

Strong Normalization for Church-Style System F

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

October 6, 2026

Contents

1 Overview

1

Abstract

This entry formalizes strong normalization for Church-style System F with explicit type abstraction and type application. Types and terms use de Bruijn indices; the reduction relation contains both term- β and type- β steps and is closed under all term contexts. The proof combines Girard-style reducibility candidates [5] for an untyped lambda-calculus erasure with an explicit type-syntax measure that reflects the erased normalization result back to the full calculus. The final theorem states that every well-typed System F term is strongly normalizing, with the so the usual closed-term result is an immediate instance.

1 Overview

System F extends the simply typed lambda calculus with universal types, polymorphic abstraction, and type instantiation. The formalization represents these constructs explicitly:

$$\text{TmTAbs } M \quad \text{and} \quad \text{TmTApp } M A.$$

Typing uses de Bruijn indices for both term and type binders. The relation $\longrightarrow_\beta$ includes ordinary term- β , type- β , and compatible-context rules.

The semantic part interprets each type as a reducibility candidate in the untyped lambda calculus. Universal types quantify over all candidates, so polymorphic instantiation is handled semantically rather than by collapsing all type variables to one simple type. Erasure removes only type syntax and annotations. Every full reduction step either produces an erased untyped- β step or leaves the erasure unchanged while decreasing the number of explicit type abstractions and applications. A lexicographic reflection argument therefore yields strong normalization for the original System F term.

The proof proceeds in three stages. First, reducibility candidates establish strong normalization of the untyped erasure of every well-typed term. Second, each System F reduction is shown either to induce an untyped β -step after erasure or to preserve the erasure while strictly decreasing the amount of explicit type syntax. Finally, a lexicographic argument reflects strong normalization of the erasure back to the Church-style calculus.

Earlier formalizations include Altenkirch’s LEGO development [1], the ATS/LF development of Donnelly and Xi [4], and the Isabelle/HOL development of Popescu, Gunter, and Osborn [6]. The latter formalization uses Curry-style terms and HOAS on top of FOAS. The present entry instead treats Church-style syntax directly: term and type binders use de Bruijn indices, type- β is an explicit reduction rule, and the erasure proof reflects normalization with an explicit type-syntax measure. This distinguishes the calculus, binding representation, and proof architecture without claiming novelty for the classical normalization theorem.

We also considered basing the development on Berghofer’s **StrongNorm** formalization in the Isabelle/HOL **HOL-Proofs-Lambda** library [2]. That development provides a closely related de Bruijn treatment of strong normalization for the simply typed lambda calculus, including lifting, substitution, subject reduction, and an inductive characterization of terminating terms. It has only term abstraction/application, simple arrow types, and term- β , however; it does not cover polymorphic type abstraction/application, impredicative universal types, or type- β reduction. The present entry therefore follows its de Bruijn and substitution discipline while developing a System-F-specific reducibility-candidate semantics and an erasure/reflection argument for explicit type reductions. Berghofer’s **POPLmark AFP** entry [3] is similarly valuable for the de Bruijn treatment of binding and substitution, but it formalizes the type-safety infrastructure for System $F_{<}$, not strong normalization. We use these developments as methodological references rather than claiming to extend either source by direct import.

The development also proves subject reduction independently of the normalization argument. It is included both as a useful metatheorem and as an independent validation of the substitution infrastructure; the normalization proof does not invoke preservation. Typed weakening and term/type substitution lemmas validate the de Bruijn operations at both term- β and type- β redexes, and the preservation theorem provides an independent consistency check on the operational semantics.

```
theory System-F
  imports Main
begin
```

```
datatype dtty =
  TyVar nat
| TyArr dtty dtty
```

```

| TyAll dtv

datatype dtm =
  TmVar nat
| TmApp dtm dtm
| TmLam dtv dtm
| TmTAbs dtm
| TmTApp dtm dtv

fun ty-shift :: nat ⇒ dtv ⇒ dtv where
  ty-shift k (TyVar n) =
    (if n < k then TyVar n else TyVar (Suc n))
| ty-shift k (TyArr A B) = TyArr (ty-shift k A) (ty-shift k B)
| ty-shift k (TyAll A) = TyAll (ty-shift (Suc k) A)

fun ty-subst :: nat ⇒ dtv ⇒ dtv ⇒ dtv where
  ty-subst k S (TyVar n) =
    (if n < k then TyVar n else
     if n = k then S else TyVar (n - 1))
| ty-subst k S (TyArr A B) =
  TyArr (ty-subst k S A) (ty-subst k S B)
| ty-subst k S (TyAll A) =
  TyAll (ty-subst (Suc k) (ty-shift 0 S) A)

definition ty-subst0 :: dtv ⇒ dtv ⇒ dtv
where
  ty-subst0 S A = ty-subst 0 S A

fun tm-shift :: nat ⇒ dtm ⇒ dtm where
  tm-shift k (TmVar n) =
    (if n < k then TmVar n else TmVar (Suc n))
| tm-shift k (TmApp M N) = TmApp (tm-shift k M) (tm-shift k N)
| tm-shift k (TmLam A M) = TmLam A (tm-shift (Suc k) M)
| tm-shift k (TmTAbs M) = TmTAbs (tm-shift k M)
| tm-shift k (TmTApp M A) = TmTApp (tm-shift k M) A

fun tm-ty-shift :: nat ⇒ dtm ⇒ dtm where
  tm-ty-shift k (TmVar n) = TmVar n
| tm-ty-shift k (TmApp M P) =
  TmApp (tm-ty-shift k M) (tm-ty-shift k P)
| tm-ty-shift k (TmLam A M) =
  TmLam (ty-shift k A) (tm-ty-shift k M)
| tm-ty-shift k (TmTAbs M) =
  TmTAbs (tm-ty-shift (Suc k) M)
| tm-ty-shift k (TmTApp M A) =
  TmTApp (tm-ty-shift k M) (ty-shift k A)

fun tm-subst :: nat ⇒ dtm ⇒ dtm ⇒ dtm where
  tm-subst k N (TmVar n) =

```

```

      (if  $n < k$  then  $TmVar\ n$  else
       if  $n = k$  then  $N$  else  $TmVar\ (n - 1)$ )
|  $tm\_subst\ k\ N\ (TmApp\ M\ P) =$ 
   $TmApp\ (tm\_subst\ k\ N\ M)\ (tm\_subst\ k\ N\ P)$ 
|  $tm\_subst\ k\ N\ (TmLam\ A\ M) =$ 
   $TmLam\ A\ (tm\_subst\ (Suc\ k)\ (tm\_shift\ 0\ N)\ M)$ 
|  $tm\_subst\ k\ N\ (TmTAbs\ M) =$ 
   $TmTAbs\ (tm\_subst\ k\ (tm\_ty\_shift\ 0\ N)\ M)$ 
|  $tm\_subst\ k\ N\ (TmTApp\ M\ A) =$ 
   $TmTApp\ (tm\_subst\ k\ N\ M)\ A$ 

```

definition $tm_subst0 :: dtm \Rightarrow dtm \Rightarrow dtm$

where

$tm_subst0\ N\ M = tm_subst\ 0\ N\ M$

fun $tm_ty_subst :: nat \Rightarrow dty \Rightarrow dtm \Rightarrow dtm$ **where**

```

   $tm\_ty\_subst\ k\ S\ (TmVar\ n) = TmVar\ n$ 
|  $tm\_ty\_subst\ k\ S\ (TmApp\ M\ N) =$ 
   $TmApp\ (tm\_ty\_subst\ k\ S\ M)\ (tm\_ty\_subst\ k\ S\ N)$ 
|  $tm\_ty\_subst\ k\ S\ (TmLam\ A\ M) =$ 
   $TmLam\ (ty\_subst\ k\ S\ A)\ (tm\_ty\_subst\ k\ S\ M)$ 
|  $tm\_ty\_subst\ k\ S\ (TmTAbs\ M) =$ 
   $TmTAbs\ (tm\_ty\_subst\ (Suc\ k)\ (ty\_shift\ 0\ S)\ M)$ 
|  $tm\_ty\_subst\ k\ S\ (TmTApp\ M\ A) =$ 
   $TmTApp\ (tm\_ty\_subst\ k\ S\ M)\ (ty\_subst\ k\ S\ A)$ 

```

definition $tm_ty_subst0 :: dty \Rightarrow dtm \Rightarrow dtm$

where

$tm_ty_subst0\ S\ M = tm_ty_subst\ 0\ S\ M$

definition $ty_env_cons :: 'a \Rightarrow (nat \Rightarrow 'a) \Rightarrow nat \Rightarrow 'a$

where

$ty_env_cons\ C\ \varrho\ n = (case\ n\ of\ 0 \Rightarrow C \mid Suc\ k \Rightarrow \varrho\ k)$

definition $tm_env_cons :: dtm \Rightarrow (nat \Rightarrow dtm) \Rightarrow nat \Rightarrow dtm$

where

$tm_env_cons\ N\ \delta\ n = (case\ n\ of\ 0 \Rightarrow N \mid Suc\ k \Rightarrow \delta\ k)$

definition $tm_env_shift :: (nat \Rightarrow dtm) \Rightarrow nat \Rightarrow dtm$

where

$tm_env_shift\ \delta\ n =$
 $(case\ n\ of\ 0 \Rightarrow TmVar\ 0 \mid Suc\ k \Rightarrow tm_shift\ 0\ (\delta\ k))$

fun $tm_subst_env :: (nat \Rightarrow dtm) \Rightarrow dtm \Rightarrow dtm$ **where**

```

   $tm\_subst\_env\ \delta\ (TmVar\ k) = \delta\ k$ 
|  $tm\_subst\_env\ \delta\ (TmApp\ M\ N) =$ 
   $TmApp\ (tm\_subst\_env\ \delta\ M)\ (tm\_subst\_env\ \delta\ N)$ 
|  $tm\_subst\_env\ \delta\ (TmLam\ A\ M) =$ 
   $TmLam\ A\ (tm\_subst\_env\ (tm\_env\_shift\ \delta)\ M)$ 

```

| $tm\text{-}subst\text{-}env\ \delta\ (TmTAbs\ M) = TmTAbs\ (tm\text{-}subst\text{-}env\ \delta\ M)$
| $tm\text{-}subst\text{-}env\ \delta\ (TmTApp\ M\ A) =$
 $TmTApp\ (tm\text{-}subst\text{-}env\ \delta\ M)\ A$

inductive $wf\text{-}ty :: nat \Rightarrow dtty \Rightarrow bool$ **where**
 $wf\text{-}var: n < k \implies wf\text{-}ty\ k\ (TyVar\ n)$
| $wf\text{-}arr: \llbracket wf\text{-}ty\ k\ A; wf\text{-}ty\ k\ B \rrbracket \implies$
 $wf\text{-}ty\ k\ (TyArr\ A\ B)$
| $wf\text{-}all: wf\text{-}ty\ (Suc\ k)\ A \implies wf\text{-}ty\ k\ (TyAll\ A)$

inductive $ctx\text{-}mem :: dtty\ list \Rightarrow nat \Rightarrow dtty \Rightarrow bool$ **where**
 $ctx\text{-}zero: ctx\text{-}mem\ (A \# \Gamma)\ 0\ A$
| $ctx\text{-}suc: ctx\text{-}mem\ \Gamma\ k\ A \implies ctx\text{-}mem\ (B \# \Gamma)\ (Suc\ k)\ A$

inductive $typing :: nat \Rightarrow dtty\ list \Rightarrow dtm \Rightarrow dtty \Rightarrow bool$
 $(\langle - \rangle; - \vdash - : \rightarrow [60,60,60,60]\ 60)$ **where**
 $ty\text{-}var: ctx\text{-}mem\ \Gamma\ n\ A \implies typing\ k\ \Gamma\ (TmVar\ n)\ A$
| $ty\text{-}app: \llbracket typing\ k\ \Gamma\ M\ (TyArr\ A\ B); typing\ k\ \Gamma\ N\ A \rrbracket$
 $\implies typing\ k\ \Gamma\ (TmApp\ M\ N)\ B$
| $ty\text{-}lam: \llbracket wf\text{-}ty\ k\ A; typing\ k\ (A \# \Gamma)\ M\ B \rrbracket$
 $\implies typing\ k\ \Gamma\ (TmLam\ A\ M)\ (TyArr\ A\ B)$
| $ty\text{-}tabs: typing\ (Suc\ k)\ (map\ (ty\text{-}shift\ 0)\ \Gamma)\ M\ B$
 $\implies typing\ k\ \Gamma\ (TmTAbs\ M)\ (TyAll\ B)$
| $ty\text{-}tapp: \llbracket typing\ k\ \Gamma\ M\ (TyAll\ B); wf\text{-}ty\ k\ A \rrbracket$
 $\implies typing\ k\ \Gamma\ (TmTApp\ M\ A)\ (ty\text{-}subst0\ A\ B)$

inductive $beta :: dtm \Rightarrow dtm \Rightarrow bool$
 $(\langle - \rangle \rightarrow_\beta \rightarrow [80,80]\ 80)$ **where**
 $beta\text{-}appL: M \rightarrow_\beta M' \implies$
 $TmApp\ M\ N \rightarrow_\beta TmApp\ M'\ N$
| $beta\text{-}appR: N \rightarrow_\beta N' \implies$
 $TmApp\ M\ N \rightarrow_\beta TmApp\ M\ N'$
| $beta\text{-}lam: M \rightarrow_\beta M' \implies$
 $TmLam\ A\ M \rightarrow_\beta TmLam\ A\ M'$
| $beta\text{-}tabs: M \rightarrow_\beta M' \implies$
 $TmTAbs\ M \rightarrow_\beta TmTAbs\ M'$
| $beta\text{-}tapp: M \rightarrow_\beta M' \implies$
 $TmTApp\ M\ A \rightarrow_\beta TmTApp\ M'\ A$
| $beta\text{-}term: TmApp\ (TmLam\ A\ M)\ N \rightarrow_\beta tm\text{-}subst0\ N\ M$
| $beta\text{-}type: TmTApp\ (TmTAbs\ M)\ A \rightarrow_\beta tm\text{-}ty\text{-}subst0\ A\ M$

