# POPLmark Challenges

Jevgenijs Sallinens

June/November 2005

## Contents

# 1 Introduction

Here are addressed POPLmark challenge parts 1a,2a,3 in the nameless framework of De Bruijn indices (without records types and evaluation contexts). Proofs are verified with the proof assistant Coq,version 8.0. As the major result, there is given explicit operator to perform reduction step for the third subtask of the third part, with direct proof that typing is preserved with this reduction. Some new results (inversions of typing rules) are presented to simplify proofs for arbitrary (possible non-empty) environments. So, the part 3 is resolved in the sense, that there are presented programs to perform reduction and these programs are verified to be compatible with subtyping and typing relations. Also others decidable relations, such as well-formedness conditions, are given in the form of computable operators to the type of booleans, so providing verified basis for programming of all related aspects (except connected with computations for undecidable subtyping and typing relations). The work in progress also shows compatibilty with name carrying formalization and possibilty to provide verified programs to translate from this formalization to the form presented in this development. Solutions for part1a are given for environments as lists of types and there is presented methodology to reuse these results for part2a, where environments are lists of types marked with a kind of associated binder (type or term). Since permutations of environments are problematic with the nameless approach, there is no complete analogy with paper proofs (for weakening lemmas, for example).

Results for operations with natural numbers, lists and types are given without proofs in this document (see separate library file for details of proofs). Used notations are listed in the last section. Some notations are simplified for typeset version of this document, so that there are cases when same notation used for different purposes. Due to some difficulties with Coq notational mechanism there are used some nonstandard notations, like for a type $T$ and a term $t$ we write $T \;.\!\!\to t$ instead of something like $\lambda T.t$. Also notations like $(X$ [->] $t)$ $s$ are used to denote the result of substitution of a term $t$ for a De Bruijn index $X$ in a term $s$.

Initial version of this development was based on the submission of Jerome Vouillon, Except already mentioned features (computability of all relations and separation of environments for parts 1 and 2), major differences from Jerome's solution are related with attempts to clarify and separate roles of involved components (such like natural numbers, lists, operations with De Bruijn indices). Also, there are given different, more explicit (but equivalent) definitions of subtyping and typing relations. Those interested for detailed information for goals resolved after every step in a proof, could browse files by using program CoqIDE from Coq 8.0 distribution (file tactics.v should be compiled with the command 'coqc tactics', in order to create file tactics.vo before browsing the file main.v).

# 2 Natural Numbers

Inductive nat : Set := | O : nat | S : nat -> nat.

Order relation for natural numbers.

Fixpoint $nat\_le$ $(n\ m{:}nat)$ $\{struct\ n\}{:}bool{:=}$
$match\ n,\ m\ with$
$|\ O,\ \_ \Rightarrow true$
$|\ S\ \_,\ O \Rightarrow false$
$|\ S\ n1,\ S\ m1 \Rightarrow nat\_le\ n1\ m1$
$end.$

Infix "$\leq$" := $nat\_le$.

Equlaity of natural numbers.

Definition $n\_eq\ (n\ m\ {:}nat){:=}\ (n \leq m)\ \&\&\ (m \leq n)$.

Infix "$==$" := $n\_eq$.

Axiom $n\_eq\_1$: $\forall\ x\ y,\ x == y \rightarrow x = y$.

Axiom $n\_eq\_2$: $\forall\ x\ y,\ x == y \rightarrow y = x$.

Axiom $n\_eq\_le$: $\forall\ n\ m,\ n == m \rightarrow n \leq m$.

Axiom $n\_eq\_refl$: $\forall\ n,\ n == n$.

Hint $Resolve\ n\_eq\_le\ n\_eq\_refl$.

Axiom $N\_Decide$: $\forall\ (x\ y{:}nat)(P{:}Prop),\ ((S\ x \leq y) \rightarrow P) \rightarrow ((y \leq x) \rightarrow P) \rightarrow P$.

Definition $S\_Next\ (Y\ X{:}nat){:=}\ if\ (nat\_le\ Y\ X)\ then\ S\ X\ else\ X$.

Definition $S\_Pred\ (Y\ X{:}nat){:=}\ if\ (nat\_le\ Y\ X)\ then\ Y\ else\ (pred\ Y)$.

Axiom $S\_Next\_le{:}\forall\ (Y\ X{:}nat),\ Y \leq X \rightarrow S\_Next\ Y\ X = S\ X$.

