

# *A dependently-typed formalization of $\lambda^{\rightarrow}$ , substitution, denotation, normalization.*

*Draft of June 18, 2007*

Matthieu Sozeau

*Univ. Paris Sud, CNRS, Laboratoire LRI, UMR 8623, Orsay, F-91405  
INRIA Futurs, ProVal, Parc Orsay Universit, F-91893  
sozeau@lri.fr*

We will develop a complete formalization of simply-typed lambda-calculus with constants in COQ, using the Program extension to write dependently-typed programs on inductive families. The development uses a pure de Bruijn encoding and dependently-typed abstract syntax. It includes the theory of lifting and substitution, a denotational semantics given by interpreting terms directly into COQ, and finally a proof of weak normalization for the call-by name strategy.

This paper describes all the technical parts of the development. It is a literate coqdoc script, missing only the proof scripts.

First we define a set of tactics that will be used later.

Rewrite using uniqueness of identity proofs  $H = refl\_equal X$ .

```
Ltac simpl_uip :=
  match goal with
  [ H : ?X = ?X ⊢ _ ] ⇒ rewrite (UIP_refl _ _ H) in ×; clear H
  end.
```

Try to abstract a proof of equality, if no proof of the same equality is present in the context.

```
Ltac abstract_eq_hyp H' p :=
  match type of p with
  ?X = ?Y ⇒
  match goal with
  | [ H : X = Y ⊢ _ ] ⇒ fail 1
  | _ ⇒ set (H':=p) ; clearbody H'
  end
  end.
```

Apply the tactic tac to proofs of equality appearing as coercion arguments.

```
Ltac on_coerce_proof tac :=
```

```

match goal with
  [ | - ?T ] =>
  match T with
    | context [ eq_rect _ _ _ ?p ] => tac p
  end
end.

```

Abstract proofs of equalities of coercions.

```

Ltac abstract_eq_proof := on_coerce_proof ltac:(fun p => let H := fresh "eqH"
in abstract_eq_hyp H p).

```

Factorize proofs, by using proof irrelevance so that two proofs of the same equality in the goal become convertible.

```

Ltac pi_eq_proof_hyp p :=
  match type of p with
  ?X = ?Y =>
  match goal with
    | [ H : X = Y ⊢ _ ] =>
      match p with
        | H => fail 2
        | _ => rewrite (proof_irrelevance (X = Y) p H)
      end
    | _ => fail " No hypothesis with same type "
  end
end.

```

Factorize proofs of equality appearing as coercion arguments.

```

Ltac pi_eq_proof := on_coerce_proof pi_eq_proof.

```

Clear unused reflexivity proofs.

```

Ltac clear_refl_eq :=
  match goal with [ H : ?X = ?X ⊢ _ ] => clear H end.
Ltac clear_refl_eqs := repeat clear_refl_eq.

```

Clear unused equality proofs.

```

Ltac clear_eq :=
  match goal with [ H : _ = _ ⊢ _ ] => clear H end.
Ltac clear_eqs := repeat clear_eq.

```

Everything is done in implicit arguments mode.

## 1 Constants

The development is parameterized by module implementing the constants and their typing.

Module Type *Constants*.

The set of constant types.

Parameter *constant\_types* : Set.

The constants themselves.

Parameter *constants* : Set.

A typing function for constants.

Parameter *type\_constant* : *constants*  $\rightarrow$  *constant\_types*.

End *Constants*.

## 2 The implementation.

The functor  $\Lambda$  implements typing, denotational semantics and normalization of simply-typed lambda-calculus plus constants  $C$ .

Module  $\Lambda$  ( $C$  : *Constants*).

Import  $C$ .

### 2.1 Abstract syntax

The types are just atoms or the arrow.

```
Inductive type : Set :=
| type_cst : constant_types  $\rightarrow$  type
| type_arrow : type  $\rightarrow$  type  $\rightarrow$  type.
```

Coercion *type\_cst* : *constant\_types*  $\hookrightarrow$  type.

Infix "  $\rightarrow$  " := *type\_arrow* (at level 20).

A context is either empty or a context and a variable. Contexts are effectively snoc-lists, hence the new inductive.

```
Inductive ctx : Set :=
| [] : ctx
| comma : ctx  $\rightarrow$  type  $\rightarrow$  ctx.
```

Infix ", " := *comma* (at level 40, left associativity).

Appending one context to another, by induction on the second one.

```
Fixpoint append ( $\Gamma$   $\Delta$  : ctx) { struct  $\Delta$  } : ctx :=
  match  $\Delta$  with
  | []  $\Rightarrow$   $\Gamma$ 
  |  $\Delta'$  ,  $x$   $\Rightarrow$  (append  $\Gamma$   $\Delta'$ ) ,  $x$ 
  end.
```

Hint *Unfold append*.

Infix ”;” := *append* (*right associativity, at level 60*).

Due to the way *append* is defined, these two are not convertible.

Lemma *app\_empty* :  $\forall \Gamma, ([ ] ; \Gamma) = \Gamma$ .

Associativity of context appending.

Lemma *app\_assoc* :  $\forall \Gamma \Delta \Lambda, \Gamma ; \Delta ; \Lambda = (\Gamma ; \Delta) ; \Lambda$ .

Hint *Rewrite app\_empty app\_assoc* : *app*.

Opaque *app\_assoc*.

Simplification tactic using the previous lemmas.

Ltac *my\_simpl* := *subtac\_simpl* ; *try simpl\_JMeq* ; *autorewrite with app* ; *subtac\_simpl*.

Variables are encoded as typed de Bruijn indices.

Inductive *var* : *ctx*  $\rightarrow$  *type*  $\rightarrow$  *Set* :=  
 | *first* :  $\forall \Gamma T, \text{var } (\Gamma, T) T$   
 | *next* :  $\forall \Gamma T, \text{var } \Gamma T \rightarrow \forall U, \text{var } (\Gamma, U) T$ .