inductive $SN :: dtm \Rightarrow bool$ **where**
 $SN\text{-}intro: (\bigwedge M'. M \rightarrow_\beta M' \implies SN\ M') \implies SN\ M$

end
theory *System-F-Subject-Reduction*
imports *System-F*
begin

This theory records the independent type-preservation check for the de

Bruijn presentation. It is not needed by the normalization argument, but it validates the substitution operations used by both beta rules.

```
fun ctx-insert :: nat  $\Rightarrow$  dty  $\Rightarrow$  dty list  $\Rightarrow$  dty list where
  ctx-insert i B [] = [B]
| ctx-insert 0 B (A #  $\Gamma$ ) = B # A #  $\Gamma$ 
| ctx-insert (Suc i) B (A #  $\Gamma$ ) = A # ctx-insert i B  $\Gamma$ 
```

```
lemma ctx-insert-Suc-cons:
  ctx-insert (Suc i) B (A #  $\Gamma$ ) = A # ctx-insert i B  $\Gamma$ 
by simp
```

```
lemma map-ctx-insert:
  map f (ctx-insert i B  $\Gamma$ ) = ctx-insert i (f B) (map f  $\Gamma$ )
proof (induct  $\Gamma$  arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons A  $\Gamma$ )
  then show ?case by (cases i) simp-all
qed
```

```
lemma ctx-mem-ctx-insert:
  assumes ctx-mem  $\Gamma$  n A
  shows ctx-mem (ctx-insert i B  $\Gamma$ )
    (if n < i then n else Suc n) A
using assms
proof (induct arbitrary: i B rule: ctx-mem.induct)
  case ctx-zero
  then show ?case by (cases i) (auto intro: ctx-mem.ctx-zero ctx-mem.ctx-suc)
next
  case (ctx-suc  $\Gamma$  n A C)
  then show ?case
  proof (cases i)
    case 0
    then show ?thesis
      using ctx-suc(1) by (auto intro: ctx-mem.ctx-suc)
  next
    case (Suc j)
    have ih:
      ctx-mem (ctx-insert j B  $\Gamma$ )
        (if n < j then n else Suc n) A
      using ctx-suc(2)[of j B] .
    from ih Suc show ?thesis
      by (cases n < j) (auto intro: ctx-mem.ctx-suc)
  qed
qed
```

```
lemma ctx-mem-sucD:
  ctx-mem (B #  $\Gamma$ ) (Suc n) A  $\implies$  ctx-mem  $\Gamma$  n A
```

```

by (erule ctx-mem.cases) simp-all

lemma ctx-mem-insert-cases:
  assumes ctx-mem (ctx-insert i C  $\Gamma$ ) n A  $i \leq \text{length } \Gamma$ 
  shows  $(n < i \wedge \text{ctx-mem } \Gamma \ n \ A) \vee$ 
     $(n = i \wedge A = C) \vee$ 
     $(i < n \wedge \text{ctx-mem } \Gamma \ (n - 1) \ A)$ 
using assms
proof (induct  $\Gamma$  arbitrary: i n C A)
  case Nil
  then show ?case
    by (cases i; cases n; auto elim: ctx-mem.cases)
next
  case (Cons D  $\Gamma$ )
  then show ?case
  proof (cases i)
    case 0
    have zero:  $i = 0$  using 0 .
    have mem0: ctx-mem (C # D #  $\Gamma$ ) n A
      using Cons.prem1 zero by simp
    then show ?thesis
      using mem0 zero
      by (cases rule: ctx-mem.cases)
      (auto intro: ctx-mem.ctx-zero ctx-mem.ctx-suc)
  next
    case (Suc j)
    have isuc:  $i = \text{Suc } j$  using Suc .
    show ?thesis
    proof (cases n)
      case 0
      have nzero:  $n = 0$  using 0 .
      have mem0: ctx-mem (D # ctx-insert j C  $\Gamma$ ) 0 A
        using Cons.prem1 isuc nzero by simp
      then show ?thesis
        using mem0 isuc nzero
        by (cases rule: ctx-mem.cases)
        (auto intro: ctx-mem.ctx-zero ctx-mem.ctx-suc)
    next
      case (Suc m)
      have nsuc:  $n = \text{Suc } m$  using Suc .
      have source:
        ctx-mem (D # ctx-insert j C  $\Gamma$ ) (Suc m) A
        using Cons.prem1 isuc nsuc by simp
      have mem:
        ctx-mem (ctx-insert j C  $\Gamma$ ) m A
        using ctx-mem-suc[OF source] .
      have len:  $j \leq \text{length } \Gamma$ 
        using Cons.prem2 isuc by simp
      have ih:

```

```

    (m < j ∧ ctx-mem Γ m A) ∨
    (m = j ∧ A = C) ∨
    (j < m ∧ ctx-mem Γ (m - 1) A)
  using Cons.hyps[of j C m A] mem len .
have lift:
  j < m ⇒ ctx-mem Γ (m - 1) A ⇒
  ctx-mem (D # Γ) m A
proof -
  assume jm: j < m
  assume mem': ctx-mem Γ (m - 1) A
  have mpos: 0 < m using jm by arith
  have eq: m = Suc (m - 1) using Suc-pred'[OF mpos] .
  from ctx-mem.ctx-suc[OF mem'] show ?thesis
    using eq by simp
qed
show ?thesis
  using ih isuc nsuc
  by (auto simp add: ctx-insert.simps
    intro: lift
    intro: ctx-mem.ctx-zero ctx-mem.ctx-suc)
qed
qed
qed

lemma typing-weakening:
  typing k Γ M A ⇒
  typing k (ctx-insert i B Γ) (tm-shift i M) A
proof (induct arbitrary: i B rule: typing.induct)
  case (ty-var Γ n A k)
  then show ?case
  proof (cases n < i)
    case True
    have mem:
      ctx-mem (ctx-insert i B Γ)
      (if n < i then n else Suc n) A
    using ctx-mem.ctx-insert[OF ty-var(1)] .
    then show ?thesis
    using True by (auto intro: typing.ty-var)
  next
    case False
    have mem:
      ctx-mem (ctx-insert i B Γ)
      (if n < i then n else Suc n) A
    using ctx-mem.ctx-insert[OF ty-var(1)] .
    then show ?thesis
    using False by (auto intro: typing.ty-var)
  qed
next
  case ty-app

```

```

then show ?case
  unfolding tm-shift.simps
  by (blast intro: typing.ty-app)
next
case (ty-lam k0 A0  $\Gamma$ 0 M0 B0)
show ?case
  unfolding tm-shift.simps
proof (rule typing.ty-lam)
  show wf-ty k0 A0 using ty-lam(1) .
  show typing k0 (A0 # ctx-insert i B  $\Gamma$ 0)
    (tm-shift (Suc i) M0) B0
    using ty-lam(3)[of Suc i B]
    by (simp add: ctx-insert-Suc-cons)
qed
next
case (ty-tabs k0  $\Gamma$ 0 M0 B0)
show ?case
  unfolding tm-shift.simps
proof (rule typing.ty-tabs)
  have ih:
    typing (Suc k0)
      (ctx-insert i (ty-shift 0 B) (map (ty-shift 0)  $\Gamma$ 0))
      (tm-shift i M0) B0
    using ty-tabs(2)[of i ty-shift 0 B] .
  then show typing (Suc k0)
    (map (ty-shift 0) (ctx-insert i B  $\Gamma$ 0))
    (tm-shift i M0) B0
    by (simp add: map-ctx-insert)
qed
next
case (ty-tapp k0  $\Gamma$ 0 M0 B0 A0)
show ?case
  unfolding tm-shift.simps
proof (rule typing.ty-tapp)
  show typing k0 (ctx-insert i B  $\Gamma$ 0) (tm-shift i M0)
    (TyAll B0)
    using ty-tapp(2)[of i B] .
  show wf-ty k0 A0 using ty-tapp(3) .
qed
qed

lemma wf-ty-shift-at:
  assumes wf-ty k A l  $\leq$  k
  shows wf-ty (Suc k) (ty-shift l A)
  using assms
proof (induct arbitrary: l rule: wf-ty.induct)
  case (wf-var n k l)
  then show ?case
  proof (cases n < l)

```

```

    case True
    have n < Suc k
      using wf-var.hyps wf-var.prem by arith
    have wf: wf-ty (Suc k) (TyVar n)
      by (rule wf-ty.wf-var) fact
    then show ?thesis
      using wf by (simp add: ty-shift.simps True)
  next
    case False
    have Suc n < Suc k
      using wf-var.hyps by simp
    have wf: wf-ty (Suc k) (TyVar (Suc n))
      by (rule wf-ty.wf-var) fact
    then show ?thesis
      using wf by (simp add: ty-shift.simps False)
  qed
next
  case (wf-arr k A B l)
  then show ?case
    by (auto intro: wf-ty.wf-arr)
next
  case (wf-all k A l)
  then show ?case
    by (auto intro: wf-ty.wf-all)
qed

lemma wf-ty-shift:
  assumes wf-ty k A
  shows wf-ty (Suc k) (ty-shift 0 A)
  using wf-ty-shift-at[OF assms, of 0] by simp

lemma ctx-mem-map:
  assumes ctx-mem  $\Gamma$  n A
  shows ctx-mem (map f  $\Gamma$ ) n (f A)
  using assms
  by (induct rule: ctx-mem.induct)
    (auto intro: ctx-mem.ctx-zero ctx-mem.ctx-suc)

lemma ty-shift-commute:
  assumes  $i \leq j$ 
  shows ty-shift i (ty-shift j A) =
    ty-shift (Suc j) (ty-shift i A)
  using assms
proof (induct A arbitrary: i j)
  case (TyVar n i j)
  then show ?case
    by (cases n < i; cases n < j; simp-all; arith)
next
  case (TyArr A B i j)

```

```

    then show ?case by simp
next
  case (TyAll A i j)
  then show ?case
    by (simp-all add: ty-shift.simps)
qed

lemma ty-subst-shift:
  assumes  $l \leq k$ 
  shows  $ty\_subst (Suc\ k) (ty\_shift\ l\ S) (ty\_shift\ l\ A) =$ 
     $ty\_shift\ l (ty\_subst\ k\ S\ A)$ 
  using assms
proof (induct A arbitrary: k l S)
  case (TyVar n k l S)
  then show ?case
    by (cases n < l; cases n < k; cases n = k;
        simp-all split: if-splits; arith)
next
  case (TyArr A B k l S)
  then show ?case by simp
next
  case (TyAll A k l S)
  then show ?case
    by (simp-all add: ty-shift-commute)
qed

lemma ty-subst-ty-shift:
   $ty\_subst (Suc\ k) (ty\_shift\ 0\ S) (ty\_shift\ 0\ A) =$ 
     $ty\_shift\ 0 (ty\_subst\ k\ S\ A)$ 
  using ty-subst-shift[of 0 k S A] by simp

lemma ty-subst-shift-suc:
  assumes  $k \leq l$ 
  shows  $ty\_subst\ k (ty\_shift\ l\ S) (ty\_shift (Suc\ l)\ A) =$ 
     $ty\_shift\ l (ty\_subst\ k\ S\ A)$ 
  using assms
proof (induct A arbitrary: k l S)
  case (TyVar n k l S)
  then show ?case
    by (cases n < l; cases n < Suc l; cases n < k;
        cases n = k; simp-all split: if-splits; arith)
next
  case (TyArr A B k l S)
  then show ?case by simp
next
  case (TyAll A k l S)
  then show ?case
    by (simp-all add: ty-shift-commute)
qed