Axiom $S\_Next\_gt{:}\forall\ (Y\ X{:}nat),\ S\ X \leq Y \rightarrow S\_Next\ Y\ X = X$.

Axiom $S\_Pred\_le{:}\forall\ (Y\ X{:}nat),\ X \leq Y \rightarrow (Y == X \rightarrow false) \rightarrow S\ (S\_Pred\ Y\ X) = Y$.

Axiom $S\_Pred\_gt{:}\forall\ (Y\ X{:}nat),\ S\ Y \leq X \rightarrow S\_Pred\ Y\ X = Y$.

# 3 Lists

Inductive list : Set := | nil : list | cons : A -> list -> list.

Infix "::" := cons.

Section *Lists.*
Variable *A*:*Set.*

Fixpoint *len* (*e*:*list A*){*struct e*}:*nat*:=
*match e with*
| *nil* ⇒ 0
| *T' :: e'* ⇒ *S* (*len e'*)
*end.*

Well-formed indices of lists.

Fixpoint *wfi* (*e*:*list A*) (*X*:*nat*) {*struct e*}:*bool*:=
*match e*, *X with*
| *nil* , _ ⇒ *false*
| *T :: e'*,*O* ⇒ *true*
| *T :: e'*,*S X'* ⇒ *wfi e' X'*
*end.*

Axiom *wfi_app_cons*: ∀ *e e0 T*, *wfi* (*e* ++ *T :: e0*) (*len e*).

Axiom *wfi_app*: ∀ *e X e0 T0*,
*wfi* (*e* ++ *e0*) *X* → *wfi* (*e* ++ *T0 :: e0*) (*S_Next* (*len e*) *X*).

Axiom *wfi_app_1*: ∀ *e X e0 T0*,
*wfi* (*e* ++ *T0 :: e0*) (*S_Next* (*len e*) *X*)->*wfi* (*e* ++ *e0*) *X*.

Axiom *wfi_app_2*:
∀ *e X e0 T0*, ((*X* == *len e*) → *false*)->
*wfi* (*e* ++ *T0 :: e0*) *X* → *wfi* (*e* ++ *e0*) (*S_Pred X* (*len e*)).

Fixpoint *lel* (*e1 e2*:*list A*) {*struct e1*}:*bool*:=
*match e1*, *e2 with*
| *nil* , _ ⇒ *true*
| *T :: e1'* , *nil* ⇒ *false*
| *T1 :: e1'*, *T2 :: e2'* ⇒ *lel e1' e2'*
*end.*

Axiom *lel_refl*: ∀ *e*, *lel e e*.

Hint *Resolve lel_refl.*

Axiom *lel_trans*: ∀ *e1 e2 e3*, *lel e1 e2* →*lel e2 e3* →*lel e1 e3*.

Axiom *lel_cons* :∀ *e1 e2 M*, *lel e1 e2* → *lel e1* (*M :: e2*).

Axiom *lel_wfi*: ∀ *e e' X*, *lel e e'* → *wfi e X* → *wfi e' X*.

Axiom $app\_lel$: $\forall$ $e$ $e1$ $e2$, $lel$ $e1$ $e2$ $\rightarrow$ $lel$ $(e$ ++ $e1)$ $(e$ ++ $e2)$.

Axiom $app\_lel\_cons$: $\forall$ $e1$ $e$ $a$,$lel$ $(e1$ ++ $e)$ $(a$ :: $e1$ ++ $e)$.

Axiom $app\_lel\_2$: $\forall$ $e1$ $e2$ $e$, $lel$ $e1$ $e2$ $\rightarrow$ $lel$ $(e1$ ++ $e)$ $(e2$ ++ $e)$.

Elements within lists.

Inductive $inl$ :$(list$ $A)$ $\rightarrow nat \rightarrow A$ $\rightarrow Prop$:=
  | $inl\_0$:$\forall$ $e$ $a$, $inl$ $(a$ :: $e)$ $0$ $a$
  | $inl\_S$:$\forall$ $e$ $X$ $a$ $b$, $inl$ $e$ $X$ $a$ $\rightarrow$ $inl$ $(b$ :: $e)$ $(S$ $X)$ $a$.

Axiom $inl\_wf$: $\forall$ $e$ $X$ $a$, $inl$ $e$ $X$ $a$ $\rightarrow$ $wfi$ $e$ $X$.

