightarrow \text{Model}' (t \cdot s))
\end{align*}
\]
The Technique: Casts

When computational behaviour is expressed in the interpreter, a cast is inserted to show that it is a valid step.

**Using the cast constructors**

\[
\llbracket \_ \rrbracket : (t : Tm \tau) \rightarrow Value t \\
\llbracket K \rrbracket = K, (\lambda p \rightarrow (K \cdot reify \_ p), (\lambda q \rightarrow \beta K :: p)) \\
\llbracket \_ \_ \rrbracket = \_ \\
\]

**The \(\beta K\) rule**

\[
K \cdot x \cdot y \sim_\beta x
\]
The Technique: Casts

When computational behaviour is expressed in the interpreter, a cast is inserted to show that it is a valid step.

Using the cast constructors

\[
\begin{align*}
\llbracket \_ \rrbracket & : (t : Tm \tau) \rightarrow Value \ t \\
\llbracket K \rrbracket & = K, (\lambda p \rightarrow (K \cdot reify \_ p), (\lambda q \rightarrow \beta K :: p)) \\
\llbracket \ldots \rrbracket & = \ldots
\end{align*}
\]

The \( \beta K \) rule

\[K \cdot x \cdot y \sim_{\beta} x\]
The Technique: Embedding

Finally, we give an embedding from $Tm'$ to $Tm$ ... 

Using the cast constructors

\[
\begin{align*}
emb & : Tm' t \rightarrow Tm \tau \\
emb \text{ ZERO} & = \text{ ZERO} \\
emb \text{ ...} & = \text{ ...} \\
emb (f \cdot x) & = emb f \cdot emb x \\
emb (p :: t) & = emb t
\end{align*}
\]
The Technique: Preservation of the Embedding

... and show that the embedding preserves the equational theory ...

Using the cast constructors

\[
\begin{align*}
\text{embResp} & : (nf : Tm' t) \rightarrow \text{emb } nf \sim_\beta t \\
\text{embResp ZERO} & = \text{refl} \\
\text{embResp ...} & = \ldots \\
\text{embResp} (f \cdot x) & = \text{appCong} (\text{embResp } f) (\text{embResp } x) \\
\text{embResp} (p :: t) & = \text{tran} (\text{embResp } t) p
\end{align*}
\]
The Proof

... and as promised, the final proof:

Using the cast constructors

\[
complete : (t : Tm \tau) \rightarrow emb \ (\text{norm } t) \sim_\beta t
\]

\[
complete \ t = embResp \ (\text{norm } t)
\]
Questions

Thankyou. Questions?