```

```

lemma typing-ty-shift-at:
  assumes typing  $k \ \Gamma \ M \ A \ l \leq k$ 
  shows typing (Suc  $k$ ) (map (ty-shift  $l$ )  $\Gamma$ )
    (tm-ty-shift  $l \ M$ ) (ty-shift  $l \ A$ )
  using assms
proof (induct arbitrary: l rule: typing.induct)
  case (ty-var  $\Gamma \ n \ A \ k \ l$ )
  then show ?case
    by (auto intro: typing.ty-var ctx-mem-map)
next
  case (ty-app  $k0 \ \Gamma0 \ M0 \ A0 \ B0 \ N0 \ l$ )
  have M:
    typing (Suc  $k0$ ) (map (ty-shift  $l$ )  $\Gamma0$ )
      (tm-ty-shift  $l \ M0$ ) (TyArr (ty-shift  $l \ A0$ ) (ty-shift  $l \ B0$ ))
    using ty-app(2)[OF ty-app(5)] by (simp add: ty-shift.simps)
  have N:
    typing (Suc  $k0$ ) (map (ty-shift  $l$ )  $\Gamma0$ )
      (tm-ty-shift  $l \ N0$ ) (ty-shift  $l \ A0$ )
    using ty-app(4)[OF ty-app(5)] by simp
  show ?case
    unfolding tm-ty-shift.simps
    using typing.ty-app[OF M N] by simp
next
  case (ty-lam  $k0 \ A0 \ \Gamma0 \ M0 \ B0 \ l$ )
  have wf:
    wf-ty (Suc  $k0$ ) (ty-shift  $l \ A0$ )
    using wf-ty-shift-at[OF ty-lam(1) ty-lam(4)] by simp
  have body:
    typing (Suc  $k0$ ) (map (ty-shift  $l$ ) ( $A0 \ \# \ \Gamma0$ ))
      (tm-ty-shift  $l \ M0$ ) (ty-shift  $l \ B0$ )
    using ty-lam(3)[OF ty-lam(4)] by simp
  then show ?case
    unfolding tm-ty-shift.simps
    using wf by (simp add: typing.ty-lam)
next
  case (ty-tabs  $k0 \ \Gamma0 \ M0 \ B0 \ l$ )
  have le: Suc  $l \leq \text{Suc } k0$ 
  using ty-tabs(3) by simp
  have body:
    typing (Suc (Suc  $k0$ ))
      (map (ty-shift (Suc  $l$ )) (map (ty-shift 0)  $\Gamma0$ ))
      (tm-ty-shift (Suc  $l$ )  $M0$ ) (ty-shift (Suc  $l$ )  $B0$ )
    using ty-tabs(2)[of Suc l, OF le] by simp
  have shift-comp:
    (ty-shift (Suc  $l$ )  $\circ$  ty-shift 0) =
      (ty-shift 0  $\circ$  ty-shift  $l$ )
    by (rule ext; simp add: ty-shift-commute)
  have body':

```

```

    typing (Suc (Suc k0))
      (map (ty-shift 0) (map (ty-shift l)  $\Gamma$ 0))
      (tm-ty-shift (Suc l) M0) (ty-shift (Suc l) B0)
  using body by (simp add: map-map shift-comp)
have tabs:
  typing (Suc k0) (map (ty-shift l)  $\Gamma$ 0)
    (TmTAbs (tm-ty-shift (Suc l) M0))
    (TyAll (ty-shift (Suc l) B0))
  using typing.ty-tabs[OF body^] by simp
then show ?case
  by (simp add: tm-ty-shift.simps ty-shift.simps)
next
case (ty-tapp k0  $\Gamma$ 0 M0 B0 A0 l)
have M:
  typing (Suc k0) (map (ty-shift l)  $\Gamma$ 0)
    (tm-ty-shift l M0) (TyAll (ty-shift (Suc l) B0))
  using ty-tapp(2)[OF ty-tapp(4)] by simp
have W:
  wf-ty (Suc k0) (ty-shift l A0)
  using wf-ty-shift-at[OF ty-tapp(3) ty-tapp(4)] by simp
have inst:
  ty-subst0 (ty-shift l A0) (ty-shift (Suc l) B0) =
    ty-shift l (ty-subst0 A0 B0)
  unfolding ty-subst0-def
  using ty-subst-shift-suc[of 0 l A0 B0] by simp
have tapp:
  typing (Suc k0) (map (ty-shift l)  $\Gamma$ 0)
    (TmTApp (tm-ty-shift l M0) (ty-shift l A0))
    (ty-subst0 (ty-shift l A0) (ty-shift (Suc l) B0))
  using typing.ty-tapp[OF M W] .
then show ?case
  using inst by (simp add: tm-ty-shift.simps)
qed

```

lemma *typing-ty-shift*:

```

  typing k  $\Gamma$  M A  $\implies$ 
    typing (Suc k) (map (ty-shift 0)  $\Gamma$ )
      (tm-ty-shift 0 M) (ty-shift 0 A)
  using typing-ty-shift-at[of k  $\Gamma$  M A 0] by simp

```

lemma *wf-ty-subst-pred*:

```

  assumes wf-ty n A wf-ty (n - 1) S j < n
  shows wf-ty (n - 1) (ty-subst j S A)
  using assms
proof (induct arbitrary: S j rule: wf-ty.induct)
  case wf-var
  then show ?case
  by (auto split: if-splits
    intro: wf-ty.wf-var)

```

```

next
  case wf-arr
  then show ?case
    by (auto intro: wf-ty.wf-arr)
next
  case (wf-all k0 A0 S0 j0)
  have kpos: 0 < k0
    using wf-all(4) by arith
  have shiftS: wf-ty k0 (ty-shift 0 S0)
    using wf-ty-shift[OF wf-all(3)] by (simp add: kpos)
  have inner: wf-ty k0
    (ty-subst (Suc j0) (ty-shift 0 S0) A0)
    using wf-all(2)[of ty-shift 0 S0 Suc j0] shiftS wf-all(4)
    by simp
  then show ?case
    unfolding ty-subst.simps
    using inner
    by (auto intro: wf-ty.wf-all simp add: kpos)
qed

```

```

lemma wf-ty-subst-at:
  assumes wf-ty (Suc k) A wf-ty k S j ≤ k
  shows wf-ty k (ty-subst j S A)
  using wf-ty-subst-pred[of Suc k A S j] assms by simp

```

```

lemma ty-subst-shift-id:
  ty-subst k T (ty-shift k A) = A
proof (induct A arbitrary: k T)
  case (TyVar n k T)
  then show ?case
    by (cases n < k; cases n = k;
        simp-all add: ty-subst.simps ty-shift.simps split: if-splits; arith)
next
  case (TyArr A1 A2 k T)
  show ?case
    unfolding ty-subst.simps ty-shift.simps
    using TyArr(1)[of k T] TyArr(2)[of k T] by simp
next
  case (TyAll A k T)
  show ?case
    unfolding ty-subst.simps ty-shift.simps
    using TyAll(1)[of Suc k ty-shift 0 T] by simp
qed

```

```

lemma ty-subst-comp-at:
  i ≤ j ⟶
    ty-subst i (ty-subst j S T)
    (ty-subst (Suc j) (ty-shift i S) A) =
    ty-subst j S (ty-subst i T A)

```

```

proof (induct A arbitrary: i j S T rule: dty.induct)
  case (TyVar x)
  then show ?case
    by (intro impI; simp add: ty-subst-shift-id split: if-splits; arith)
next
  case (TyArr A1 A2)
  then show ?case
    by (intro impI; simp-all; blast)
next
  case (TyAll A)
  have result:
     $i \leq j \longrightarrow$ 
    
$$\begin{aligned}
& \text{ty-subst } (Suc\ i) \ (\text{ty-shift } 0 \ (\text{ty-subst } j\ S\ T)) \\
& \quad (\text{ty-subst } (Suc\ (Suc\ j)) \\
& \quad \quad (\text{ty-shift } 0 \ (\text{ty-shift } i\ S))\ A) = \\
& \text{ty-subst } (Suc\ j) \ (\text{ty-shift } 0\ S) \\
& \quad (\text{ty-subst } (Suc\ i) \ (\text{ty-shift } 0\ T)\ A)
\end{aligned}$$

  proof (intro impI)
  assume le:  $i \leq j$ 
  have ih:
    
$$\begin{aligned}
& \text{ty-subst } (Suc\ i) \\
& \quad (\text{ty-subst } (Suc\ j) \ (\text{ty-shift } 0\ S) \ (\text{ty-shift } 0\ T)) \\
& \quad (\text{ty-subst } (Suc\ (Suc\ j)) \\
& \quad \quad (\text{ty-shift } (Suc\ i) \ (\text{ty-shift } 0\ S))\ A) = \\
& \text{ty-subst } (Suc\ j) \ (\text{ty-shift } 0\ S) \\
& \quad (\text{ty-subst } (Suc\ i) \ (\text{ty-shift } 0\ T)\ A)
\end{aligned}$$

  using TyAll(1)[of Suc i Suc j ty-shift 0 S ty-shift 0 T]
  using le by simp
  have comp:
    
$$\begin{aligned}
& \text{ty-subst } (Suc\ j) \ (\text{ty-shift } 0\ S) \ (\text{ty-shift } 0\ T) = \\
& \text{ty-shift } 0 \ (\text{ty-subst } j\ S\ T) \\
& \text{using } \text{ty-subst-shift}[of\ 0\ j\ S\ T] \text{ by } simp
\end{aligned}$$

  have shift:
    
$$\begin{aligned}
& \text{ty-shift } (Suc\ i) \ (\text{ty-shift } 0\ S) = \\
& \text{ty-shift } 0 \ (\text{ty-shift } i\ S) \\
& \text{using } \text{ty-shift-commute}[of\ 0\ i\ S] \text{ by } simp
\end{aligned}$$

  have ih':
    
$$\begin{aligned}
& \text{ty-subst } (Suc\ i) \\
& \quad (\text{ty-subst } (Suc\ j) \ (\text{ty-shift } 0\ S) \ (\text{ty-shift } 0\ T)) \\
& \quad (\text{ty-subst } (Suc\ (Suc\ j)) \\
& \quad \quad (\text{ty-shift } 0 \ (\text{ty-shift } i\ S))\ A) = \\
& \text{ty-subst } (Suc\ j) \ (\text{ty-shift } 0\ S) \\
& \quad (\text{ty-subst } (Suc\ i) \ (\text{ty-shift } 0\ T)\ A)
\end{aligned}$$

  using ih by (simp only: shift[symmetric])
  have outer:
    
$$\begin{aligned}
& \text{ty-subst } (Suc\ i) \ (\text{ty-shift } 0 \ (\text{ty-subst } j\ S\ T)) \\
& \quad (\text{ty-subst } (Suc\ (Suc\ j)) \\
& \quad \quad (\text{ty-shift } 0 \ (\text{ty-shift } i\ S))\ A) = \\
& \text{ty-subst } (Suc\ i)
\end{aligned}$$


```

```

      (ty-subst (Suc j) (ty-shift 0 S) (ty-shift 0 T))
      (ty-subst (Suc (Suc j))
        (ty-shift 0 (ty-shift i S)) A)
    by (rule arg-cong[OF comp[symmetric]])
  from outer ih' show
    ty-subst (Suc i) (ty-shift 0 (ty-subst j S T))
      (ty-subst (Suc (Suc j))
        (ty-shift 0 (ty-shift i S)) A) =
    ty-subst (Suc j) (ty-shift 0 S)
      (ty-subst (Suc i) (ty-shift 0 T) A)
  by simp
qed
then show ?case by simp
qed

lemma ty-subst-comp:
  ty-subst j S (ty-subst 0 T A) =
  ty-subst 0 (ty-subst j S T)
    (ty-subst (Suc j) (ty-shift 0 S) A)
  using ty-subst-comp-at[of 0 j S T A] by simp

lemma typing-ty-subst-at-aux:
  assumes typing n  $\Gamma$  M A n = Suc k wf-ty k S j  $\leq$  k
  shows typing k (map (ty-subst j S)  $\Gamma$ )
    (tm-ty-subst j S M) (ty-subst j S A)
  using assms
proof (induct arbitrary: k j S rule: typing.induct)
  case (ty-var  $\Gamma$  n0 A0 k0 k j S)
  then show ?case
    by (auto intro: typing.ty-var ctx-mem-map)
next
  case (ty-app k0  $\Gamma$ 0 M0 A0 B0 N0 k j S)
  have M:
    typing k (map (ty-subst j S)  $\Gamma$ 0)
      (tm-ty-subst j S M0)
      (TyArr (ty-subst j S A0) (ty-subst j S B0))
    using ty-app(2)[OF ty-app(5) ty-app(6) ty-app(7)] by simp
  have N:
    typing k (map (ty-subst j S)  $\Gamma$ 0)
      (tm-ty-subst j S N0) (ty-subst j S A0)
    using ty-app(4)[OF ty-app(5) ty-app(6) ty-app(7)] by simp
  show ?case
    unfolding tm-ty-subst.simps
    using typing.ty-app[OF M N] by simp
next
  case (ty-lam k0 A0  $\Gamma$ 0 M0 B0 k j S)
  have hA: wf-ty (Suc k) A0
    using ty-lam(1) ty-lam(4) by simp
  have wf:

```

```

wf-ty k (ty-subst j S A0)
using wf-ty-subst-at[OF hA ty-lam(5) ty-lam(6)] .
have body:
  typing k (map (ty-subst j S) (A0 #  $\Gamma 0$ ))
    (tm-ty-subst j S M0) (ty-subst j S B0)
  using ty-lam(3)[OF ty-lam(4) ty-lam(5) ty-lam(6)] .
have body':
  typing k (ty-subst j S A0 # map (ty-subst j S)  $\Gamma 0$ )
    (tm-ty-subst j S M0) (ty-subst j S B0)
  using body by simp
have lam:
  typing k (map (ty-subst j S)  $\Gamma 0$ )
    (TmLam (ty-subst j S A0) (tm-ty-subst j S M0))
    (TyArr (ty-subst j S A0) (ty-subst j S B0))
  using typing.ty-lam[OF wf body'] .
show ?case
  unfolding tm-ty-subst.simps
  using lam by simp
next
case (ty-tabs k0  $\Gamma 0$  M0 B0 k j S)
have k0-eq: k0 = Suc k
  using ty-tabs(3) by simp
have le: Suc j  $\leq$  k0
  using ty-tabs(5) k0-eq by simp
have wfS: wf-ty k0 (ty-shift 0 S)
  using wf-ty-shift[OF ty-tabs(4)] k0-eq by simp
have body:
  typing k0
    (map (ty-subst (Suc j) (ty-shift 0 S))
      (map (ty-shift 0)  $\Gamma 0$ ))
    (tm-ty-subst (Suc j) (ty-shift 0 S) M0)
    (ty-subst (Suc j) (ty-shift 0 S) B0)
  using ty-tabs(2) wfS le
  by simp
have ctx-eq:
  (ty-subst (Suc j) (ty-shift 0 S)  $\circ$  ty-shift 0) =
    (ty-shift 0  $\circ$  ty-subst j S)
  by (rule ext; simp add: ty-subst-ty-shift)
have body':
  typing k0
    (map (ty-shift 0) (map (ty-subst j S)  $\Gamma 0$ ))
    (tm-ty-subst (Suc j) (ty-shift 0 S) M0)
    (ty-subst (Suc j) (ty-shift 0 S) B0)
  using body by (simp add: map-map ctx-eq)
have tabs:
  typing k (map (ty-subst j S)  $\Gamma 0$ )
    (TmTABS (tm-ty-subst (Suc j) (ty-shift 0 S) M0))
    (TyAll (ty-subst (Suc j) (ty-shift 0 S) B0))
  using typing.ty-tabs[of k map (ty-subst j S)  $\Gamma 0$ ]