Axiom $inl\_cons$:$\forall$ $e$ $X$ $T$ $a$, $inl$ $(a$ :: $e)$ $(S$ $X)$ $T$ $\rightarrow$ $inl$ $e$ $X$ $T$.

Axiom $inl\_inv$:$\forall$ $e$ $X$ $T1$ $T2$,
  $inl$ $e$ $X$ $T1$ $\rightarrow$ $inl$ $e$ $X$ $T2$ $\rightarrow$ $T1$ = $T2$.

Axiom $inl\_ex$: $\forall$ $e$ $X$, $wfi$ $e$ $X$ $\rightarrow \exists$ $a$, $inl$ $e$ $X$ $a$.

Axiom $inl\_app\_neq$: $\forall$ $e$ $X$ $e0$ $M$ $N$ $T$,
  $(X$ == $len$ $e$ $\rightarrow false)$ $\rightarrow$
  $inl$ $(e$ ++ $M$ :: $e0)$ $X$ $T$ $\rightarrow$ $inl$ $(e$ ++ $N$ :: $e0)$ $X$ $T$.

Axiom $inl\_app\_cons$: $\forall$ $e$ $e0$ $T$, $inl$ $(e$ ++ $T$ :: $e0)$ $(len$ $e)$ $T$.

Axiom $inl\_app\_1$: $\forall$ $e$ $X$ $e0$ $T0$ $T$,
  $inl$ $(e$ ++ $T0$ :: $e0)$ $(S\_Next$ $(len$ $e)$ $X)$ $T$ $\rightarrow inl$ $(e$ ++ $e0)$ $X$ $T$.

Axiom $inl\_app\_2$: $\forall$ $e$ $X$ $e0$ $T0$ $T$,
  $((X$ == $len$ $e)$ $\rightarrow$ $false)$->
  $inl$ $(e$ ++ $T0$ :: $e0)$ $X$ $T$ $\rightarrow inl$ $(e$ ++ $e0)$ $(S\_Pred$ $X$ $(len$ $e))$ $T$.

Fixpoint $head$ $(e$:$list$ $A)$ $(X$:$nat)$ $\{struct$ $e\}$:$list$ $A$:=
  $match$ $e$, $X$ $with$
  | $nil$ , _ $\Rightarrow$ $nil$
  | $T'$ :: $e'$, $0$ $\Rightarrow$ $nil$
  | $T'$ :: $e'$, $S$ $X'$ $\Rightarrow$ $T'$ :: $head$ $e'$ $X'$
$end$.

Axiom $head\_len$: $\forall$ $e$ $X$ , $wfi$ $e$ $X$ $\rightarrow len$ $(head$ $e$ $X)$ = $X$.

Fixpoint $tail$ $(e$:$list$ $A)$ $(X$:$nat)$ $\{struct$ $e\}$:$list$ $A$:=
  $match$ $e$, $X$ $with$
  | $nil$ , _ $\Rightarrow$ $nil$
  | $T'$ :: $e'$, $0$ $\Rightarrow$ $e'$
  | $T'$ :: $e'$, $S$ $X'$ $\Rightarrow$ $tail$ $e'$ $X'$
$end$.

Axiom $decomp\_list$: $\forall$ $e$ $X$ $U$, $inl$ $e$ $X$ $U$ $\rightarrow$
  $e$ = $head$ $e$ $X$ ++ $U$::$tail$ $e$ $X$.

Applying functions to lists

```
Section Maps.
Variable f:nat → A → A.

Fixpoint list_map (e:list A) {struct e}:list A:=
    match e with
    | nil ⇒ nil
    | T' :: e' ⇒ f (len e') T' :: list_map e'
    end.

Axiom len_list_map:∀ e, len (list_map e) =len e.

Axiom lel_list_map:∀ e, lel e (list_map e).

Axiom inl_map_app_le: ∀ e e0 X T,
    len e ≤ X → inl (e ++ e0) X T → inl (list_map e ++ e0 ) X T.

Axiom inl_map_cons_le:∀ e2 e1 X T1 T2 T0,
      inl (e2 ++ e1) X T1 → inl (list_map e2 ++ T0 :: e1) (S X) T2 →
      len e2 ≤ X → T1 = T2.

Axiom inl_map_cons_gt:∀ e2 e1 X T1 T2 T0,
    inl (e2 ++ e1) X T1 → inl (list_map e2 ++ T0 :: e1) X T2 →
    S X ≤ len e2 → T2 = f (len e2 - S X) T1.
End Maps.
```