The AST of well-typed terms. Constants get their type from the *type\_constant* function. A *rel* just embeds a variable. The abstraction constructor *lam* contains a body typed in an extended environment. Application applies only functions to objects having their domain type.

Inductive *term* ( $\Gamma : \text{ctx}$ ) : *type*  $\rightarrow$  *Set* :=  
 | *cst* :  $\forall c : \text{constants}, \text{term } \Gamma (\text{type\_constant } c)$   
 | *rel* :  $\forall T, \text{var } \Gamma T \rightarrow \text{term } \Gamma T$   
 | *lam* :  $\forall (T T' : \text{type}),$   
      $\text{term } (\Gamma, T) T' \rightarrow \text{term } \Gamma (T \rightarrow T')$   
 | *app* :  $\forall T T' : \text{type},$   
      $\text{term } \Gamma (T \rightarrow T') \rightarrow \text{term } \Gamma T \rightarrow \text{term } \Gamma T'$ .

The really dependent scheme for term. Not used in the development but may be useful.

Scheme *term\_dep* := *Induction for term Sort Prop*.

Tactic to automatically invert contradictory hypotheses.

Ltac *try\_inversion* :=  
 match *goal* with  
 | *H* :  $\_ = \_ \vdash \_ \Rightarrow \text{inversion } H$  ; *auto*  
 | *H* :  $\text{var } [ ] \_ \vdash \_ \Rightarrow \text{inversion } H$   
 end.

Use these for solving obligations from now on.

Obligations *Tactic* := *my\_simpl* ; *try try\_inversion* ; *try omega*.

## 2.2 Dealing with coercions and equality.

The next two functions reindex variables and terms in  $\Gamma$  to a new context  $\Gamma'$  with respect to an equality between  $\Gamma$  and  $\Gamma'$ . These are dynamic coercions in the sense that they can be reduced when applied to a constructed term, contrary to the substitution principle *eq\_rect*.

```

Program Fixpoint coerce_var_context  $\Gamma$   $T$  ( $v$  : var  $\Gamma$   $T$ )  $\Gamma'$  ( $H$  :  $\Gamma = \Gamma'$ )
{ struct  $v$  } : var  $\Gamma'$   $T$  :=
  match  $\Gamma'$  with
  | []  $\Rightarrow$  !
  |  $\Gamma''$  ,  $V'$   $\Rightarrow$ 
    match  $v$  with
    | first  $vG$   $vT$   $\Rightarrow$  first  $\Gamma''$   $V'$ 
    | next  $v'G$   $v'T$   $v'$   $V$   $\Rightarrow$ 
      @next  $\Gamma''$   $v'T$  (coerce_var_context  $v'$   $-$ )  $V'$ 
    end
  end
end.

```

```

Program Fixpoint coerce_term_context  $\Gamma$   $T$  ( $t$  : term  $\Gamma$   $T$ )  $\Gamma'$  ( $H$  :  $\Gamma = \Gamma'$ )
{ struct  $t$  } : term  $\Gamma'$   $T$  :=
  match  $t$  with
  | cst  $c$   $\Rightarrow$  cst  $\Gamma'$   $c$ 
  | rel  $T$   $v$   $\Rightarrow$  rel (coerce_var_context  $v$   $H$ )
  | lam  $\tau$   $\tau'$   $b$   $\Rightarrow$  @lam  $\Gamma'$   $-$   $-$  (@coerce_term_context ( $\Gamma$  ,  $\tau$ )  $\tau'$   $b$  ( $\Gamma'$  ,  $\tau$ )  $-$ )
  | app  $\tau$   $\tau'$   $f$   $e$   $\Rightarrow$  @app  $\Gamma'$   $\tau$   $\tau'$  (coerce_term_context  $f$   $H$ )
    (coerce_term_context  $e$   $H$ )
  end
end.

```

Like *eq\_rect*, *coerce* reduces to the identity when  $\Gamma$  and  $\Gamma'$  are convertible.

**Lemma** *coerce\_var\_context\_id* :  $\forall$  ( $\Gamma$  : ctx)  $T$  ( $v$  : var  $\Gamma$   $T$ ),  
*coerce\_var\_context*  $v$  (*refl\_equal*  $\Gamma$ ) =  $v$ .

**Lemma** *coerce\_term\_context\_id* :  $\forall$  ( $\Gamma$  : ctx)  $T$  ( $t$  : term  $\Gamma$   $T$ ),  
*coerce\_term\_context*  $t$  (*refl\_equal*  $\Gamma$ ) =  $t$ .

We now define some tactics that help to deal with coercions. They work in multiple passes. First, we abstract any equality proof appearing in the goal as an argument of a coercion, so as not to pollute it with irrelevant proof terms. Then we factorize the proofs and normalize the term using proof irrelevance so that two references to the same equality proof are the same. Next we try to simplify the equations using Streicher's axiom K (a proof  $p$  of  $x = x$  can be substituted by *refl\_equal*  $x$ ). Finally we try to rewrite the term using the previous lemmas about *coerce* and *refl\_equal*. This procedure is very effective, most of the coercions can be eliminated this way. We get stuck only when an induction is needed.

Abstract a proof of equality, if none is already present in the context.

```

Ltac on_coerce_proof tac :=
  match goal with
  [ | - ?T ] =>
  match T with
  | context [ coerce_var_context _?p ] =>
    tac p
  | context [ coerce_term_context _?p ] =>
    tac p
  | context [ eq_rect ----?p ] =>
    tac p
  end
end.

```

```

Ltac abstract_coerce_proof := on_coerce_proof
  ltac:(fun p =>
    let H := fresh "coerceH" in
    abstract_eq_hyp H p).

```

```

Ltac abstract_coerce_proofs := repeat abstract_coerce_proof.

