A formalization of normalization by evaluation Deep and shallow embeddings of simple types in Coc

François Garillot, Benjamin Werner

INRIA-Futurs, ENS and LIX

TPHOLs, September 2007, Kaiserslautern

Today: a modest contribution

Main Related Work:

- ► Catarina Coquand (first heard in 1992)
- ► Ulrich Berger, Helmut Schwichtenberg, Stefan Berghofer, Pierre Letouzey...
- ► Olivier Danvy a.o.

Motivations:

- ▶ not very precise
- understanding and handling of binders

Today: a modest contribution

Main Related Work:

- Catarina Coquand (first heard in 1992)
- ► Ulrich Berger, Helmut Schwichtenberg, Stefan Berghofer, Pierre Letouzey...
- ► Olivier Danvy a.o.

Motivations:

- ▶ not very precise
- understanding and handling of binders
- ► Challenging problem (for me)

deep vs. shallow

Two "representations" of $\lambda x.x$ in Type Theory :

► Shallow embedding fun x => x : T -> T

deep vs. shallow

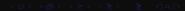
```
Two "representations" of \lambda x.x in Type Theory : 

Shallow embedding fun x => x : T -> T
```

➤ or the deep embedding.
Define:

```
Inductive term : Type :=
   Var : id -> term
| Lam : id -> term -> term
| App : term -> term -> term.
Lam x (Var x) : term
p : WT (Lam x (Var x)) (Arr Iota Iota)
```

How can we switch from one another?



Talk outline

- 1. The picture : basic definitions
- 2. From deep to shallow
- 3. From shallow to deep

A syntax with named variables

```
Inductive ST : Set :=
  |Iota : ST | Arr : ST -> ST -> ST.
Record id : Type := mkid {idx : nat ; idT : ST}.
Inductive term : Type :=
 Var : id -> term
Lam : id -> term -> term
App : term -> term -> term.
Regular concrete data-types
```

Inductively:

$$\frac{t:B}{x^A:A} \qquad \frac{t:B}{\lambda x^A.t:A\to B} \qquad \frac{t:A\to B \qquad u:A}{t\;u:B}$$

Inductively:

$$\frac{t:B}{x^A:A} \qquad \frac{t:B}{\lambda x^A.t:A\to B} \qquad \frac{t:A\to B \quad u:A}{t\;u:B}$$

Not the most practical way when we have dependent types

We take a more computational approach...

```
Fixpoint inferc (t:term) : option ST :=
match t with
| Var n => Some n.(idT)
 App t u =>
  match inferc t, inferc u with
     | Some (Arr A B) , Some C =>
        if C == A then Some B else None
     _ , _ => None
  end
Lam n t => match inferc t with
   | Some B => Some (Arr n.(idT) B)
   ___ => None
end
end.
Definition WT t T := inferc t = Some T.
```

```
(* The key definition : lifting types to Coq *)
Fixpoint tr (alpha:Type)(T:ST) {struct T}: Type:=
  match T with
| Iota => alpha
| Arr A B => (tr alpha A)->(tr alpha B)
end.
```

Two choices to be made:

- ► One may prefer a more complex interpretation for arrow types (see C. Coquand).
- ► One needs to chose alpha.

I will chose alpha=term

Not the best solution



shallow	deep
Syntax	Semantics
Source code	Executable code
t :term	f:(tr term T)

shallow		deep
Syntax		Semantics
Source code		Executable code
t :term		f:(tr term T)
t:term,WT t T	\longrightarrow comp	[t] ₁ :(tr term T)
	$\leftarrow\!$	

shallow		deep
Syntax		Semantics
Source code		Executable code
t :term		f:(tr term T)
WT t T		condition(s) on f
t:term,WT t T	\longrightarrow comp	$[t]_i:(tr term T)$
	$\overset{\longleftarrow}{decomp}$	

shallow		deep
Syntax		Semantics
Source code		Executable code
t :term		f:(tr term T)
WT t T		condition(s) on f
t:term,WT t T	$\stackrel{\longrightarrow}{comp}$	$[t]_I: (tr term T)$
	←—	
	decomp	

- ► compilation : (relatively) easy
- ► decompilation : a little trickier

Going up : compilation

Idea: straightforward semantics

$$\begin{aligned} [x]_{\mathcal{I}} &= & \mathcal{I}(x) \\ [\lambda x.t]_{\mathcal{I}} &= & \text{fun } \alpha \mapsto [t]_{\mathcal{I};x \leftarrow \alpha} \\ [t \ u]_{\mathcal{I}} &= & [t]_{\mathcal{I}}([u]_{\mathcal{I}}) \end{aligned}$$

Only technical difficulty : The semantics is only defined for well-typed terms

Going up : compilation

Idea: straightforward semantics

$$\begin{aligned} [x]_{\mathcal{I}} &= & \mathcal{I}(x) \\ [\lambda x.t]_{\mathcal{I}} &= & \text{fun } \alpha \mapsto [t]_{\mathcal{I};x \leftarrow \alpha} \\ [t \ u]_{\mathcal{I}} &= & [t]_{\mathcal{I}}([u]_{\mathcal{I}}) \end{aligned}$$

Only technical difficulty : The semantics is only defined for well-typed terms

```
env := forall x:id, tr term (x).idT
comp : forall t T, WT t T -> env -> tr T alpha
```

Going up : compilation

Idea: straightforward semantics

$$\begin{aligned} [x]_{\mathcal{I}} &= & \mathcal{I}(x) \\ [\lambda x.t]_{\mathcal{I}} &= & \text{fun } \alpha \mapsto [t]_{\mathcal{I};x \leftarrow \alpha} \\ [t \ u]_{\mathcal{I}} &= & [t]_{\mathcal{I}}([u]_{\mathcal{I}}) \end{aligned}$$

Only technical difficulty:

The semantics is only defined for well-typed terms

```
env := forall x:id, tr term (x).idT
comp : forall t T, WT t T -> env -> tr T alpha
```

works but is not practical: the function depends upon t but the types depend upon T.