Mixed lists to be used as environmets in part2a.

```
Inductive list2:Set:=
    | nil2:list2
    | cons1:A → list2 → list2
    | cons2:A → list2 → list2.

Infix ":;" := cons1.
Infix ";:" := cons2.

Fixpoint lst (e:list2) {struct e}:list A:=
    match e with
    | nil2 ⇒ nil
    | T :; e' ⇒ lst e'
    | T ;: e' ⇒ T :: (lst e')
    end.

Notation "[ x ]" := (lst x).
```

Fixpoint $len1\ (e{:}list2)\{struct\ e\}{:}nat{:}=$
$match\ e\ with$
$|\ nil2 \Rightarrow 0$
$|\ T\ {:};\ e' \Rightarrow S\ (len1\ e')$
$|\ T\ ;{:}\ e' \Rightarrow len1\ e'$
$end.$

Concatenation of lists

Fixpoint $list2\_app\ (e{:}list2)\ (e0{:}list2)\{struct\ e\}{:}list2{:}=$
$match\ e\ with$
$|\ nil2 \Rightarrow e0$
$|\ T'\ {:};\ e' \Rightarrow T'\ {:};\ (list2\_app\ e'\ e0)$
$|\ T'\ ;{:}\ e' \Rightarrow T'\ ;{:}\ (list2\_app\ e'\ e0)$
$end.$

Infix "$++$" $:=$ $list2\_app.$

Axiom $app2\_inv$: $\forall\ (e1\ \ e2\ :list2)\ U\ \ V\ \ u\ \ v,$
$e1\ ++\ U\ {:};\ u = e2\ ++\ V\ {:};\ v \rightarrow len1\ \ e1\ == len1\ \ e2 \rightarrow U = V.$

Axiom $list2list\_app$ : $\forall\ e\ \ e0, [e\ ++\ e0] = [\mathrm{e}]\ ++\ [\mathrm{e0}].$

Axiom $list2list\_cons$ : $\forall\ e\ \ M,\ [M\ ;{:}\ e] = M\ {:}{:}\ ([e]).$

Elements within lists.

Inductive $ine\ :list2 \rightarrow nat \rightarrow A \rightarrow Prop{:}=$
$|\ ine\_0{:}\forall\ e\ \ a,\ ine\ (a\ {:};\ e)\ 0\ \ a$
$|\ ine\_S1{:}\forall\ e\ \ X\ \ a\ \ b,\ ine\ \ e\ \ X\ \ a \rightarrow ine\ (b\ {:};\ e)\ (S\ \ X)\ \ a$
$|\ ine\_S2{:}\forall\ e\ \ X\ \ a\ \ b,\ ine\ \ e\ \ X\ \ a \rightarrow ine\ (b\ ;{:}\ e)\ \ X\ \ a.$

Axiom $ine\_cons1{:}\forall\ e\ \ X\ \ T\ \ a,\ ine\ (a\ {:};\ e)\ (S\ \ X)\ \ T \rightarrow ine\ \ e\ \ X\ \ T.$

Axiom $ine\_cons2{:}\forall\ e\ \ X\ \ T\ \ a,\ ine\ (a\ ;{:}\ e)\ \ X\ \ T \rightarrow ine\ \ e\ \ X\ \ T.$

Axiom $ine\_app\_cons$: $\forall\ e\ \ e0\ \ T,\ ine\ (e\ ++\ T\ {:};\ e0)\ (len1\ \ e)\ \ T.$

Axiom $ine\_app\_cons\_2$: $\forall\ e\ \ e0\ \ N\ \ X\ \ T,$
$ine\ (e\ ++\ N\ ;{:}\ e0)\ \ X\ \ T \rightarrow ine\ (e\ ++\ e0)\ \ X\ \ T.$

Axiom $ine\_cons$: $\forall\ e\ \ e0\ \ N\ \ X\ \ T,$
$ine\ (e\ ++\ e0)\ \ X\ \ T \rightarrow ine\ (e\ ++\ N\ ;{:}\ e0)\ \ X\ \ T.$

Axiom $ine\_app$: $\forall\ e\ \ X\ \ e0\ \ T0\ \ T,$
$ine\ (e\ ++\ e0)\ \ X\ \ T \rightarrow ine\ (e\ ++\ T0\ {:};\ e0)\ (S\_Next\ (len1\ \ e)\ \ X)\ \ T.$