```

Use proof-irrelevance to eliminate remaining non-abstracted proofs.

```

Ltac pi_coerce_proof := on_coerce_proof ltac:(pi_eq_proof_hyp).
Ltac pi_coerce_proofs := repeat pi_coerce_proof.

```

The two preceding tactics in sequence.

```

Ltac clear_coerce_proofs :=
  abstract_coerce_proofs ; pi_coerce_proofs.

```

Rewrite  $\text{coerce} \times t$  (*refl\_equal* \_) to *t*.

```

Hint Rewrite coerce_term_context_id coerce_var_context_id : coerce_id.
Ltac rewrite_coerce_id := autorewrite with coerce_id.

```

Clear the context and term of equality proofs.

```

Ltac clear_coerce_ctx :=
  rewrite_coerce_id ; clear_coerce_proofs.

```

Repeated elimination of *eq\_rect* applications. Abstracting equalities makes it run much faster than a naive implementation.

```

Ltac simpl_eqs :=
  repeat (real_elim_eq_rect ; simpl ; clear_coerce_ctx).

```

Combine all the tactics to simplify goals containing coercions.

```

Ltac simpl_term :=
  simpl ; simpl_eqs ; clear_coerce_ctx ; clear_refl_eqs ;
  try subst ; simpl ; repeat simpl UIP ; rewrite_coerce_id.

```

In fact *eq\_rect* and *coerce* are always equal, so we can switch between frozen and dynamic coercions at will.

**Lemma** *eq\_rect\_coerce* :  $\forall \Gamma T (t : \text{term } \Gamma T) \Gamma' (Heq : \Gamma = \Gamma'),$   
*eq\_rect* \_ (fun  $\Gamma' \Rightarrow \text{term } \Gamma' T$ )  $t$  \_ *Heq* = *coerce\_term\_context*  $t$  *Heq*.

**Lemma** *eq\_rect\_coerce\_var* :  $\forall \Gamma T (t : \text{var } \Gamma T) \Gamma' (Heq : \Gamma = \Gamma'),$   
*eq\_rect* \_ (fun  $\Gamma' \Rightarrow \text{var } \Gamma' T$ )  $t$  \_ *Heq* = *coerce\_var\_context*  $t$  *Heq*.

Tactic to change *eq\_rect* applications to coercions. Allows to reduce coercions sometimes.

**Hint** *Rewrite eq\_rect\_coerce eq\_rect\_coerce\_var : eq\_rect\_coerce.*

**Ltac** *rew\_eq\_coerce := autorewrite with eq\_rect\_coerce.*

### 3 Lifting and substitution.

We now define lifting and substitution of terms. The specifications of the functions are exactly the lemmas you would expect.

**Section** *Substitution.*

Lifting a variable  $t$  in context  $\Gamma ; \Delta$  by  $U$  to get a variable in  $\Gamma , U ; \Delta$ . The context  $\Delta$  represents the bindings we must avoid to capture.

```

Program Fixpoint lift_var ( $\Gamma \Delta : \text{ctx}$ ) ( $T : \text{type}$ ) ( $t : \text{var } (\Gamma ; \Delta) T$ )
  ( $U : \text{type}$ ) {struct  $t$ } :  $\text{var } (\Gamma , U ; \Delta) T :=
  match \Delta with
  | []  $\Rightarrow$  next  $t$   $U$ 
  |  $\Delta' , - \Rightarrow$ 
    match  $t$  with
    | first  $\Gamma' - \Rightarrow$  first ( $\Gamma , U ; \Delta'$ )  $T$ 
    | next  $\Gamma' T' v V' \Rightarrow$  next (lift_var  $\Gamma \Delta' v U$ )  $V'$ 
    end
  end.$ 
```

We prove the defining equations as lemmas.

**Lemma** *lift\_var\_empty* :  $\forall \Gamma T U (v : \text{var } \Gamma T),$  *lift\_var*  $\Gamma [] v U = \text{next } v U$ .

Lifting a term by a type  $U$  is just a fold.

```

Program Fixpoint lift_rec  $\Gamma \Delta T (t : \text{term } (\Gamma ; \Delta) T) U$ 
  {struct  $t$ } :  $\text{term } (\Gamma , U ; \Delta) T :=
  match  $t$  with
  | cst  $c \Rightarrow$  cst _  $c$ 
  | rel  $T v \Rightarrow$  rel (lift_var  $\Gamma \Delta v U$ )
  | lam  $\tau \tau' b \Rightarrow$  lam (lift_rec  $\Gamma (\Delta , \tau) b U$ )
  | app  $\tau \tau' f e \Rightarrow$  app (lift_rec  $\Gamma \Delta f U$ ) (lift_rec  $\Gamma \Delta e U$ )
  end.$ 
```

The non recursive version, a.k.a. weakening.

**Program Definition**  $lift$  ( $\Gamma : \text{ctx}$ )  $T$  ( $t : \text{term } \Gamma T$ )  $U : \text{term } (\Gamma, U) T := lift\_rec \Gamma [] t U$ .

The defining equations for the variable case.

**Lemma**  $lift\_rec\_empty$  :  $\forall \Gamma T U (v : \text{var } \Gamma T)$ ,  
 $lift\_rec \Gamma [] (\text{rel } v) U = \text{rel } (\text{next } v U)$ .

**Lemma**  $lift\_rec\_comma\_first$  :  $\forall \Gamma T U \Delta$ ,  
 $lift\_rec \Gamma (\Delta, T) (\text{rel } (\text{first } (\Gamma; \Delta) T)) U = \text{rel } (\text{first } (\Gamma, U; \Delta) T)$ .

**Lemma**  $lift\_rec\_comma\_next$  :  $\forall \Gamma T U \Delta T' (v : \text{var } (\Gamma; \Delta) T')$ ,  
 $lift\_rec \Gamma (\Delta, T) (\text{rel } (\text{next } v T)) U = \text{rel } (\text{next } (lift\_var \_ \_ v U) T)$ .