```

```

      tm-ty-subst (Suc j) (ty-shift 0 S) M0
      ty-subst (Suc j) (ty-shift 0 S) B0]
      body'
    by (simp add: k0-eq)
  then show ?case by simp
next
case (ty-tapp k0  $\Gamma$ 0 M0 B0 A0 k j S)
have hA: wf-ty (Suc k) A0
  using ty-tapp(3) ty-tapp(4) by simp
have M:
  typing k (map (ty-subst j S)  $\Gamma$ 0)
  (tm-ty-subst j S M0)
  (TyAll (ty-subst (Suc j) (ty-shift 0 S) B0))
  using ty-tapp(2)[OF ty-tapp(4) ty-tapp(5) ty-tapp(6)] by simp
have wf:
  wf-ty k (ty-subst j S A0)
  using wf-ty-subst-at[OF hA ty-tapp(5) ty-tapp(6)] .
have tapp:
  typing k (map (ty-subst j S)  $\Gamma$ 0)
  (TmTApp (tm-ty-subst j S M0) (ty-subst j S A0))
  (ty-subst0 (ty-subst j S A0)
    (ty-subst (Suc j) (ty-shift 0 S) B0))
  using typing.ty-tapp[OF M wf] .
have result:
  ty-subst j S (ty-subst0 A0 B0) =
  ty-subst0 (ty-subst j S A0)
  (ty-subst (Suc j) (ty-shift 0 S) B0)
  using ty-subst-comp[of j S A0 B0] by (simp add: ty-subst0-def)
then show ?case
  using tapp by (simp add: tm-ty-subst.simps result)
qed

```

lemma *typing-ty-subst-at*:

```

  assumes typing (Suc k)  $\Gamma$  M A wf-ty k S j  $\leq$  k
  shows typing k (map (ty-subst j S)  $\Gamma$ )
    (tm-ty-subst j S M) (ty-subst j S A)
proof (rule typing-ty-subst-at-aux[of Suc k  $\Gamma$  M A k S j])
  show typing (Suc k)  $\Gamma$  M A using assms(1) .
  show Suc k = Suc k by simp
  show wf-ty k S using assms(2) .
  show j  $\leq$  k using assms(3) .
qed

```

lemma *typing-ty-subst*:

```

  assumes typing (Suc k)  $\Gamma$  M A wf-ty k S
  shows typing k (map (ty-subst 0 S)  $\Gamma$ )
    (tm-ty-subst 0 S M) (ty-subst 0 S A)
  using typing-ty-subst-at[OF assms, of 0] by simp

```

```

lemma typing-tm-subst-at:
  assumes typing  $k \ \Gamma \ N \ C$ 
    typing  $k \ (ctx\text{-insert } i \ C \ \Gamma) \ M \ A$ 
     $i \leq \text{length } \Gamma$ 
  shows typing  $k \ \Gamma \ (tm\text{-subst } i \ N \ M) \ A$ 
  using assms
proof (induct  $M$  arbitrary:  $k \ \Gamma \ N \ C \ i \ A$  rule: dtm.induct)
  case (TmVar  $n \ k \ \Gamma \ N \ C \ i \ A$ )
  have hmem: ctx-mem (ctx-insert  $i \ C \ \Gamma$ )  $n \ A$ 
    using TmVar.prems(2)
    by (cases rule: typing.cases) auto
  have mem:
    ( $n < i \wedge ctx\text{-mem } \Gamma \ n \ A$ )  $\vee$ 
    ( $n = i \wedge A = C$ )  $\vee$ 
    ( $i < n \wedge ctx\text{-mem } \Gamma \ (n - 1) \ A$ )
    using ctx-mem-insert-cases[OF hmem TmVar.prems(3)] .
  show ?case
    using mem TmVar.prems(1)
    by (auto simp add: tm-subst.simps
      intro: typing.ty-var ctx-mem.ctx-zero ctx-mem.ctx-suc)
next
  case (TmApp  $M \ P \ k \ \Gamma \ N \ C \ i \ A$ )
  obtain  $B$  where
    hM: typing  $k \ (ctx\text{-insert } i \ C \ \Gamma) \ M \ (TyArr \ B \ A)$  and
    hP: typing  $k \ (ctx\text{-insert } i \ C \ \Gamma) \ P \ B$ 
    using TmApp.prems(2)
    by (cases rule: typing.cases) auto
  have M:
    typing  $k \ \Gamma$ 
    (tm-subst  $i \ N \ M$ ) (TyArr  $B \ A$ )
    using TmApp.hyps(1)[OF TmApp.prems(1) hM TmApp.prems(3)] .
  have  $N'$ :
    typing  $k \ \Gamma \ (tm\text{-subst } i \ N \ P) \ B$ 
    using TmApp.hyps(2)[OF TmApp.prems(1) hP TmApp.prems(3)] .
  show ?case
    unfolding tm-subst.simps
    using typing.ty-app[OF  $M \ N'$ ] by simp
next
  case (TmLam  $D \ M \ k \ \Gamma \ N \ C \ i \ A$ )
  obtain  $B$  where
    typeA:  $A = TyArr \ D \ B$  and
    wfD: wf-ty  $k \ D$  and
    hM: typing  $k \ (D \ \# \ ctx\text{-insert } i \ C \ \Gamma) \ M \ B$ 
    using TmLam.prems(2)
    by (cases rule: typing.cases) auto
  have Nshift:
    typing  $k \ (D \ \# \ \Gamma) \ (tm\text{-shift } 0 \ N) \ C$ 
    using typing-weakening[OF TmLam.prems(1), of  $0 \ D$ ]
    by (cases  $\Gamma$ ; simp)

```

```

have len:
  Suc i ≤ length (D #  $\Gamma$ )
  using TmLam.prems(3) by simp
have hM':
  typing k (ctx-insert (Suc i) C (D #  $\Gamma$ )) M B
  using hM by simp
have body:
  typing k (D #  $\Gamma$ )
    (tm-subst (Suc i) (tm-shift 0 N) M) B
  using TmLam.hyps[OF Nshift hM' len]
  by simp
have lam:
  typing k  $\Gamma$ 
    (TmLam D (tm-subst (Suc i) (tm-shift 0 N) M))
    (TyArr D B)
  using typing.ty-lam[OF wfD body] .
show ?case
  unfolding tm-subst.simps
  using lam typeA by simp
next
case (TmTAbs M k  $\Gamma$  N C i A)
obtain B where
  typeA: A = TyAll B and
  hM: typing (Suc k) (map (ty-shift 0) (ctx-insert i C  $\Gamma$ )) M B
  using TmTAbs.prems(2)
  by (cases rule: typing.cases) auto
have Nshift:
  typing (Suc k) (map (ty-shift 0)  $\Gamma$ )
    (tm-ty-shift 0 N) (ty-shift 0 C)
  using typing-ty-shift[OF TmTAbs.prems(1)] .
have hM':
  typing (Suc k)
    (ctx-insert i (ty-shift 0 C) (map (ty-shift 0)  $\Gamma$ )) M B
  using hM by (simp add: map-ctx-insert)
have len:
  i ≤ length (map (ty-shift 0)  $\Gamma$ )
  using TmTAbs.prems(3) by simp
have body:
  typing (Suc k) (map (ty-shift 0)  $\Gamma$ )
    (tm-subst i (tm-ty-shift 0 N) M) B
  using TmTAbs.hyps[OF Nshift hM' len] .
have tabs:
  typing k  $\Gamma$ 
    (TmTAbs (tm-subst i (tm-ty-shift 0 N) M))
    (TyAll B)
  using typing.ty-tabs[OF body] .
show ?case
  unfolding tm-subst.simps
  using tabs typeA by simp

```

```

next
  case (TmTApp M D k  $\Gamma$  N C i A)
  obtain B where
    typeA: A = ty-subst0 D B and
    hM: typing k (ctx-insert i C  $\Gamma$ ) M (TyAll B) and
    wfD: wf-ty k D
    using TmTApp.prems(2)
    by (cases rule: typing.cases) auto
  have M:
    typing k  $\Gamma$  (tm-subst i N M) (TyAll B)
    using TmTApp.hyps[OF TmTApp.prems(1) hM TmTApp.prems(3)] .
  have tapp:
    typing k  $\Gamma$ 
      (TmTApp (tm-subst i N M) D)
      (ty-subst0 D B)
    using typing.ty-tapp[OF M wfD] .
  show ?case
    unfolding tm-subst.simps
    using tapp typeA by simp
qed

```

```

lemma typing-tm-subst0:
  assumes typing k  $\Gamma$  N C
    typing k (C #  $\Gamma$ ) M A
  shows typing k  $\Gamma$  (tm-subst0 N M) A
proof –
  have major: typing k (ctx-insert 0 C  $\Gamma$ ) M A
    using assms(2) by (cases  $\Gamma$ ; simp)
  have sub:
    typing k  $\Gamma$  (tm-subst 0 N M) A
    using typing-tm-subst-at[OF assms(1) major] by simp
  then show ?thesis
    by (simp add: tm-subst0-def)
qed

```

```

theorem subject-reduction:
  assumes typing k  $\Gamma$  M A M  $\longrightarrow_{\beta}$  M'
  shows typing k  $\Gamma$  M' A
  using assms(2) assms(1)
proof (induct arbitrary: k  $\Gamma$  A rule: beta.induct)
  case (beta-appL M M' N k  $\Gamma$  A)
  obtain B where
    hM: typing k  $\Gamma$  M (TyArr B A) and
    hN: typing k  $\Gamma$  N B
    using beta-appL.prems
    by (cases rule: typing.cases) auto
  have hM':
    typing k  $\Gamma$  M' (TyArr B A)
    using beta-appL.hyps(2)[OF hM] .

```

```

show ?case
  using typing.ty-app[OF hM' hN] .
next
  case (beta-appR N N' M k  $\Gamma$  A)
  obtain B where
    hM: typing k  $\Gamma$  M (TyArr B A) and
    hN: typing k  $\Gamma$  N (B)
    using beta-appR.prems
    by (cases rule: typing.cases) auto
  have hN':
    typing k  $\Gamma$  N' B
    using beta-appR.hyps(2)[OF hN] .
  show ?case
    using typing.ty-app[OF hM hN'] .
next
  case (beta-lam M M' D k  $\Gamma$  A)
  obtain B where
    wfD: wf-ty k D and
    hM: typing k (D #  $\Gamma$ ) M B and
    typeA: A = TyArr D B
    using beta-lam.prems
    by (cases rule: typing.cases) auto
  have hM':
    typing k (D #  $\Gamma$ ) M' B
    using beta-lam.hyps(2)[OF hM] .
  have lam:
    typing k  $\Gamma$  (TmLam D M') (TyArr D B)
    using typing.ty-lam[OF wfD hM'] .
  show ?case
    using lam typeA by simp
next
  case (beta-tabs M M' k  $\Gamma$  A)
  obtain B where
    hM: typing (Suc k) (map (ty-shift 0)  $\Gamma$ ) M B and
    typeA: A = TyAll B
    using beta-tabs.prems
    by (cases rule: typing.cases) auto
  have hM':
    typing (Suc k) (map (ty-shift 0)  $\Gamma$ ) M' B
    using beta-tabs.hyps(2)[OF hM] .
  have tabs:
    typing k  $\Gamma$  (TmTAbs M') (TyAll B)
    using typing.ty-tabs[OF hM'] .
  show ?case
    using tabs typeA by simp
next
  case (beta-tapp M M' D k  $\Gamma$  A)
  obtain B where
    hM: typing k  $\Gamma$  M (TyAll B) and

```

```

    wfD: wf-ty k D and
    typeA: A = ty-subst0 D B
    using beta-tapp.premis
    by (cases rule: typing.cases) auto
  have hM':
    typing k  $\Gamma$  M' (TyAll B)
    using beta-tapp.hyps(2)[OF hM] .
  have tapp:
    typing k  $\Gamma$  (TmTApp M' D) (ty-subst0 D B)
    using typing.ty-tapp[OF hM' wfD] .
  show ?case
    using tapp typeA by simp
next
case (beta-term D M N k  $\Gamma$  A)
obtain B where
  hM: typing k  $\Gamma$  (TmLam D M) (TyArr D B) and
  hN: typing k  $\Gamma$  N D and
  typeA: A = B
  using beta-term.premis
  by (auto elim: typing.cases)
obtain wfD: wf-ty k D and hbody: typing k (D #  $\Gamma$ ) M B
  using hM
  by (auto elim: typing.cases)
have sub:
  typing k  $\Gamma$  (tm-subst0 N M) B
  using typing-tm-subst0[OF hN hbody] .
show ?case
  using sub typeA by simp
next
case (beta-type M D k  $\Gamma$  A)
obtain B where
  hM: typing k  $\Gamma$  (TmTAbs M) (TyAll B) and
  wfD: wf-ty k D and
  typeA: A = ty-subst0 D B
  using beta-type.premis
  by (cases rule: typing.cases) auto
obtain hbody where
  hbody: typing (Suc k) (map (ty-shift 0)  $\Gamma$ ) M B
  using hM
  by (cases rule: typing.cases) auto
have ctr:
  (ty-subst 0 D  $\circ$  ty-shift 0) = id
  by (rule ext; simp add: ty-subst-shift-id)
have sub:
  typing k  $\Gamma$  (tm-ty-subst0 D M) (ty-subst0 D B)
  using typing-ty-subst[OF hbody wfD]
  by (simp add: tm-ty-subst0-def ty-subst0-def map-map ctr)
show ?case
  using sub typeA by simp