Axiom $ine\_app1$: $\forall\ e\ \ X\ \ e0\ \ T0\ \ T,$
$ine\ (e\ ++\ T0\ {:};\ e0)\ (S\_Next\ (len1\ \ e)\ \ X)\ \ T \rightarrow ine\ (e\ ++\ e0)\ \ X\ \ T.$

Axiom $ine\_app\_2$: $\forall$ $e$ $X$ $e0$ $T0$ $T$,
$((X == len1\ e) \rightarrow false)$->
$ine\ (e ++ T0 :; e0)\ X\ T \rightarrow ine\ (e ++ e0)\ (S\_Pred\ X\ (len1\ e))\ T$.

Fixpoint $head1$ $(e{:}list2)$ $(X{:}nat)$ $\{struct\ e\}{:}list2{:=}$
$match\ e,\ X\ with$
$|\ nil2\ ,\ \_\ \Rightarrow nil2$
$|\ T'\ :;\ e',\ 0 \Rightarrow nil2$
$|\ T'\ :;\ e',\ S\ X' \Rightarrow T'\ :;\ head1\ e'\ X'$
$|\ T'\ ;:\ e',\ \_\ \Rightarrow T'\ ;:\ head1\ e'\ X$
$end.$

Fixpoint $tail1$ $(e{:}list2)$ $(X{:}nat)$ $\{struct\ e\}{:}list2{:=}$
$match\ e,\ X\ with$
$|\ nil2\ ,\ \_\ \Rightarrow nil2$
$|\ T'\ :;\ e',\ 0 \Rightarrow e'$
$|\ T'\ :;\ e',\ S\ X' \Rightarrow tail1\ e'\ X'$
$|\ T'\ ;:\ e',\ \_\ \Rightarrow tail1\ e'\ X$
$end.$

Axiom $decomp\_list2$: $\forall$ $e$ $X$ $U$,
$ine\ e\ X\ U \rightarrow e = head1\ e\ X{++}\ U{:;}tail1\ e\ X$.

Axiom $head\_len1$: $\forall$ $e$ $X$ $U$, $ine\ e\ X\ U \rightarrow len1\ (head1\ e\ X) = X$.

Axiom $head1\_prop\_0$: $\forall$ $e$ $e0$ $T$,
$len\ ([head1\ (e ++ T :; e0)\ (len1\ e)]) = len\ ([e])$.

Axiom $head1\_prop\_1$: $\forall$ $e$ $e0$ $X$,
$len1\ e \leq X \rightarrow len\ ([e]) \leq len\ ([head1 (e ++ e0)\ X])$ .

Axiom $head1\_prop\_2$: $\forall$ $e$ $e0$ $X$,
$S\ X \leq len1\ e \rightarrow len\ ([head1 (e ++ e0)\ X]) \leq len\ ([e])$.

Axiom $head1\_cons1$:
$\forall$ $e$ $X$ $e0$ $T$,
$len\ ([head1\ (e ++ T :; e0)\ (S\_Next\ (len1\ e)\ X)]) = len\ ([head1\ (e ++ e0)\ X])$.

Axiom $head1\_cons1\_neq$:
$\forall$ $e$ $X$ $e0$ $T$,
$((X == len1\ e) \rightarrow false)$->
$len\ ([head1\ (e ++ T :; e0)\ X]) = len\ ([head1\ (e ++ e0)\ (S\_Pred\ X\ (len1\ e))])$.

Axiom $head1\_cons2\_le$: $\forall$ $e$ $e0$ $T$ $X$,
$len1\ e \leq X \rightarrow len\ ([head1\ (e ++ T ;: e0)\ X]) = S\ (len\ ([head1\ (e ++ e0)\ X]))$.

Axiom $head1\_cons2\_gt$: $\forall$ $e$ $e0$ $T$ $X$,
$(S\ X \leq len1\ e) \rightarrow$
$len\ ([head1\ (e ++ T ;: e0)\ X]) = len\ ([head1\ (e ++ e0)\ X])$.

Axiom $head1\_cons2$: $\forall$ $e$ $e0$ $M$ $N$ $X$,
$len$ ([$head1$ ($e$ ++ $M$ ;: $e0$) $X$]) = $len$ ([$head1$ ($e$ ++ $N$ ;: $e0$) $X$]).