Lifting by a context is lifting by each variable in turn.

**Program Fixpoint**  $lift\_ctx$  ( $\Gamma \Delta : \text{ctx}$ )  $T$  ( $t : \text{term } \Gamma T$ )  
 $\{ \text{struct } \Delta \} : \text{term } (\Gamma; \Delta) T :=$   
 $\text{match } \Delta \text{ with}$   
 $| [] \Rightarrow t$   
 $| \Delta', U \Rightarrow \text{let } t' := lift\_ctx \Delta' t \text{ in}$   
 $lift t' U$   
 $\text{end}$ .

Substitution of a variable by a term in context  $\Gamma$ , under a context  $\Delta$ .

**Program Fixpoint**  $subst\_var\_rec$   $\Gamma \Delta T U$  ( $t : \text{var } (\Gamma, T; \Delta) U$ )  
 $(s : \text{term } \Gamma T) \{ \text{struct } \Delta \} : \text{term } (\Gamma; \Delta) U :=$   
 $\text{match } \Delta \text{ with}$   
 $| [] \Rightarrow$   
 $\text{match } t \text{ with}$   
 $| \text{first } tG tT \Rightarrow s$  (*\*\* substitution takes place \**)  
 $| \text{next } tG' tT' tv \_ \Rightarrow$   
 $\text{rel } tv$  (*\*\* unlift the var to account for substitution \**)  
 $\text{end}$   
 $| \Delta', V \Rightarrow$   
 $\text{match } t \text{ with}$   
 $| \text{first } tG tT \Rightarrow \text{rel } (\text{first } (\Gamma; \Delta') V)$  (*\*\* do nothing \**)  
 $| \text{next } tG' tT' tv V \Rightarrow$  (*\*\* substitute inside, then lift by one \**)  
 $lift (@subst\_var\_rec \Gamma \Delta' T U tv s) V$   
 $\text{end}$   
 $\text{end}$ .

The defining equations.

**Lemma**  $subst\_var\_rec\_empty\_first$  :  $\forall \Gamma T (s : \text{term } \Gamma T)$ ,  
 $subst\_var\_rec [] (\text{first } \Gamma T) s = s$ .

**Lemma**  $subst\_var\_rec\_empty\_next$  :  $\forall \Gamma T (v : \text{var } \Gamma T) T' (s : \text{term } \Gamma T')$ ,  
 $subst\_var\_rec [] (\text{next } v T') s = \text{rel } v$ .

**Lemma** *subst\_var\_rec\_comma\_first* :  $\forall \Gamma T T' (t : \text{term } \Gamma T') \Delta,$   
 $\text{subst\_var\_rec } (\Delta, T) (\text{first } (\Gamma, T'; \Delta) T) t = \text{rel } (\text{first } (\Gamma; \Delta) T).$

**Lemma** *subst\_var\_rec\_comma\_next* :  $\forall \Gamma T T' (t : \text{term } \Gamma T') \Delta$   
 $T'' (v : \text{var } (\Gamma, T'; \Delta) T),$   
 $\text{subst\_var\_rec } (\Delta, T'') (\text{next } v T'') t = \text{lift } (\text{subst\_var\_rec } \Delta v t) T''.$

Non-recursive wrapper to substitute the first variable in the context.

**Program Definition** *subst\_var*  $\Gamma T U (t : \text{var } (\Gamma, T) U) (s : \text{term } \Gamma T) :$   
 $\text{term } \Gamma U := \text{subst\_var\_rec } [] t s.$

Substitution in a term.

**Program Fixpoint** *subst\_rec*  $\Gamma \Delta T U (t : \text{term } (\Gamma, T; \Delta) U)$   
 $(s : \text{term } \Gamma T) \{ \text{struct } t \} : \text{term } (\Gamma; \Delta) U :=$   
 $\text{match } t \text{ with}$   
 $\quad | \text{cst } c \Rightarrow \text{cst } (\Gamma; \Delta) c$   
 $\quad | \text{rel } _ v \Rightarrow @\text{subst\_var\_rec } \Gamma \Delta T U v s$   
 $\quad | \text{lam } \tau \tau' b \Rightarrow$   
 $\quad \quad \text{lam } (\text{subst\_rec } (\Delta, \tau) b s)$   
 $\quad | \text{app } \tau \tau' f e \Rightarrow \text{app } (\text{subst\_rec } \Delta f s) (\text{subst\_rec } \Delta e s)$   
 $\text{end.}$

Wrapped again.

**Program Definition** *subst*  $(\Gamma : \text{ctx}) (T U : \text{type}) (t : \text{term } (\Gamma, T) U)$   
 $(s : \text{term } \Gamma T) : \text{term } \Gamma U := \text{subst\_rec } [] t s.$

The defining equations.

**Lemma** *subst\_rec\_empty\_first* :  $\forall \Gamma T (t : \text{term } \Gamma T),$   
 $\text{subst\_rec } [] (\text{rel } (\text{first } \Gamma T)) t = t.$

**Lemma** *subst\_rec\_empty\_next* :  $\forall \Gamma T T' (t : \text{term } \Gamma T) (v : \text{var } \Gamma T'),$   
 $\text{subst\_rec } [] (\text{rel } (\text{next } v T)) t = \text{rel } v.$

**Lemma** *subst\_rec\_comma\_first* :  $\forall \Gamma T T' (t : \text{term } \Gamma T') \Delta,$   
 $\text{subst\_rec } (\Delta, T) (\text{rel } (\text{first } (\Gamma, T'; \Delta) T)) t = \text{rel } (\text{first } (\Gamma; \Delta) T).$

**Lemma** *subst\_rec\_comma\_next* :  $\forall \Gamma T T' (t : \text{term } \Gamma T') \Delta$   
 $T'' (v : \text{var } (\Gamma, T'; \Delta) T),$   
 $\text{subst\_rec } (\Delta, T'') (\text{rel } (\text{next } v T'')) t = \text{lift } (\text{subst\_rec } \Delta (\text{rel } v) t) T''.$

### 3.1 Commutation lemmas.

We now relate lifting and substitution to ultimately prove the substitution lemma.