```

```

qed

end
theory System-F-Untyped
  imports System-F
begin

datatype ulam =
  UVar nat
| UApp ulam ulam
| ULam ulam

fun u-shift :: nat ⇒ ulam ⇒ ulam where
  u-shift k (UVar n) =
    (if n < k then UVar n else UVar (Suc n))
| u-shift k (UApp M N) = UApp (u-shift k M) (u-shift k N)
| u-shift k (ULam M) = ULam (u-shift (Suc k) M)

fun u-subst :: nat ⇒ ulam ⇒ ulam ⇒ ulam where
  u-subst k N (UVar n) =
    (if n < k then UVar n else
     if n = k then N else UVar (n - 1))
| u-subst k N (UApp M P) =
  UApp (u-subst k N M) (u-subst k N P)
| u-subst k N (ULam M) =
  ULam (u-subst (Suc k) (u-shift 0 N) M)

definition u-subst0 :: ulam ⇒ ulam ⇒ ulam
where u-subst0 N M = u-subst 0 N M

lemma u-subst-shift:
  u-subst k N (u-shift k M) = M
by (induct M arbitrary: k N rule: ulam.induct)
   (simp-all split: if-splits)

lemma u-shift-shift:
  l ≤ k ⟶
    u-shift l (u-shift k M) =
      u-shift (Suc k) (u-shift l M)
proof (induct M arbitrary: k l rule: ulam.induct)
  case (UVar n)
  then show ?case
    by (auto split: if-splits; arith)
next
  case (UApp M P)
  then show ?case by (intro impI; simp-all; blast)
next
  case (ULam M)
  then show ?case by (intro impI; simp-all; blast)

```

qed

lemma *u-shift-subst-comm*:

$l \leq k \longrightarrow$
 $u\text{-shift } l \ (u\text{-subst } k \ N \ M) =$
 $u\text{-subst } (Suc \ k) \ (u\text{-shift } l \ N) \ (u\text{-shift } l \ M)$
proof (*induct M arbitrary: k l N rule: ulam.induct*)
 case (*UVar n*)
 then show ?case
 by (*intro impI; simp split: if-splits; arith*)
next
 case (*UApp M P*)
 then show ?case by (*intro impI; simp-all; blast*)
next
 case (*ULam M*)
 then show ?case
 by (*intro impI; simp-all add: u-shift-shift; blast*)
qed

lemma *u-subst-comp-at*:

$l \leq k \longrightarrow$
 $u\text{-subst } l \ (u\text{-subst } k \ N \ P)$
 $(u\text{-subst } (Suc \ k) \ (u\text{-shift } l \ N) \ M) =$
 $u\text{-subst } k \ N \ (u\text{-subst } l \ P \ M)$
proof (*induct M arbitrary: k l N P rule: ulam.induct*)
 case (*UVar x*)
 then show ?case
 by (*intro impI; simp add: u-subst-shift split: if-splits; arith*)
next
 case (*UApp M P*)
 then show ?case by (*intro impI; simp-all; blast*)
next
 case (*ULam M*)
 then show ?case
 by (*intro impI; simp-all add: u-shift-shift u-shift-subst-comm; blast*)
qed

inductive *ubeta* :: *ulam* $\Rightarrow$ *ulam* $\Rightarrow$ *bool*

($\langle \cdot \longrightarrow_u \cdot \rangle \ [80,80] \ 80$) **where**
uappL: $M \longrightarrow_u M' \Longrightarrow$
 $UApp \ M \ N \longrightarrow_u UApp \ M' \ N$
| *uappR*: $N \longrightarrow_u N' \Longrightarrow$
 $UApp \ M \ N \longrightarrow_u UApp \ M \ N'$
| *ulam*: $M \longrightarrow_u M' \Longrightarrow$
 $ULam \ M \longrightarrow_u ULam \ M'$
| *ured*: $UApp \ (ULam \ M) \ N \longrightarrow_u u\text{-subst0 } N \ M$

inductive *uSN* :: *ulam* $\Rightarrow$ *bool* **where**

uSN-intro: $(\bigwedge M'. M \longrightarrow_u M' \Longrightarrow uSN \ M') \Longrightarrow uSN \ M$

```

lemma uSN-preserved:
  assumes uSN M and  $M \longrightarrow_u M'$ 
  shows uSN M'
  using assms by (cases) auto

lemma u-subst-beta-at:
  assumes  $M \longrightarrow_u M'$ 
  shows  $u\text{-subst } k \ N \ M \longrightarrow_u u\text{-subst } k \ N \ M'$ 
  using assms
proof (induct arbitrary: k N rule: ubeta.induct)
  case (uappL M M' P)
  show ?case
    unfolding u-subst.simps
    by (rule ubeta.uappL, rule uappL(2)[of k N])
next
  case (uappR P P' M)
  show ?case
    unfolding u-subst.simps
    by (rule ubeta.uappR, rule uappR(2)[of k N])
next
  case (ulam M M')
  show ?case
    unfolding u-subst.simps
    apply (rule ubeta.ulam)
    using ulam(2)[of Suc k u-shift 0 N] .
next
  case (ured M P)
  have eq:
    
$$\begin{aligned}
&u\text{-subst0 } (u\text{-subst } k \ N \ P) \\
&\quad (u\text{-subst } (Suc \ k) \ (u\text{-shift } 0 \ N) \ M) = \\
&\quad u\text{-subst } k \ N \ (u\text{-subst0 } P \ M)
\end{aligned}$$

    using u-subst-comp-at[of 0 k N P M]
    by (simp add: u-subst0-def)
  have root:
    
$$\begin{aligned}
&UApp \ (ULam \ (u\text{-subst } (Suc \ k) \ (u\text{-shift } 0 \ N) \ M)) \\
&\quad (u\text{-subst } k \ N \ P) \longrightarrow_u \\
&\quad u\text{-subst0 } (u\text{-subst } k \ N \ P) \\
&\quad (u\text{-subst } (Suc \ k) \ (u\text{-shift } 0 \ N) \ M)
\end{aligned}$$

    by (rule ubeta.ured)
  then show ?case using eq by (simp add: u-subst0-def)
qed

lemma uSN-subst-beta:
  assumes  $M \longrightarrow_u M'$ 
  shows  $u\text{-subst0 } N \ M \longrightarrow_u u\text{-subst0 } N \ M'$ 
  using u-subst-beta-at[OF assms, of 0 N]
  by (simp add: u-subst0-def)

```

```

lemma uSN-reflect-subst:
  assumes uSN (u-subst0 (UVar 0) M)
  shows uSN M
proof –
  have aux:  $\bigwedge X. uSN\ X \implies$ 
     $\forall M. X = u\text{-subst0}\ (UVar\ 0)\ M \longrightarrow uSN\ M$ 
  proof –
    fix X
    assume uSN X
    then show  $\forall M. X = u\text{-subst0}\ (UVar\ 0)\ M \longrightarrow uSN\ M$ 
    proof (induct rule: uSN.induct)
      case (uSN-intro X)
      show ?case
      proof (intro allI impI)
        fix M
        assume eq:  $X = u\text{-subst0}\ (UVar\ 0)\ M$ 
        show uSN M
        proof (rule uSN.uSN-intro)
          fix M'
          assume redM:  $M \longrightarrow_u M'$ 
          have redX:  $X \longrightarrow_u u\text{-subst0}\ (UVar\ 0)\ M'$ 
          using eq
          by (auto intro: uSN-subst-beta[OF redM])
          have ih:
             $\forall R. u\text{-subst0}\ (UVar\ 0)\ M' =$ 
               $u\text{-subst0}\ (UVar\ 0)\ R \longrightarrow uSN\ R$ 
          using uSN-intro(2)[OF redX] .
          from ih show uSN M' by blast
        qed
      qed
    qed
  then show uSN M using assms by blast
qed

```

```

inductive uFst :: ulam  $\Rightarrow$  ulam  $\Rightarrow$  bool where
  uFst-intro: uFst (UApp M N) M

```

```

lemma uSN-of-fst:
  assumes uSN (UApp M N)
  shows uSN M
proof –
  have aux:  $\bigwedge X. uSN\ X \implies$ 
     $\forall P\ Q. X = UApp\ P\ Q \longrightarrow uSN\ P$ 
  proof –
    fix X
    assume uSN X
    then show  $\forall P\ Q. X = UApp\ P\ Q \longrightarrow uSN\ P$ 
    proof (induct rule: uSN.induct)

```

```

case (uSN-intro X)
show ?case
proof (intro allI allI impI)
  fix P Q
  assume eq: X = UApp P Q
  show uSN P
  proof (rule uSN.uSN-intro)
    fix P'
    assume redP: P  $\longrightarrow_u$  P'
    have redX: X  $\longrightarrow_u$  UApp P' Q
      using eq redP by (auto intro: uappL)
    have ih:  $\forall R S. \text{UApp } P' Q = \text{UApp } R S \longrightarrow \text{uSN } R$ 
      using uSN-intro(2)[OF redX] .
    from ih show uSN P' by blast
  qed
qed
qed
qed
then show uSN M using assms by blast
qed

```

```

lemma udouble-SN-aux:
  assumes a: uSN a
  and b: uSN b
  and hyp:  $\bigwedge x z. \llbracket \bigwedge y. x \longrightarrow_u y \Longrightarrow \text{uSN } y; \bigwedge y. x \longrightarrow_u y \Longrightarrow P y z; \bigwedge u. z \longrightarrow_u u \Longrightarrow \text{uSN } u; \bigwedge u. z \longrightarrow_u u \Longrightarrow P x u \rrbracket \Longrightarrow P x z$ 
  shows P a b
proof -
  from a
  have r:  $\bigwedge b. \text{uSN } b \Longrightarrow P a b$ 
  proof (induct a rule: uSN.induct)
    case (uSN-intro x)
    note SNI' = uSN-intro
    have uSN b by fact
    thus ?case
    proof (induct b rule: uSN.induct)
      case (uSN-intro y)
      with SNI' show ?case
        by (metis uSN.simps hyp)
    qed
  qed
  from b show ?thesis by (rule r)
qed

```

```

lemma udouble-SN[consumes 2]:
  assumes a: uSN a

```

```

and  $b: uSN\ b$ 
and  $c: \bigwedge x\ z.$ 
   $\llbracket \bigwedge y. x \rightarrow_u y \implies P\ y\ z;$ 
   $\bigwedge u. z \rightarrow_u u \implies P\ x\ u \rrbracket \implies P\ x\ z$ 
shows  $P\ a\ b$ 
using  $a\ b\ c$ 
by (smt (verit, best) udouble-SN-aux)

```

definition *uneutral* :: *ulam set* $\Rightarrow$ *bool* **where**

```

uneutral  $M \longleftrightarrow$ 
   $(\exists k. M = UVar\ k) \vee$ 
   $(\exists P\ Q. M = UApp\ P\ Q)$ 

```

The neutral terms used by the candidate condition are variables and arbitrary applications. In particular, applications are considered neutral even when their head is an abstraction. This deliberately broad formulation makes the candidate closure stable under the full compatible reduction relation; the redex case is handled explicitly when the saturation lemmas analyze the reducts of an application.

lemma *uneutral-var*:

```

uneutral (UVar  $k$ )
by (auto simp add: uneutral-def)

```

lemma *uneutral-app*:

```

uneutral (UApp  $M\ N$ )
by (auto simp add: uneutral-def)

```

definition *uCR1* :: *ulam set* $\Rightarrow$ *bool* **where**

```

uCR1  $C \longleftrightarrow (\forall M. M \in C \longrightarrow uSN\ M)$ 

```

definition *uCR2* :: *ulam set* $\Rightarrow$ *bool* **where**

```

uCR2  $C \longleftrightarrow$ 
   $(\forall M\ M'. M \in C \longrightarrow M \rightarrow_u M' \longrightarrow M' \in C)$ 

```

definition *uCR3* :: *ulam set* $\Rightarrow$ *bool* **where**

```

uCR3  $C \longleftrightarrow$ 
   $(\forall M. \text{uneutral}\ M \longrightarrow$ 
   $(\forall M'. M \rightarrow_u M' \longrightarrow M' \in C) \longrightarrow M \in C)$ 

```

definition *ucandidate* :: *ulam set* $\Rightarrow$ *bool* **where**

```

ucandidate  $C \longleftrightarrow uCR1\ C \wedge uCR2\ C \wedge uCR3\ C$ 