Applying functions to lists

Section $Maps2$.
Variable $f$:$nat \rightarrow A \rightarrow A$.

Fixpoint $list2\_map$ ($e$:$list2$) {$struct$ $e$}:$list2$:=
$match$ $e$ $with$
| $nil2$ $\Rightarrow$ $nil2$
| $T'$ :; $e'$ $\Rightarrow$ $f$ ($len$ ([$e'$])) $T'$ :; $list2\_map$ $e'$
| $T'$ ;: $e'$ $\Rightarrow$ $f$ ($len$ ([$e'$])) $T'$ ;: $list2\_map$ $e'$
$end$.

Axiom $list2\_apps$: $\forall$ $e$ $e0$,
[list2_map e ++ e0] = $list\_map$ $f$ ([$e$]) ++ [e0].

Axiom $head1\_map$: $\forall$ $e$ $e0$ $X$,
$len$ ([$head1$ ($list2\_map$ $e$ ++ $e0$) $X$]) = $len$ ([$head1$ ($e$ ++ $e0$) $X$]).

Axiom $ine\_app\_le$: $\forall$ $e$ $e0$ $X$ $T$,
$len1$ $e \leq X \rightarrow ine$ ($e$ ++ $e0$) $X$ $T \rightarrow ine$ ($list2\_map$ $e$ ++ $e0$) $X$ $T$.

Axiom $ine\_app\_gt$: $\forall$ $e$ $e0$ $X$ $T$,
($S$ $X \leq len1$ $e$)-> $ine$ ($e$ ++ $e0$) $X$ $T \rightarrow$
$ine$ ($list2\_map$ $e$ ++ $e0$) $X$ ($f$ ($len$ ([$e$]) - $len$ ([$head1$ ($e$ ++ $e0$) $X$])) $T$).

End $Maps2$.

End $Lists$.

Hint $Resolve$ $lel\_refl$.

# 4 Types

Inductive $type$:$Set$:=
$\quad$| $top$ :$type$
$\quad$| $ref$ :$nat$ →$type$
$\quad$| $arrow$:$type$ → $type$ → $type$
$\quad$| $all$ :$type$ → $type$ → $type$.

Infix "−>" := $arrow$.
Infix "." := $all$.

By using coercion $ref$:$nat$ >-> $type$ we identify $X$ with $ref\ X$.

*Coercion ref:nat >-> type.*

Fixpoint $tshift$ ($X$:$nat$) ($T$:$type$) {$struct\ T$}:$type$:=
$\quad$*match T with*
$\quad$| $top \Rightarrow top$
$\quad$| $ref\ Y \Rightarrow S\_Next\ X\ Y$
$\quad$| $T1$ −> $T2 \Rightarrow$ ($tshift\ X\ T1$) −> ($tshift\ X\ T2$)
$\quad$| $T1$ . $T2 \Rightarrow$ ($tshift\ X\ T1$) . ($tshift$ ($S\ X$) $T2$)
$\quad$*end.*

Notation " + " := ($tshift$ 0).

Axiom $tshift\_tshift\_prop$: $\forall\ T\ n$,
$\quad$$tshift\ (S\ n)\ (+\ T) = +\ (tshift\ n\ T)$.

Operator $lift\ X$ is operator 'tshift 0', applied X times.

Fixpoint $lift$ ($X$:$nat$)($T$:$type$){$struct\ X$}:$type$:=
$\quad$*match X with*
$\quad$| $0 \Rightarrow T$
$\quad$| $S\ X' \Rightarrow +\ (lift\ X'\ T)$
$\quad$*end.*

Axiom $lift\_tshift$:$\forall\ X\ T$,
$\quad$$lift\ (S\ X)\ T = lift\ X\ (+\ T)$.

Axiom $lift\_tshift\_0$:$\forall\ X\ T$,
$\quad$$lift\ X\ (+\ T) = +\ (lift\ X\ T)$.

Axiom $tshift\_lift\_le$:$\forall\ Y\ X\ T$,
$\quad$$Y \le X \rightarrow tshift\ Y\ (lift\ X\ T) = lift\ (S\ X)\ T$.

Axiom $tshift\_lift$:$\forall\ Y\ X\ T$,
$\quad$$X \le Y \rightarrow tshift\ Y\ (lift\ X\ T) = lift\ X\ (tshift\ (Y - X)\ T)$.