Substituting for a variable just lifted is a no-op.

We first typecheck the statement, which will insert coercions around  $t$  to get the right indices. The obligations are automatically resolved using the *Heq* hypothesis.

Note that we separate the equality so that induction can proceed on  $t$  of type `term`  $\Gamma' T$  while still retaining the equation between  $\Gamma'$  and  $\Gamma ; \Delta$ .

**Program Definition** *subst\_lift\_rec\_stmt* :=  
 $\forall \Gamma' T (t : \text{term } \Gamma' T) \Gamma \Delta T' (u : \text{term } \Gamma T') (Heq : \Gamma' = (\Gamma ; \Delta)),$   
 $\text{subst\_rec } \Delta (\text{lift\_rec } \Gamma \Delta t T') u = t.$

Now we prove it, by induction on  $t$ . We first rewrite *eq\_rect* to coercions, then simplify the term using *simpl\_term*. This takes care of almost all the plumbing.

**Lemma** *subst\_lift\_rec* : *subst\_lift\_rec\_stmt*.

The real lemma is then just an instance of the previous one.

**Lemma** *subst\_lift* :  $\forall \Gamma \Delta T (t : \text{term } (\Gamma ; \Delta) T) T' (u : \text{term } \Gamma T'),$   
 $\text{subst\_rec } \Delta (\text{lift\_rec } \Gamma \Delta t T') u = t.$

We now prove that equality coercion commute with lift to solve some of the later lemmas which involve coercions of lift calls. Ideally this should be either easily derivable or automatically generated by some kind of generic programming.

**Lemma** *coerce\_var\_context\_lift\_var* :  
 $\forall \Gamma'' T (t : \text{var } \Gamma'' T) \Gamma \Delta (Heq : \Gamma'' = \Gamma ; \Delta)$   
 $\Gamma' U (H : (\Gamma , U ; \Delta) = (\Gamma' , U ; \Delta)) (H' : \Gamma'' = (\Gamma' ; \Delta)),$   
 $\text{coerce\_var\_context } (\text{lift\_var } \Gamma \Delta (\text{coerce\_var\_context } t \text{ Heq}) U) H =$   
 $\text{lift\_var } \Gamma' \Delta (\text{coerce\_var\_context } t H') U.$

**Lemma** *coerce\_term\_context\_lift\_rec\_aux* :  
 $\forall \Gamma'' T (t : \text{term } \Gamma'' T) \Gamma \Delta (Heq : \Gamma'' = \Gamma ; \Delta) \Gamma' U$   
 $(H : (\Gamma , U ; \Delta) = (\Gamma' , U ; \Delta)) (H' : \Gamma'' = (\Gamma' ; \Delta)),$   
 $\text{coerce\_term\_context } (\text{lift\_rec } \Gamma \Delta (\text{coerce\_term\_context } t \text{ Heq}) U) H =$   
 $\text{lift\_rec } \Gamma' \Delta (\text{coerce\_term\_context } t H') U.$

**Lemma** *coerce\_term\_context\_lift\_rec* :  
 $\forall \Gamma \Delta T (t : \text{term } (\Gamma ; \Delta) T) \Gamma' U (H : (\Gamma , U ; \Delta) = (\Gamma' , U ; \Delta))$   
 $(H' : (\Gamma ; \Delta) = (\Gamma' ; \Delta)),$   
 $\text{coerce\_term\_context } (\text{lift\_rec } \Gamma \Delta t U) H =$   
 $\text{lift\_rec } \Gamma' \Delta (\text{coerce\_term\_context } t H') U.$

We have to take care of the opaqueness of these constants all the time, spurious unfoldings can occur otherwise, greatly obfuscating the goal.

*Opaque coerce\_var\_context coerce\_term\_context.*  
*Opaque lift\_rec subst\_rec.*

We continue on our road to the substitution lemma. Here we prove commutation of *lift\_var*.

**Program Definition** *lift\_lift\_var\_stmt* :=  
 $\forall \Gamma' U (t : \text{var } \Gamma' U) \Gamma \Delta \Delta' (Heq : \Gamma' = \Gamma ; \Delta ; \Delta') T T',$   
 $\text{lift\_var } (\Gamma , T ; \Delta) \Delta' (\text{lift\_var } \Gamma (\Delta ; \Delta') t T) T' =$   
 $\text{lift\_var } \Gamma (\Delta , T' ; \Delta') (\text{lift\_var } (\Gamma ; \Delta) \Delta' t T') T.$

**Lemma** *lift\_lift\_var* : *lift\_lift\_var\_stmt*.

**Program Definition** *lift\_lift\_stmt* :=  
 $\forall \Gamma' U (t : \text{term } \Gamma' U) \Gamma \Delta \Delta' T T' (\text{Heq} : \Gamma' = \Gamma ; \Delta ; \Delta'),$   
 $\text{lift\_rec } (\Gamma , T ; \Delta) \Delta' (\text{lift\_rec } \Gamma (\Delta ; \Delta') t T) T' =$   
 $\text{lift\_rec } \Gamma (\Delta , T' ; \Delta') (\text{lift\_rec } (\Gamma ; \Delta) \Delta' t T') T.$

**Lemma** *lift\_lift* : *lift\_lift\_stmt*.

Commutation of lifting and substitution, for variables.

**Program Definition** *subst\_lift\_var\_comm\_stmt* :=  
 $\forall \Gamma' T (t : \text{var } \Gamma' T) \Gamma \Delta' \Delta U T' (s : \text{term } \Gamma U)$   
 $(\text{Heq} : \Gamma' = \Gamma , U ; \Delta ; \Delta'),$   
 $\text{subst\_var\_rec } (\Delta , T' ; \Delta') (\text{lift\_var } (\Gamma , U ; \Delta) \Delta' t T') s =$   
 $\text{lift\_rec } (\Gamma ; \Delta) \Delta' (\text{subst\_var\_rec } (\Delta ; \Delta') t s) T'.$

Commutation of lifting and substitution, for terms.