```

definition *uSN-set* :: *ulam set* **where**

```

uSN-set =  $\{M. uSN\ M\}$ 

```

lemma *uSN-set-candidate*:

```

ucandidate uSN-set

```

proof (*unfold ucandidate-def, intro conjI*)

```

show uCR1 uSN-set

```

```

    unfolding uCR1-def uSN-set-def by simp
next
  show uCR2 uSN-set
    unfolding uCR2-def uSN-set-def
    by (blast intro: uSN-preserved)
next
  show uCR3 uSN-set
    unfolding uCR3-def uSN-set-def
    by (blast intro: uSN.uSN-intro)
qed

lemma uvar-mem-candidate:
  assumes ucandidate C
  shows UVar k ∈ C
  using assms
  unfolding ucandidate-def uCR3-def
  by (blast intro: uneutral-var elim: ubeta.cases)

definition uarr :: ulam set ⇒ ulam set ⇒ ulam set where
  uarr C D = {M. ∀ N. N ∈ C ⟶ UApp M N ∈ D}

lemma uarr-CR2:
  assumes uCR2 D
  shows uCR2 (uarr C D)
  using assms
  unfolding uCR2-def uarr-def
  by (blast intro: uappL)

lemma uarr-CR1:
  assumes uCR1 C and uCR1 D and uCR3 C
  shows uCR1 (uarr C D)
proof (unfold uCR1-def, intro strip)
  fix M
  assume hM: M ∈ uarr C D
  have UVar 0 ∈ C
  using assms(3)
  unfolding uCR3-def
  by (blast intro: uneutral-var elim: ubeta.cases)
  then have uSN (UApp M (UVar 0))
  using assms(2) hM by (simp add: uCR1-def uarr-def)
  then show uSN M by (rule uSN-of-fst)
qed

lemma uarr-CR3:
  assumes uCR1 C and uCR2 C and uCR3 C
  and uCR3 D
  shows uCR3 (uarr C D)
proof (unfold uCR3-def, intro strip)
  fix M

```

```

assume  $hM$ : uneutral  $M$ 
assume  $hMred$ :  $\forall M'. M \rightarrow_u M' \rightarrow M' \in uarr\ C\ D$ 
show  $M \in uarr\ C\ D$ 
proof (simp add: uarr-def, intro strip)
  fix  $N$ 
  assume  $N$ :  $N \in C$ 
  have  $SN$ :  $uSN\ N$ 
    using assms(1)  $N$  unfolding uCR1-def by blast
  show  $UApp\ M\ N \in D$ 
  using  $SN\ N\ hM\ hMred$ 
  proof (induct N rule: uSN.induct)
    case (uSN-intro N)
    note  $SNI' = uSN\text{-intro}$ 
    have  $hN$ :  $N \in C$  by fact
    have neutral: uneutral ( $UApp\ M\ N$ )
      by (rule uneutral-app)
    have reds:  $\bigwedge R. UApp\ M\ N \rightarrow_u R \implies R \in D$ 
    proof –
      fix  $R$ 
      assume red:  $UApp\ M\ N \rightarrow_u R$ 
      then show  $R \in D$ 
      proof (cases rule: ubeta.cases)
        case (uappL P)
        then show ?thesis using  $hMred\ hN$ 
          by (auto simp add: uarr-def)
        next
        case (uappR Q)
        have  $Q \in C$ 
          using  $hN$  assms(2) uappR unfolding uCR2-def by blast
        then show ?thesis using  $SNI'\ uappR$ 
          by blast
        next
        case ured
        then show ?thesis using  $hM$ 
          by (auto simp add: uneutral-def)
      qed
    qed
  show ?case using assms(4) neutral reds
    unfolding uCR3-def by blast
  qed
qed
qed

```

lemma *uarr-candidate*:

```

assumes ucandidate C and ucandidate D
shows ucandidate (uarr C D)
using assms
unfolding ucandidate-def
by (intro conjI; blast intro: uarr-CR1 uarr-CR2 uarr-CR3)

```

lemma *ubeta-lam-cases*:

assumes $ULam\ M \longrightarrow_u P$
obtains M' **where** $M \longrightarrow_u M'$ **and** $P = ULam\ M'$
using *assms* **by** (*cases rule: ubeta.cases*) *auto*

The abstraction lemma is the semantic heart of the untyped development. Its premise says that substituting every argument from the domain candidate into the body produces an element of the codomain candidate. The proof first obtains strong normalization of the body by testing it with a fresh variable, and then uses simultaneous induction on body and argument reductions to establish the candidate condition for the application of the abstraction.

lemma *uabs-RED*:

assumes $C: ucandidate\ C$
and $D: ucandidate\ D$
and $asm: \forall s. s \in C \longrightarrow u\text{-subst0}\ s\ M \in D$
shows $ULam\ M \in uarr\ C\ D$

proof –

have $b1: uSN\ M$

proof –

have $UVar\ 0 \in C$
using C **by** (*blast intro: uvar-mem-candidate*)
then have $u\text{-subst0}\ (UVar\ 0)\ M \in D$
using asm **by** *blast*
then have $uSN\ (u\text{-subst0}\ (UVar\ 0)\ M)$
using D **unfolding** *ucandidate-def uCR1-def* **by** *blast*
then have $uSN\ M$ **by** (*rule uSN-reflect-subst*)
moreover have $uCR1\ D$
using D **unfolding** *ucandidate-def* **by** *blast*
ultimately show $uSN\ M$ **by** *blast*

qed

show $ULam\ M \in uarr\ C\ D$

proof (*simp add: uarr-def, intro strip*)

fix u

assume $u: u \in C$

then have $uSN: uSN\ u$

using C **unfolding** *ucandidate-def uCR1-def* **by** *blast*

show $UApp\ (ULam\ M)\ u \in D$

using $b1\ uSN\ u\ asm$

proof (*induct M u rule: udouble-SN*)

fix $M\ u$

assume $ih1$:

$\bigwedge M'. \llbracket M \longrightarrow_u M' ;$
 $u \in C ; \forall s. s \in C \longrightarrow u\text{-subst0}\ s\ M' \in D \rrbracket$
 $\implies UApp\ (ULam\ M')\ u \in D$

assume $ih2$:

$\bigwedge u'. \llbracket u \longrightarrow_u u' ;$
 $u' \in C ; \forall s. s \in C \longrightarrow u\text{-subst0}\ s\ M \in D \rrbracket$

```

     $\implies UApp (ULam M) u' \in D$ 
  assume  $uC: u \in C$ 
  assume  $hM: \forall s. s \in C \longrightarrow u\text{-subst0 } s M \in D$ 
  have reds:
     $\bigwedge R. UApp (ULam M) u \longrightarrow_u R \implies R \in D$ 
  proof -
    fix R
    assume red:  $UApp (ULam M) u \longrightarrow_u R$ 
    then show  $R \in D$ 
    proof (cases rule: ubeta.cases)
      case (uappL P)
      obtain  $M'$  where hred:  $M \longrightarrow_u M'$ 
        and P-eq:  $P = ULam M'$ 
        using ubeta-lam-cases[OF uappL(2)] .
      have  $hM': \forall s. s \in C \longrightarrow u\text{-subst0 } s M' \in D$ 
      proof (intro allI impI)
        fix s
        assume sC:  $s \in C$ 
        have hsr:  $u\text{-subst0 } s M \longrightarrow_u u\text{-subst0 } s M'$ 
          using hred by (rule uSN-subst-beta)
        have D2:  $uCR2 D$ 
          using D unfolding ucandidate-def by blast
        show  $u\text{-subst0 } s M' \in D$ 
          using hM sC hsr D2
          unfolding uCR2-def by blast
      qed
    show ?thesis
      using ih1 hred uC hM' uappL P-eq by simp
    next
      case (uappR u')
      have C2:  $uCR2 C$ 
        using C unfolding ucandidate-def by blast
      have  $u' \in C$ 
        using uC C2 uappR unfolding uCR2-def by blast
      then show ?thesis
        using ih2 uappR hM by blast
    next
      case ured
      then show ?thesis using hM uC by blast
    qed
  qed
  have neutral:  $uneutral (UApp (ULam M) u)$ 
    by (rule uneutral-app)
  show  $UApp (ULam M) u \in D$ 
    using D neutral reds unfolding ucandidate-def uCR3-def by blast
  qed
qed
qed

```

```

end
theory System-F-Erasure
  imports System-F System-F-Untyped
begin

fun f-erase :: dtm  $\Rightarrow$  ulam where
  f-erase (TmVar n) = UVar n
| f-erase (TmApp M N) = UApp (f-erase M) (f-erase N)
| f-erase (TmLam A M) = ULam (f-erase M)
| f-erase (TmTAbs M) = f-erase M
| f-erase (TmTApp M A) = f-erase M

lemma f-erase-tm-shift:
  f-erase (tm-shift k M) = u-shift k (f-erase M)
  by (induct M arbitrary: k rule: dtm.induct) simp-all

lemma f-erase-tm-ty-shift:
  f-erase (tm-ty-shift k M) = f-erase M
  by (induct M arbitrary: k rule: dtm.induct) simp-all

lemma f-erase-tm-subst:
  f-erase (tm-subst k N M) =
    u-subst k (f-erase N) (f-erase M)
  by (induct M arbitrary: k N rule: dtm.induct)
    (simp-all add: f-erase-tm-shift f-erase-tm-ty-shift)

lemma f-erase-tm-ty-subst:
  f-erase (tm-ty-subst k S M) = f-erase M
  by (induct M arbitrary: k S rule: dtm.induct) simp-all

lemma f-erase-tm-subst0:
  f-erase (tm-subst0 N M) =
    u-subst0 (f-erase N) (f-erase M)
  by (simp add: tm-subst0-def u-subst0-def f-erase-tm-subst)

lemma f-erase-tm-ty-subst0:
  f-erase (tm-ty-subst0 S M) = f-erase M
  by (simp add: tm-ty-subst0-def f-erase-tm-ty-subst)

fun f-type-count :: dtm  $\Rightarrow$  nat where
  f-type-count (TmVar n) = 0
| f-type-count (TmApp M N) = f-type-count M + f-type-count N
| f-type-count (TmLam A M) = f-type-count M
| f-type-count (TmTAbs M) = Suc (f-type-count M)
| f-type-count (TmTApp M A) = Suc (f-type-count M)

lemma f-type-count-tm-ty-subst:
  f-type-count (tm-ty-subst k S M) = f-type-count M
  by (induct M arbitrary: k S rule: dtm.induct) simp-all

```

lemma *f-type-count-tm-ty-subst0*:
 $f\text{-type-count } (tm\text{-ty-subst0 } S \ M) = f\text{-type-count } M$
by (*simp add: tm-ty-subst0-def f-type-count-tm-ty-subst*)

lemma *f-beta-progress*:
assumes $M \longrightarrow_{\beta} M'$
shows $ubeta \ (f\text{-erase } M) \ (f\text{-erase } M') \vee$
 $f\text{-erase } M = f\text{-erase } M' \wedge$
 $f\text{-type-count } M' < f\text{-type-count } M$
using *assms*
proof (*induct rule: beta.induct*)
case *beta-appL*
then show *?case*
by (*auto intro: ubeta.uappL*)
next
case *beta-appR*
then show *?case*
by (*auto intro: ubeta.uappR*)
next
case *beta-lam*
then show *?case*
by (*auto intro: ubeta.ulam*)
next
case *beta-tabs*
then show *?case*
by *auto*
next
case *beta-tapp*
then show *?case*
by *auto*
next
case (*beta-term A M N*)
have *root*:
 $ubeta \ (UApp \ (ULam \ (f\text{-erase } M)) \ (f\text{-erase } N))$
 $\ (u\text{-subst0 } (f\text{-erase } N) \ (f\text{-erase } M))$
by (*rule ubeta.ured*)
then show *?case*
by (*simp add: f-erase-tm-subst0 u-subst0-def*)
next
case *beta-type*
then show *?case*
by (*rule disjI2;*
 $simp \ add: \ f\text{-erase-tm-ty-subst0 } f\text{-type-count-tm-ty-subst0}$)
qed

lemma *fSN-reflect*:
assumes $uSN \ (f\text{-erase } M)$
shows $SN \ M$

```

proof –
  have aux:  $\bigwedge U. uSN\ U \implies$ 
     $\forall M. f\text{-erase}\ M = U \longrightarrow SN\ M$ 
proof –
  fix U
  assume uSN U
  then show  $\forall M. f\text{-erase}\ M = U \longrightarrow SN\ M$ 
proof (induct rule: uSN.induct)
  case (uSN-intro U)
  show ?case
proof (intro allI impI)
  fix M
  assume eM: f-erase M = U
  have type-aux:
     $\bigwedge n. \forall X. f\text{-type-count}\ X = n \longrightarrow$ 
       $f\text{-erase}\ X = U \longrightarrow SN\ X$ 
proof –
  fix n
  show  $\forall X. f\text{-type-count}\ X = n \longrightarrow$ 
     $f\text{-erase}\ X = U \longrightarrow SN\ X$ 
proof (induct n rule: less-induct)
  case (less n)
  show ?case
proof (intro allI impI impI)
  fix X
  assume countX: f-type-count X = n
  assume eX: f-erase X = U
  show SN X
proof (rule SN.SN-intro)
  fix X'
  assume red:  $X \longrightarrow_{\beta} X'$ 
  have prog:
    ubeta (f-erase X) (f-erase X')  $\vee$ 
     $f\text{-erase}\ X = f\text{-erase}\ X' \wedge$ 
     $f\text{-type-count}\ X' < f\text{-type-count}\ X$ 
    using f-beta-progress[OF red] .
  then show SN X'
proof (elim disjE)
  assume ured: ubeta (f-erase X) (f-erase X')
  have ured':  $U \longrightarrow_u f\text{-erase}\ X'$ 
    using ured by (simp only: eX[symmetric])
  have ih:
     $\forall R. f\text{-erase}\ X' = f\text{-erase}\ R \longrightarrow SN\ R$ 
proof (intro allI impI)
  fix R
  assume eq: f-erase X' = f-erase R
  have eq': f-erase R = f-erase X'
    using eq by simp
  from uSN-intro(2)[OF ured'] show SN R