Reasonning about such functions can be surprisingly tedious.



Solution:

- ▶ Type-checking is done at compile-time : comp : term -> option {T:ST |env -> tr T alpha}
- ► Hide the equality test

Interpreting free variables

The "default" interpretation of variables :

$$\mathcal{I}(x^A) \equiv \log(A, \operatorname{Var}(x^A))$$

Really simple...

Really simple semantics

The semantics are actually simpler than the syntax: fun f (h:term->term) u => h (f u (Var x)(App (Var y) u)) is the "semantics" of : Lam (mkid 0 (Iota ==> Iota ==> Iota ==> Iota)) (Lam (mkid 5 (Iota ==> Iota)) (Lam (mkid 4 Iota) (App (Var (mkid 5 (Iota ==> Iota))) (App (App (App (Var (mkid 0 (Iota ==> Iota ==> Iota ==> Iota))) (Var (mkid 4 Iota))) (Var x)) (App (Var y) (Var (mkid 4 Iota)))))))

Decompilation : principle

```
idea: look for the \beta-normal, \eta-long form.
This time, really "type" directed:
     decomp(Iota, t) = t
   decomp(A \Rightarrow B, f) = Lam(x, decomp(B, f long(A, x)))
        long(Iota, t) = t
      long(A \Rightarrow B, t) = a \mapsto long(B, App(t, decomp(A, a)))
```

Decompilation: principle

```
idea: look for the \beta-normal, \eta-long form.
This time, really "type" directed:
     decomp(Iota, t) = t
   decomp(A \Rightarrow B, f) = Lam(x, decomp(B, f long(A, x)))
        long(Iota, t) = t
      long(A \Rightarrow B, t) = a \mapsto long(B, App(t, decomp(A, a)))
"little problem" : find a fresh x...
```

Good solution : Berger

Have the decompiled function to be parametrized by its context.

context = number upon which variables are free.

```
use (tr (nat \rightarrow term) T) (more complex semantics, free variables more difficult to handle) But if I want to stick to (tr term T)?
```

"Horrible" trick

- 1. take a fixed dummy variable d
- 2. compute decomp(B, f(long(A,d)))
- 3. find a variable y not free in (decomp(B, f(long(A,d))))
- 4. return decomp(f(long(y)))

```
Works but... exponentially slower (with some optimization, quadratically slower)
```

Can one do (really) better? I do not know

Actually, a related construction can be found in Berger & Schwichtenberg 1991 (LICS).

Normalization proof

Pasting things together

Two steps:

- ▶ show that decomp o comp returns normal forms (easy)
- lacktriangle show that it preserves the $=_{eta\eta}$ class (where things happen).

"Main theorem" : weak normalization of simply typed calculus

logical relation :

$$t \simeq_{\iota} st \Leftrightarrow t =_{\beta\eta} st$$

 $t \simeq_{A \to B} st \Leftrightarrow \forall u \ su, u \simeq_{A} su \Rightarrow App(t, u) \simeq_{B} st(su)$

let σ be a substitution,

$$\forall x \in FV(t), \sigma(x) \simeq I(x)$$

then

$$t \simeq [t]_I$$

(* to be precise : see code *)

How is the dummy trick treated?

Lemma : if $(t \ x) =_{\beta \eta} u$ and $y \notin FV(u)$, there exists $t' =_{\beta \eta} t$, with $y \notin FV(t')$.

François found nice definitions and lemmas for α -conversion in a paper by Allen Stoughton : Substitution Revisited.

Use a notion of " α -normalization"

Not surprisingly, the most tedious part of the proof. See http://benjamin.werner.name

Future versions should be done with nameless variables

Technical conclusion

- NbE is possible with a very simple typing on the semantics' side
- ▶ The good categorical interpretation is for λ -terms with context; thus the routine is less elegant and less efficient (price to pay for the simplicity of typing)

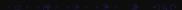
Technical conclusion

- NbE is possible with a very simple typing on the semantics' side
- The good categorical interpretation is for λ-terms with context; thus the routine is less elegant and less efficient (price to pay for the simplicity of typing)
- No context ⇒ free variables can be added freely ⇒ convenient but need for dynamically checking which variables are free (other explanation for the overhead)

Technical conclusion

- NbE is possible with a very simple typing on the semantics' side
- The good categorical interpretation is for λ-terms with context; thus the routine is less elegant and less efficient (price to pay for the simplicity of typing)
- No context ⇒ free variables can be added freely ⇒ convenient but need for dynamically checking which variables are free (other explanation for the overhead)

Is this of some use?



Applications?

Remember Higher-Order Abstract Syntax

A language with binders is described by a context in simply typed λ -calculus :

$$[APP : \iota \to \iota \to \iota; LAM : (\iota \to \iota) \to \iota]$$

a (pure) λ -term is described by a simply typed λ -term of type ι , whose variables are APP, LAM of variables of type ι .

Applications?

Remember Higher-Order Abstract Syntax

A language with binders is described by a context in simply typed λ -calculus :

$$[APP : \iota \to \iota \to \iota; LAM : (\iota \to \iota) \to \iota]$$

a (pure) λ -term is described by a simply typed λ -term of type ι , whose variables are APP, LAM of variables of type ι .

Future Work

Re-do it with locally nameless (ie. de Bruijn for bounded var.) à la Pierce, Weirich, Charguéraud...

Try to use it : construct the good induction schemes for these terms, the nice syntactic sugar...

...Work in progress...