**Program Definition** *subst\_lift\_comm\_stmt* :=  
 $\forall \Gamma' T (t : \text{term } \Gamma' T) \Gamma U \Delta \Delta' T' (s : \text{term } \Gamma U)$   
 $(\text{Heq} : \Gamma' = \Gamma , U ; \Delta ; \Delta'),$   
 $\text{subst\_rec } (\Delta , T' ; \Delta') (\text{lift\_rec } (\Gamma , U ; \Delta) \Delta' t T') s =$   
 $\text{lift\_rec } (\Gamma ; \Delta) \Delta' (\text{subst\_rec } (\Delta ; \Delta') t s) T'.$

The non-recursive case, with no  $\Delta'$  context. This time the statement need not contain any coercions.

**Lemma** *subst\_lift\_comm\_full* :  
 $\forall \Gamma U \Delta T (t : \text{term } (\Gamma , U ; \Delta) T) (s : \text{term } \Gamma U) T',$   
 $\text{subst\_rec } (\Delta , T') (\text{lift\_rec } (\Gamma , U ; \Delta) [] t T') s =$   
 $\text{lift\_rec } (\Gamma ; \Delta) [] (\text{subst\_rec } \Delta t s) T'.$

Finally, we can state the substitution lemma as usual. The proof uses all the lemmas proven above, including commutation of *lift\_rec* and *coerce\_term\_context*.

**Program Definition** *subst\_rec\_comm\_stmt* :=  
 $\forall \Gamma' T (u : \text{term } \Gamma' T) \Delta' \Gamma \Delta T' T''$   
 $(s : \text{term } (\Gamma , T'' ; \Delta) T') (t : \text{term } \Gamma T'')$   
 $(\text{Heq} : \Gamma' = (\Gamma , T'' ; \Delta , T' ; \Delta')),$   
 $\text{subst\_rec } \Delta' (\text{subst\_rec } (\Delta , T' ; \Delta') u t) (\text{subst\_rec } \Delta s t) =$   
 $\text{subst\_rec } (\Delta ; \Delta') (\text{subst\_rec } \Delta' u s) t.$

**Lemma** *subst\_rec\_comm* : *subst\_rec\_comm\_stmt*.

Non-recursive case, where no coercion is needed anymore.

**Lemma** *subst\_comm* :  
 $\forall \Gamma T'' \Delta T' T (u : \text{term } (\Gamma , T'' ; \Delta , T') T)$   
 $(s : \text{term } (\Gamma , T'' ; \Delta) T') (t : \text{term } \Gamma T''),$   
 $\text{subst } (\text{subst\_rec } (\Delta , T') u t) (\text{subst\_rec } \Delta s t) =$

```
subst_rec Δ (subst u s) t.
```

End *Substitution*.

#### 4 Interpretation into CIC.

There is a direct embedding of  $\lambda^\rightarrow$  into the Calculus of Constructions, which we build now.

Section *Interpretation*.

We require interpretations for the constants and their types.

```
Variable interp_cst_type : constant_types → Set.
Variable interp_cst : ∀ c : constants, interp_cst_type (type_constant c).
```

Interpret a type into a coq **Set**. We need an impredicate **Set** to work with this definition.

```
Fixpoint interp_type (t : type) : Set :=
  match t with
  | type_cst c ⇒ interp_cst_type c
  | a → b ⇒ interp_type a → interp_type b
  end.
```

A valuation is a function which associates a term to each variable in a context.

```
Program Definition valuation (Γ : ctx) :=
  ∀ T, var Γ T → interp_type T.
```

We can extend a valuation by a term.

```
Program Definition augment_valuation (Γ : ctx) (rho : valuation Γ)
  (T : type) (x : interp_type T) : valuation (Γ , T) :=
  fun T v ⇒
  match v with
  | first _ _ ⇒ x
  | next Γ' T' v' _ ⇒ rho T' v'
  end.
```

Now we can interpret terms easily.

```
Fixpoint interp_term (Γ : ctx) (T : type) (t : term Γ T)
  (v : valuation Γ) {struct t} : interp_type T :=
  match t in term _ T return valuation Γ → interp_type T with
  | cst c ⇒ fun _ ⇒ interp_cst c
  | rel T n ⇒ fun v ⇒ v T n
  | lam T T' b ⇒ (fun v ⇒
    (fun x : interp_type T ⇒
      interp_term b (augment_valuation v x)))
  | app T T' f e ⇒ fun v ⇒ (interp_term f v) (interp_term e v)
```

end  $v$ .

The empty valuation.

**Program Definition**  $nil\_valuation : valuation [] := fun n v \Rightarrow !$ .

Interpretation of closed terms.

**Definition**  $interp\_closed\_term (T : type) (t : term [] T) : interp\_type T := interp\_term t nil\_valuation$ .

**End Interpretation.**

## 5 A proof of weak normalization.

Inspired by (Biernacka *et al.*, 2006), we build a proof of weak-head normalization using a tait-style proof. This is an instance of normalization by evaluation where the semantic domain is given by  $R$ , the strong computability predicate.

**Section Normalization.**

Beta-reduction and its contextual closure.

**Inductive**  $reduces (\Gamma : ctx) : \forall T : type, term \Gamma T \rightarrow term \Gamma T \rightarrow Prop :=$   
 $| red\_beta : \forall \tau \tau' (b : term (\Gamma, \tau) \tau') (e : term \Gamma \tau),$   
 $reduces (app (lam b) e) (subst b e)$   
 $| red\_app : \forall \tau \tau' f f', @reduces \Gamma (\tau \rightarrow \tau') f f' \rightarrow$   
 $\forall e, reduces (app f e) (app f' e).$

Evaluation gives the definition of normal forms, either:

- $eval\_trans$  If a term  $t$  reduces to  $s$  which itself evaluates to  $r$  then  $t$  evaluates to  $r$ . So evaluation is "backward-closed" by reduction.
- $eval\_lam$  Lambda expressions are all in normal form.
- $eval\_cst$  A constant is a normal form.