```

```

      using eq' by blast
    qed
  from ih show SN X' by blast
next
  assume dec:
    f-erase X = f-erase X' ∧
    f-type-count X' < f-type-count X
  have count-lt: f-type-count X' < n
    using countX dec by arith
  have ih:
    ∀ R. f-type-count R = f-type-count X' ⟶
    f-erase R = U ⟶ SN R
    using less[of f-type-count X] count-lt by blast
  have eEq: f-erase X = f-erase X'
    using dec by blast
  have eX': f-erase X' = U
    using eEq eX by simp
  from ih eX' show SN X' by simp
qed
qed
qed
qed
qed
from type-aux[of f-type-count M] eM
show SN M by simp
qed
qed
qed
then show SN M using assms by blast
qed

end
theory System-F-Normalization
  imports System-F-Erasure
begin

definition f-env-ok :: (nat ⇒ ulam set) ⇒ bool where
  f-env-ok q ⟷ (∀ k. ucandidate (q k))

definition f-all-sem :: (ulam set ⇒ ulam set) ⇒ ulam set where
  f-all-sem F = {M. ∀ C. ucandidate C ⟶ M ∈ F C}

lemma f-all-candidate:
  assumes H: ∀ C. ucandidate C ⟶ ucandidate (F C)
  shows ucandidate (f-all-sem F)
proof (unfold ucandidate-def, intro conjI)
  show uCR1 (f-all-sem F)
    unfolding uCR1-def f-all-sem-def
    using H uSN-set-candidate

```

```

    unfolding ucandidate-def uCR1-def
    by blast
next
show uCR2 (f-all-sem F)
  unfolding uCR2-def f-all-sem-def
  using H
  unfolding ucandidate-def uCR2-def
  by blast
next
show uCR3 (f-all-sem F)
  unfolding uCR3-def f-all-sem-def
  using H
  unfolding ucandidate-def uCR3-def
  by blast
qed

definition f-env-insert ::
  nat  $\Rightarrow$  ulam set  $\Rightarrow$  (nat  $\Rightarrow$  ulam set)  $\Rightarrow$ 
  nat  $\Rightarrow$  ulam set
where
  f-env-insert k C  $\varrho$  n =
    (if n < k then  $\varrho$  n else
     if n = k then C else  $\varrho$  (n - 1))

lemma f-env-insert-zero:
  f-env-insert 0 C  $\varrho$  = ty-env-cons C  $\varrho$ 
proof (rule ext)
  fix x
  show f-env-insert 0 C  $\varrho$  x = ty-env-cons C  $\varrho$  x
  proof (cases x)
  case 0
  then show ?thesis
    by (simp add: f-env-insert-def ty-env-cons-def)
  next
  case (Suc n)
  then show ?thesis
    by (simp add: f-env-insert-def ty-env-cons-def split: if-splits)
qed
qed

lemma f-env-insert-suc-cons:
  f-env-insert (Suc k) C (ty-env-cons D  $\varrho$ ) =
    ty-env-cons D (f-env-insert k C  $\varrho$ )
proof (rule ext)
  fix x
  show f-env-insert (Suc k) C (ty-env-cons D  $\varrho$ ) x =
    ty-env-cons D (f-env-insert k C  $\varrho$ ) x
  proof (induct x)
  case 0

```

```

    then show ?case
    by (simp add: f-env-insert-def ty-env-cons-def)
next
  case (Suc n)
  then show ?case
  proof (cases n)
    case 0
    then show ?thesis
    by (simp add: f-env-insert-def ty-env-cons-def)
  next
    case (Suc m)
    then show ?thesis
    by (simp add: f-env-insert-def ty-env-cons-def
        split: if-splits; arith)
  qed
qed
qed

lemma f-env-ok-cons:
  assumes f-env-ok  $\varrho$  and ucandidate C
  shows f-env-ok (ty-env-cons C  $\varrho$ )
  using assms
  unfolding f-env-ok-def ty-env-cons-def
  by (intro allI; case-tac k; simp-all)

fun f-sem :: dty  $\Rightarrow$  (nat  $\Rightarrow$  ulam set)  $\Rightarrow$  ulam set where
  f-sem (TyVar k)  $\varrho$  =  $\varrho$  k
| f-sem (TyArr A B)  $\varrho$  =
  uarr (f-sem A  $\varrho$ ) (f-sem B  $\varrho$ )
| f-sem (TyAll A)  $\varrho$  =
  f-all-sem ( $\lambda C$ . f-sem A (ty-env-cons C  $\varrho$ ))

lemma f-sem-candidate:
  f-env-ok  $\varrho \longrightarrow$  ucandidate (f-sem A  $\varrho$ )
proof (induct A arbitrary:  $\varrho$ )
  case (TyVar k)
  then show ?case
  unfolding f-sem.simps f-env-ok-def ucandidate-def by blast
next
  case (TyArr A B)
  then show ?case
  by (auto intro: uarr-candidate)
next
  case (TyAll A)
  have body:
    f-env-ok  $\varrho \longrightarrow$ 
    ( $\forall C$ . ucandidate C  $\longrightarrow$ 
      ucandidate (f-sem A (ty-env-cons C  $\varrho$ )))
  using TyAll(1)

```

by (*blast intro: f-env-ok-cons*)
 then show ?case
 unfolding *f-sem.simps*
 by (*intro impI; blast intro: f-all-candidate*)
 qed

lemma *f-sem-ty-shift*:
 $f\text{-sem } (ty\text{-shift } k \ A) \ (f\text{-env-insert } k \ C \ \varrho) =$
 $f\text{-sem } A \ \varrho$
proof (*induct A arbitrary: k C ρ*)
 case (*TyVar n*)
 then show ?case
 by (*simp add: f-env-insert-def split: if-splits; arith*)
 next
 case (*TyArr A B*)
 then show ?case by *simp*
 next
 case (*TyAll A*)
 have *body-eq*:
 $\forall D. f\text{-sem } (ty\text{-shift } (Suc \ k) \ A)$
 $(ty\text{-env-cons } D \ (f\text{-env-insert } k \ C \ \varrho)) =$
 $f\text{-sem } A \ (ty\text{-env-cons } D \ \varrho)$
 using *TyAll(1)*
 by (*simp add: f-env-insert-suc-cons[symmetric]*)
 show ?case
 using *body-eq*
 by (*simp add: f-all-sem-def*)
 qed

lemma *f-sem-ty-subst*:
 $f\text{-sem } (ty\text{-subst } k \ S \ A) \ \varrho =$
 $f\text{-sem } A \ (f\text{-env-insert } k \ (f\text{-sem } S \ \varrho) \ \varrho)$
proof (*induct A arbitrary: k S ρ*)
 case (*TyVar n*)
 then show ?case
 by (*simp add: f-env-insert-def split: if-splits; arith*)
 next
 case (*TyArr A B*)
 then show ?case by *simp*
 next
 case (*TyAll A*)
 have *shift0*:
 $\forall D. f\text{-sem } (ty\text{-shift } 0 \ S) \ (ty\text{-env-cons } D \ \varrho) =$
 $f\text{-sem } S \ \varrho$
 using *f-sem-ty-shift[of 0 S] f-env-insert-zero* by *simp*
 have *body-eq*:
 $\forall D. f\text{-sem } (ty\text{-subst } (Suc \ k) \ (ty\text{-shift } 0 \ S) \ A)$
 $(ty\text{-env-cons } D \ \varrho) =$
 $f\text{-sem } A$

```

      (ty-env-cons D (f-env-insert k (f-sem S ρ) ρ))
    using TyAll(1)
    by (simp add: shift0 f-sem-ty-shift f-env-insert-zero
      f-env-insert-suc-cons)
  show ?case
    using body-eq
    by (simp add: f-all-sem-def)
qed

```

definition *f-tenv-cons* :: *ulam* $\Rightarrow$ (*nat* $\Rightarrow$ *ulam*) $\Rightarrow$ *nat* $\Rightarrow$ *ulam*

where

```

  f-tenv-cons N η n =
    (case n of 0  $\Rightarrow$  N | Suc k  $\Rightarrow$  η k)

```

definition *f-tenv-shift* :: (*nat* $\Rightarrow$ *ulam*) $\Rightarrow$ *nat* $\Rightarrow$ *ulam*

where

```

  f-tenv-shift η n =
    (case n of 0  $\Rightarrow$  UVar 0 | Suc k  $\Rightarrow$  u-shift 0 (η k))

```

definition *f-env-subst* :: *nat* $\Rightarrow$ *ulam* $\Rightarrow$ (*nat* $\Rightarrow$ *ulam*) $\Rightarrow$ *nat* $\Rightarrow$ *ulam*

where

```

  f-env-subst k N η n = u-subst k N (η n)

```

lemma *f-env-subst-shift*:

```

  f-env-subst (Suc k) (u-shift 0 N) (f-tenv-shift η) =
    f-tenv-shift (f-env-subst k N η)

```

proof (*rule ext*)

```

  fix n
  show f-env-subst (Suc k) (u-shift 0 N) (f-tenv-shift η) n =
    f-tenv-shift (f-env-subst k N η) n
  proof (induct n)
    case 0
    then show ?case
      by (simp add: f-env-subst-def f-tenv-shift-def)
  next
    case (Suc n)
    then show ?case
      by (simp add: f-env-subst-def f-tenv-shift-def
        u-shift-subst-comm)
  qed
qed

```

lemma *f-env-subst-shift0*:

```

  f-env-subst 0 N (f-tenv-shift η) =
    f-tenv-cons N η

```

proof (*rule ext*)

```

fix n
show  $f\text{-env}\text{-subst } 0 \ N \ (f\text{-tenv}\text{-shift } \eta) \ n =$ 
   $f\text{-tenv}\text{-cons } N \ \eta \ n$ 
proof (induct n)
  case 0
  then show ?case
    by (simp add: f-env-subst-def f-tenv-shift-def f-tenv-cons-def
       $u\text{-subst0}\text{-def}$ )
  next
  case (Suc n)
  then show ?case
    by (simp add: f-env-subst-def f-tenv-shift-def f-tenv-cons-def
       $u\text{-subst0}\text{-def } u\text{-subst}\text{-shift}$ )
qed
qed

```

```

fun  $f\text{-interp} :: (\text{nat} \Rightarrow \text{ulam}) \Rightarrow \text{dtm} \Rightarrow \text{ulam}$  where
   $f\text{-interp } \eta \ (TmVar \ n) = \eta \ n$ 
|  $f\text{-interp } \eta \ (TmApp \ M \ N) =$ 
   $UApp \ (f\text{-interp } \eta \ M) \ (f\text{-interp } \eta \ N)$ 
|  $f\text{-interp } \eta \ (TmLam \ A \ M) =$ 
   $ULam \ (f\text{-interp } (f\text{-tenv}\text{-shift } \eta) \ M)$ 
|  $f\text{-interp } \eta \ (TmTAbs \ M) = f\text{-interp } \eta \ M$ 
|  $f\text{-interp } \eta \ (TmTApp \ M \ A) = f\text{-interp } \eta \ M$ 

```

```

lemma  $f\text{-interp}\text{-subst}$ :
   $u\text{-subst } k \ N \ (f\text{-interp } \eta \ M) =$ 
   $f\text{-interp } (f\text{-env}\text{-subst } k \ N \ \eta) \ M$ 
proof (induct M arbitrary: k N η rule: dtm.induct)
  case (TmVar n)
  then show ?case by (simp add: f-env-subst-def)
next
  case (TmApp M P)
  then show ?case by simp
next
  case (TmLam A M)
  then show ?case
    by (simp add: f-env-subst-shift)
next
  case (TmTAbs M)
  then show ?case by simp
next
  case (TmTApp M A)
  then show ?case by simp
qed

```

```

lemma  $f\text{-tenv}\text{-shift}\text{-id}$ :
   $f\text{-tenv}\text{-shift } (\lambda n. \ UVar \ n) = (\lambda n. \ UVar \ n)$ 
proof (rule ext)

```

```

fix x :: nat
show f-tenv-shift (λn. UVar n) x = UVar x
proof (induct x)
  case 0
  then show ?case by (simp add: f-tenv-shift-def)
next
  case (Suc n)
  then show ?case by (simp add: f-tenv-shift-def)
qed
qed

```

```

lemma f-interp-id:
  f-interp (λn. UVar n) M = f-erase M
by (induct M rule: dtm.induct)
  (simp-all add: f-tenv-shift-id)

```

```

lemma f-interp-subst0-shift:
  u-subst0 N (f-interp (f-tenv-shift η) M) =
    f-interp (f-tenv-cons N η) M
using f-interp-subst[of 0 N f-tenv-shift η M]
  f-env-subst-shift0
by (simp add: u-subst0-def)

```

```

definition f-tenv-ok ::
  dty list ⇒ (nat ⇒ ulam) ⇒
  (nat ⇒ ulam set) ⇒ bool
where
  f-tenv-ok Γ η ρ ⟷
    (∀ n A. ctx-mem Γ n A ⟶
      η n ∈ f-sem A ρ)