**Inductive**  $evaluates (\Gamma : ctx) : \forall T : type, term \Gamma T \rightarrow term \Gamma T \rightarrow Prop :=$   
 $| eval\_trans : \forall T (t : term \Gamma T) s r,$   
 $reduces t s \rightarrow evaluates s r \rightarrow evaluates t r$   
 $| eval\_lam : \forall \tau \tau' b, @evaluates \Gamma (\tau \rightarrow \tau') (lam b) (lam b)$   
 $| eval\_cst : \forall c, evaluates (cst \Gamma c) (cst \Gamma c).$

$R$  is the glueing model in the sense of Coquand et al. We have a syntactic part  $v$  which is the value of the term  $t$  and a semantic one given by the function in the arrow case which ensures closure by application. It is trivial for constants. Note that  $R$  lives in  $Set$  so that we can extract it later.

**Program Fixpoint**  $R \Gamma T (t : term \Gamma T) \{ struct T \} : Set :=$   
 $match T with$   
 $| type\_cst c \Rightarrow \{ v : term \Gamma T \mid evaluates t v \}$

```

|  $\tau \rightarrow \tau' \Rightarrow$ 
  ( $\{v : \text{term } \Gamma \ T \mid \text{evaluates } t \ v\} \times$ 
   ( $\forall s, R \ s \rightarrow R \ (\text{@app } \Gamma \ \tau \ \tau' \ t \ s)$ ))%type
end.

```

Extract the "syntactic" part.

**Lemma** *R\_eval* :  $\forall \Gamma \ T \ (t : \text{term } \Gamma \ T), R \ t \rightarrow \{v : \text{term } \Gamma \ T \mid \text{evaluates } t \ v\}$ .

Backward-closedness of evaluation extends to computability.

**Lemma** *red\_R\_R* :  $\forall \Gamma \ T \ (t \ s : \text{term } \Gamma \ T), \text{reduces } t \ s \rightarrow R \ s \rightarrow R \ t$ .

### 5.1 Telescopes.

A substitution is a telescope of terms.

```

Inductive substitution : ctx  $\rightarrow$  Set :=
| SNil : substitution []
| SCons :  $\forall \Gamma, \text{substitution } \Gamma \rightarrow \forall T, \text{term } \Gamma \ T \rightarrow \text{substitution } (\Gamma, T)$ .

```

Substitute a term by a telescope.

```

Program Fixpoint mult_subst  $\Gamma \ \Delta \ U \ (u : \text{term } (\Gamma ; \Delta) \ U)$ 
  ( $s : \text{substitution } \Gamma$ ) { struct s } : term  $\Delta \ U$  :=
match s with
| SNil  $\Rightarrow u$ 
| SCons  $\Gamma' \ s' \ T \ t \Rightarrow$ 
  mult_subst  $\Delta \ (\text{subst\_rec } \Delta \ u \ t) \ s'$ 
end.

```

This predicate ensures *R*-ness of all the terms in the telescope once they are substituted by previous terms in the telescope. It needs to be in **Set** as *R* is in **Set**.

```

Fixpoint substitution_R  $\Gamma \ (s : \text{substitution } \Gamma)$  { struct s } : Set :=
match s with
| SNil  $\Rightarrow \text{unit}$ 
| SCons  $\Gamma' \ s' \ T \ t \Rightarrow (R \ (\text{mult\_subst } [] \ t \ s') \times \text{substitution\_R } s')$ %type
end.

```

Inversion of non-empty substitutions.

**Lemma** *substitution\_cons\_aux* :  $\forall \Gamma \ (s : \text{substitution } \Gamma), \forall \Gamma' \ T, \Gamma = \Gamma', T \rightarrow \{(s', t) : \text{substitution } \Gamma' \times \text{term } \Gamma' \ T \mid \text{JMeq } s \ (\text{SCons } s' \ t)\}$ .

**Lemma** *substitution\_cons* :  $\forall \Gamma \ T \ (s : \text{substitution } (\Gamma, T)), \{(s', t) : \text{substitution } \Gamma \times \text{term } \Gamma \ T \mid \text{JMeq } s \ (\text{SCons } s' \ t)\}$ .

Substitution has no effect on constants.

**Lemma** *subst\_cst* :  $\forall \Gamma \ c \ (s : \text{substitution } \Gamma),$

$mult\_subst [] (cst \Gamma c) s = cst [] c.$

Unfolding of telescope substitution on lambdas.

**Lemma**  $mult\_subst\_lam : \forall \Gamma T T' (u : term (\Gamma, T) T') (s : substitution \Gamma),$   
 $mult\_subst [] (lam u) s = lam (mult\_subst ([], T) u s).$

Unfolding of telescope substitution on applications.

**Lemma**  $mult\_subst\_app : \forall \Gamma T T' (f : term \Gamma (T \rightarrow T'))$   
 $(e : term \Gamma T) (s : substitution \Gamma),$   
 $mult\_subst [] (app f e) s = app (mult\_subst [] f s) (mult\_subst [] e s).$

Commutation of multiple substitutions is needed to prove that  $R$  is substitutive. It is just an application of the substitution lemma proved previously.

**Lemma**  $mult\_subst\_comm : \forall \Gamma (s : substitution \Gamma) \Delta T' T$   
 $(u : term (\Gamma ; \Delta, T') T) (s0 : term (\Gamma ; \Delta) T'),$   
 $subst (mult\_subst (\Delta, T') u s) (mult\_subst \Delta s0 s) =$   
 $mult\_subst \Delta (subst u s0) s.$

We extend  $subst\_lift$  to contexts using substitution of telescopes and lifting by contexts.

**Program Definition**  $mult\_subst\_lift\_stmt :=$   
 $\forall \Delta (s : substitution \Delta) T (t : term [] T),$   
 $(mult\_subst [] (lift\_ctx \Delta t) s) = t.$

We need to have a dynamic coercion to prove these lemmas.

**Program Fixpoint**  $coerce\_subst\_context \Gamma (s : substitution \Gamma)$   
 $\Gamma' (Heq:\Gamma = \Gamma') \{struct s\} : substitution \Gamma' :=$   
 $match \Gamma' with$   
 $| [] \Rightarrow SNil$   
 $| \Gamma', T' \Rightarrow$   
 $match s with$   
 $| SNil \Rightarrow !$   
 $| SCons _ s' T t \Rightarrow$   
 $@SCons \Gamma' (coerce\_subst\_context s' _) T' t$   
 $end$   
 $end.$

Extension of the  $on\_coerce\_proof$  tactical.

**Ltac**  $on\_coerce\_proof tac :=$   
 $match goal with$   
 $[ | - ?T ] \Rightarrow$   
 $match T with$   
 $| context [ coerce\_var\_context _?p ] \Rightarrow tac p$   
 $| context [ coerce\_term\_context _?p ] \Rightarrow tac p$   
 $| context [ eq\_rect _?p ] \Rightarrow tac p$

```

      | context [ coerce_subst_context _?p ] => tac p
    end
  end.

```

The rest is just instantiation of tactics with the new *on\_coerce\_proof*.

We can finally prove the *mult\_subst* lemma.

**Lemma** *mult\_subst\_lift* : *mult\_subst\_lift\_stmt*.

## 5.2 Main lemma.

A little lemma on variables is all that is missing before proving our main lemma.

We can destruct variables, keeping the equality handy.

**Lemma** *var\_case\_aux* :  $\forall \Gamma' T (v : \text{var } \Gamma' T), \forall \Gamma t, \Gamma' = \Gamma, t \rightarrow$   
 $(t = T \wedge \text{JMeq } v (\text{first } \Gamma T)) + \{ v' : \text{var } \Gamma T \mid \text{JMeq } v (\text{next } v' t) \}.$

**Lemma** *var\_case* :  $\forall \Gamma t T (v : \text{var } (\Gamma, t) T),$   
 $(t = T \wedge \text{JMeq } v (\text{first } \Gamma T)) + \{ v' : \text{var } \Gamma T \mid \text{JMeq } v (\text{next } v' t) \}.$

We now prove the main lemma, substitutivity of *R*. Note that we end with a closed term, escaping the problem of variables.

**Lemma** *R\_substitutive* :  $\forall \Gamma U (u : \text{term } \Gamma U) (s : \text{substitution } \Gamma),$   
 $\text{substitution}_R s \rightarrow R (\text{mult\_subst } [] u s).$

We get weak normalization as a trivial consequence of the previous lemma and *R\_eval*.

**Lemma** *term\_wn* :  $\forall T (t : \text{term } [] T), \{ v : \text{term } [] T \mid \text{evaluates } t v \}.$

**End Normalization.**

**End  $\Lambda$ .**

## 6 Instantiation.

We will now instantiate  $\Lambda$  with the naturals and booleans. Our constants are naturals or booleans of *Coq*.

**Inductive** *cst\_types* : **Set** := *cst\_nat\_t* | *cst\_bool\_t*.

**Inductive** *csts* : **Set** :=

```

  | cst_nat : nat → csts
  | cst_bool : bool → csts.

```

**Definition** *type\_constants* (cst : *csts*) : *cst\_types* :=

```

  match cst with
  | cst_nat _ => cst_nat_t
  | cst_bool _ => cst_bool_t
  end.

```

Module *C*.

Definition *constant\_types* : Set := *cst\_types*.

Definition *constants* : Set := *csts*.

Definition *type\_constant* : *constants*  $\rightarrow$  *constant\_types* := *type\_constants*.

End *C*.

Module *LambdaC* :=  $\Lambda$  *C*. Import *LambdaC*.

We construct trivial interpretations of the naturals and booleans in *Coq*.

Definition *interp\_constant\_type* (*cst\_t* : *cst\_types*) : Set :=

```
match cst_t with
| cst_nat_t  $\Rightarrow$  nat
| cst_bool_t  $\Rightarrow$  bool
end.
```

Definition *interp\_constant* (*c* : *csts*) : *interp\_constant\_type* (*type\_constants* *c*) :=

```
match c return interp_constant_type (type_constants c) with
| cst_nat n  $\Rightarrow$  n
| cst_bool b  $\Rightarrow$  b
end.
```

Wrap interpretation using the previous functions.

Definition *interp\_nb\_type* := *interp\_type interp\_constant\_type*.

Definition *interp\_nb* := *interp\_term interp\_constant*.

Definition *interp\_closed\_nb* := *interp\_closed\_term interp\_constant\_type interp\_constant*. ■

### 6.1 An example term.

Definition *nat\_type* : type := *type\_cst cst\_nat\_t*.

The identity on naturals.

Program Definition *nat\_id\_term* : term [] (*nat\_type*  $\rightarrow$  *nat\_type*) :=  
lam (rel (first [] *nat\_type*)).

*Eval compute in* (*interp\_closed\_nb nat\_id\_term*).

*Eval compute in* (*interp\_type interp\_constant\_type* (*nat\_type*  $\rightarrow$  *nat\_type*)).

We can extract the term normalization function from this development.

*Extraction "nbe.ml" term\_wn*.

## References

Biernacka, Malgorzata, Danvy, Olivier, & Størring, Kristian. (2006). Program extraction from proofs of weak head normalization. *Electr. notes theor. comput. sci.*, **155**, 169–189.