```

```

lemma f-ctx-mem-mapE:
  ctx-mem (map (ty-shift 0) Γ) n B ⟶
    (∃ A. ctx-mem Γ n A ∧ B = ty-shift 0 A)
proof (induct Γ arbitrary: n B)
  case Nil
  then show ?case by (auto elim: ctx-mem.cases)
next
  case (Cons A Γ)
  then show ?case
  proof (intro impI)
    assume h: ctx-mem (map (ty-shift 0) (A # Γ)) n B
    have h': ctx-mem (ty-shift 0 A # map (ty-shift 0) Γ) n B
      using h by simp
    then show ∃ A'. ctx-mem (A # Γ) n A' ∧
      B = ty-shift 0 A'
  proof (cases rule: ctx-mem.cases)
    case ctx-zero
    then show ?thesis

```

```

      by (auto intro: ctx-mem.ctx-zero)
    next
      case ctx-suc
      have h'':
        ctx-mem (map (ty-shift 0)  $\Gamma$ ) (n - 1) B
      using ctx-suc(1) ctx-suc(2) by simp
      have ex:
         $\exists A'. \text{ctx-mem } \Gamma (n - 1) A' \wedge$ 
         $B = \text{ty-shift } 0 A'$ 
      using Cons(1)[of n - 1 B] h'' by blast
      have ex':
         $\exists A'. \text{ctx-mem } (A \# \Gamma) (\text{Suc } (n - 1)) A' \wedge$ 
         $B = \text{ty-shift } 0 A'$ 
      using ex by (blast intro: ctx-mem.ctx-suc)
      from ex' ctx-suc(1) show ?thesis by simp
    qed
  qed
qed

lemma f-tenv-ok-cons:
  assumes hctx: f-tenv-ok  $\Gamma \eta \varrho$ 
  and hval:  $N \in \text{f-sem } A \varrho$ 
  shows f-tenv-ok (A #  $\Gamma$ ) (f-tenv-cons N  $\eta$ )  $\varrho$ 
  using assms
  unfolding f-tenv-ok-def
  by (auto simp add: f-tenv-cons-def elim: ctx-mem.cases)

lemma f-tenv-ok-ty-shift:
  assumes hctx: f-tenv-ok  $\Gamma \eta \varrho$ 
  and henv: f-env-ok  $\varrho$ 
  and hcan: ucandidate C
  shows f-tenv-ok (map (ty-shift 0)  $\Gamma$ )  $\eta$ 
  (ty-env-cons C  $\varrho$ )
proof (unfold f-tenv-ok-def, intro allI allI impI)
  fix n B
  assume memB: ctx-mem (map (ty-shift 0)  $\Gamma$ ) n B
  have ex:
     $\exists A. \text{ctx-mem } \Gamma n A \wedge B = \text{ty-shift } 0 A$ 
  using f-ctx-mem-mapE[of  $\Gamma$  n B] memB by blast
  have sh:
     $\forall A. \text{f-sem } (\text{ty-shift } 0 A) (\text{ty-env-cons } C \varrho) =$ 
     $\text{f-sem } A \varrho$ 
  using f-sem-ty-shift[of 0] f-env-insert-zero by simp
  have sem:
     $\forall A. \text{ctx-mem } \Gamma n A \longrightarrow$ 
     $\eta n \in \text{f-sem } (\text{ty-shift } 0 A) (\text{ty-env-cons } C \varrho)$ 
  using hctx sh unfolding f-tenv-ok-def by blast
  from ex sem show  $\eta n \in \text{f-sem } B (\text{ty-env-cons } C \varrho)$ 
  by blast

```

qed

definition $f\text{-prop} ::$

$dtv\ list \Rightarrow dtm \Rightarrow dtv \Rightarrow bool$

where

$f\text{-prop}\ \Gamma\ M\ A \longleftrightarrow$

$(\forall\ \varrho\ \eta.$

$f\text{-env-ok}\ \varrho \longrightarrow$

$f\text{-tenv-ok}\ \Gamma\ \eta\ \varrho \longrightarrow$

$f\text{-interp}\ \eta\ M \in f\text{-sem}\ A\ \varrho)$

lemma $f\text{-fundamental}:$

$typing\ k\ \Gamma\ M\ A \implies f\text{-prop}\ \Gamma\ M\ A$

proof (*induct rule: typing.induct*)

case $ty\text{-}var$

then show $?case$

unfolding $f\text{-prop-def}\ f\text{-interp.simps}\ f\text{-sem.simps}\ f\text{-tenv-ok-def}$

by $blast$

next

case $ty\text{-}app$

then show $?case$

unfolding $f\text{-prop-def}\ f\text{-interp.simps}\ f\text{-sem.simps}\ uarr\text{-def}$

by $blast$

next

case $(ty\text{-}lam\ k0\ A0\ G0\ M0\ B0)$

show $?case$

unfolding $f\text{-prop-def}\ f\text{-interp.simps}\ f\text{-sem.simps}$

proof (*intro allI allI impI impI*)

fix $\varrho\ \eta$

assume $env: f\text{-env-ok}\ \varrho$

assume $ctx: f\text{-tenv-ok}\ G0\ \eta\ \varrho$

have $Ccan: ucandidate\ (f\text{-sem}\ A0\ \varrho)$

using $f\text{-sem-candidate}[of\ \varrho\ A0]\ env$ **by** $blast$

have $Dcan: ucandidate\ (f\text{-sem}\ B0\ \varrho)$

using $f\text{-sem-candidate}[of\ \varrho\ B0]\ env$ **by** $blast$

have $body:$

$\forall\ s. s \in f\text{-sem}\ A0\ \varrho \longrightarrow$

$u\text{-subst0}\ s\ (f\text{-interp}\ (f\text{-tenv-shift}\ \eta)\ M0) \in f\text{-sem}\ B0\ \varrho$

proof (*intro allI impI*)

fix s

assume $sA: s \in f\text{-sem}\ A0\ \varrho$

have $ctxs:$

$f\text{-tenv-ok}\ (A0\ \# \ G0)\ (f\text{-tenv-cons}\ s\ \eta)\ \varrho$

using $f\text{-tenv-ok-cons}[OF\ ctx\ sA]$.

have $ih:$

$f\text{-interp}\ (f\text{-tenv-cons}\ s\ \eta)\ M0 \in f\text{-sem}\ B0\ \varrho$

using $ty\text{-}lam(3)\ env\ ctxs$

unfolding $f\text{-prop-def}$ **by** $blast$

have $eq:$

```

    u-subst0 s (f-interp (f-tenv-shift  $\eta$ ) M0) =
      f-interp (f-tenv-cons s  $\eta$ ) M0
    using f-interp-subst0-shift by blast
  from ih show
    u-subst0 s (f-interp (f-tenv-shift  $\eta$ ) M0)  $\in$  f-sem B0  $\varrho$ 
    using eq by simp
qed
have abs:
  ULam (f-interp (f-tenv-shift  $\eta$ ) M0)  $\in$ 
    uarr (f-sem A0  $\varrho$ ) (f-sem B0  $\varrho$ )
  using Ccan Dcan body by (rule uabs-RED)
then show ULam (f-interp (f-tenv-shift  $\eta$ ) M0)  $\in$ 
  uarr (f-sem A0  $\varrho$ ) (f-sem B0  $\varrho$ ) .
qed
next
case (ty-tabs k0 G0 M0 B0)
show ?case
  unfolding f-prop-def
proof (intro allI allI impI impI)
  fix  $\varrho$   $\eta$ 
  assume env: f-env-ok  $\varrho$ 
  assume ctx: f-tenv-ok G0  $\eta$   $\varrho$ 
  have all:
     $\forall C. \text{ucandidate } C \longrightarrow$ 
      f-interp  $\eta$  M0  $\in$  f-sem B0 (ty-env-cons C  $\varrho$ )
  proof (intro allI impI)
    fix C
    assume Ccan: ucandidate C
    have envC: f-env-ok (ty-env-cons C  $\varrho$ )
      using f-env-ok-cons[OF env Ccan] .
    have ctxC:
      f-tenv-ok (map (ty-shift 0) G0)  $\eta$ 
      (ty-env-cons C  $\varrho$ )
      using f-tenv-ok-ty-shift[OF ctx env Ccan] .
    have ih:
      f-interp  $\eta$  M0  $\in$  f-sem B0 (ty-env-cons C  $\varrho$ )
      using ty-tabs(2) envC ctxC
      unfolding f-prop-def by blast
    from ih show f-interp  $\eta$  M0  $\in$  f-sem B0 (ty-env-cons C  $\varrho$ )
      by assumption
  qed
qed
show f-interp  $\eta$  (TmTAbs M0)  $\in$  f-sem (TyAll B0)  $\varrho$ 
  using all unfolding f-interp.simps f-sem.simps f-all-sem-def by blast
qed
next
case (ty-tapp k0 G0 M0 B0 A0)
show ?case
  unfolding f-prop-def
proof (intro allI allI impI impI)

```

```

fix  $\varrho$   $\eta$ 
assume env: f-env-ok  $\varrho$ 
assume ctx: f-tenv-ok  $G0$   $\eta$   $\varrho$ 
have ih:
  f-interp  $\eta$   $M0 \in f\text{-sem } (TyAll\ B0) \ \varrho$ 
  using ty-tapp(2) env ctx
  unfolding f-prop-def by blast
have Acan: ucandidate (f-sem  $A0$   $\varrho$ )
  using f-sem-candidate[of  $\varrho$   $A0$ ] env by blast
have body:
  f-interp  $\eta$   $M0 \in$ 
    f-sem  $B0$  (ty-env-cons (f-sem  $A0$   $\varrho$ )  $\varrho$ )
  using ih Acan unfolding f-sem.simps f-all-sem-def by blast
have sem-eq:
  f-sem (ty-subst0  $A0$   $B0$ )  $\varrho =$ 
    f-sem  $B0$  (ty-env-cons (f-sem  $A0$   $\varrho$ )  $\varrho$ )
  using f-sem-ty-subst[of 0  $A0$   $B0$   $\varrho$ ] f-env-insert-zero
  by (simp add: ty-subst0-def)
show f-interp  $\eta$  (TmTApp  $M0$   $A0$ )  $\in$ 
  f-sem (ty-subst0  $A0$   $B0$ )  $\varrho$ 
  using body sem-eq by simp
qed
qed

```

```

lemma f-identity-tenv-ok:
  f-tenv-ok  $\Gamma$  ( $\lambda n. \ UVar\ n$ ) ( $\lambda -. \ uSN\text{-set}$ )
proof (unfold f-tenv-ok-def, intro allI allI impI)
  fix  $n$   $A$ 
  assume ctx-mem  $\Gamma$   $n$   $A$ 
  have can:
    ucandidate (f-sem  $A$  ( $\lambda -. \ uSN\text{-set}$ ))
    using f-sem-candidate[of ( $\lambda -. \ uSN\text{-set}$ )  $A$ ]
    by (simp add: f-env-ok-def uSN-set-candidate)
  from uvar-mem-candidate[OF can]
  show  $UVar\ n \in f\text{-sem } A$  ( $\lambda -. \ uSN\text{-set}$ ) .
qed

```

```

lemma f-welltyped-fundamental:
  assumes typing  $k$   $\Gamma$   $M$   $A$ 
  shows uSN (f-erase  $M$ )
proof -
  have env: f-env-ok ( $\lambda -. \ uSN\text{-set}$ )
    unfolding f-env-ok-def
    by (simp add: uSN-set-candidate)
  have ctx:
    f-tenv-ok  $\Gamma$  ( $\lambda n. \ UVar\ n$ ) ( $\lambda -. \ uSN\text{-set}$ )
    using f-identity-tenv-ok .
  have sem:
    f-interp ( $\lambda n. \ UVar\ n$ )  $M \in$ 

```

```

    f-sem A (λ-. uSN-set)
  using f-fundamental[OF assms] env ctx
  unfolding f-prop-def by blast
  have semM:
    f-erase M ∈ f-sem A (λ-. uSN-set)
  using sem f-interp-id by simp
  have can:
    ucandidate (f-sem A (λ-. uSN-set))
  using f-sem-candidate[of (λ-. uSN-set) A] env by blast
  from can semM show uSN (f-erase M)
  unfolding ucandidate-def uCR1-def by blast
qed

theorem strong-normalization:
  assumes typing k Γ M A
  shows SN M
  using fSN-reflect[OF f-welltyped-fundamental[OF assms]] .

end

```

References

- [1] T. Altenkirch. A formalization of the strong normalization proof for system f in lego. In *Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer, 1993. DOI: <https://doi.org/10.1007/BFB0037095>.
- [2] S. Berghofer. Strong normalization for simply-typed lambda calculus. Isabelle/HOL library theory HOL/Proofs/Lambda/StrongNorm.thy, 2000. <https://isabelle.in.tum.de/library/HOL/HOL-Proofs-Lambda/document.pdf>.
- [3] S. Berghofer. Poplmark challenge via de bruijn indices. *Archive of Formal Proofs*, August 2007. <https://isa-afp.org/entries/POPLmark-deBruijn.html>, Formal proof development.
- [4] K. Donnelly and H. Xi. A formalization of strong normalization for simply-typed lambda-calculus and system f. *Electronic Notes in Theoretical Computer Science*, 174(5):109–125, 2007. DOI: <https://doi.org/10.1016/j.entcs.2007.01.021>.
- [5] J.-Y. Girard. Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur. *Thèse de doctorat, Université Paris VII*, 1972.
- [6] A. Popescu, E. L. Gunter, and C. J. Osborn. Strong normalization for system f by hoas on top of foas. In *2010 25th Annual IEEE Symposium*

on Logic in Computer Science, pages 31–40. IEEE, 2010. DOI: <https://doi.org/10.1109/LICS.2010.48>.