Fixpoint $tsubst\ (T'{:}type)(X{:}nat)(T{:}type)\{struct\ T\}{:}type{:=}$

$match\ T\ with$

$|\ top \Rightarrow top$

$|\ ref\ Y \Rightarrow if\ (Y == X)\ then\ T'\ else\ (S\_Pred\ Y\ X)$

$|\ T1\ {-}{>}\ T2 \Rightarrow (tsubst\ T'\ X\ T1)\ {-}{>}\ (tsubst\ T'\ X\ T2)$

$|\ T1\ .\ T2 \Rightarrow (tsubst\ T'\ X\ T1)\ .\ (tsubst\ (+\ T')\ (S\ X)\ T2)$

$end.$

Definition $f\_tsubst\ (T0{:}type)(X\ {:}nat)(T{:}type){:=}tsubst\ (lift\ X\ T0)\ X\ T.$

Notation " $X\ [{=}{>}]\ T0$" $:= (f\_tsubst\ T0\ X).$

Axiom $tshift\_tsubst\_prop\_1{:}\forall\ X\ T\ T0,$

$(S\ X\ [{=}{>}]\ T0)\ (+\ T) = +\ ((X\ [{=}{>}]\ T0)\ T).$

Axiom $tsubst\_lift\_le{:}\ \forall\ Y\ X\ T\ N,$

$Y \leq X \rightarrow (Y\ [{=}{>}]N)\ (lift\ (S\ X)\ T) = lift\ X\ T.$

Axiom $tsubst\_lift\_gt{:}\ \forall\ X\ Y\ T\ N,$

$X \leq Y \rightarrow lift\ X\ ((Y\ \text{-}\ X\ [{=}{>}]\ N)\ T) = (Y\ [{=}{>}]\ N)\ (lift\ X\ T).$

Axiom $tshift\_tsubst\_2{:}\ \forall\ T\ T'\ n\ n',$

$n \leq n' \rightarrow$

$tshift\ n'\ (tsubst\ T'\ n\ T) = tsubst\ (tshift\ n'\ T')\ n\ (tshift\ (S\ n')\ T).$

Axiom $tshift\_tsubst\_prop\_2{:}\ \forall\ X\ T\ T0,$

$tshift\ X\ ((0\ [{=}{>}]\ T0)\ T) = (0\ [{=}{>}]\ tshift\ X\ T0)\ (tshift\ (S\ X)\ T).$

Axiom $tsubst\_tsubst\_prop{:}\forall\ X\ T\ U\ V,$

$(X\ [{=}{>}]\ V)((0\ [{=}{>}]\ U)\ T) = (0\ [{=}{>}]\ (X\ [{=}{>}]\ V)\ U)\ ((S\ X\ [{=}{>}]\ V)\ T).$

# 5 Type Environments for Part1a.

Operations with lists are defined in the library file.

```
Definition type_env:=list type.
Definition Nil :=nil (A:=type).
```

Well-formed types.

```
Fixpoint wft (e:type_env) (T:type) {struct T}:bool:=
    match T with
    | top ⇒ true
    | ref X ⇒ wfi e X
    | T1 -> T2 ⇒ wft e T1 && wft e T2
    | T1 . T2 ⇒ wft e T1 && wft (T1 :: e) T2
    end.
```

Relation *lel e1 e2* is the same as (*length e1*) ≤ (*length e2*).

```
Lemma lel_wft: ∀ T e e', lel e e' → wft e T → wft e' T.
Proof.
induction T; simpl;intros;b_auto.
apply lel_wfi with e;auto.
apply IHT2 with (T1 :: e);auto.
Qed.
```

Well-formed lists of types.

```
Fixpoint wfl (e:type_env){struct e}:bool:=
    match e with
    | nil ⇒ true
    | T :: e ⇒ wft e T && wfl e
    end.
```

```
Lemma wfl_app: ∀ e e0, wfl (e ++ e0) → wfl e0.
Proof.
induction e;simpl;intros;b_auto.
Qed.
```

```
Lemma list_app_wfl: ∀ e e0 T1 T2,
    wft e0 T2 → wfl (e ++ T1 :: e0) → wfl (e ++ T2 :: e0).
Proof.
induction e;simpl;intros;b_auto.
apply lel_wft with (e ++ T1 :: e0);simpl;auto.
apply app_lel;simpl;auto.
Qed.
```

# 6 Subtyping Relation

For environments of the form *e ++ U :: e0* it is ensured that *len e* is well-formed index, positioning *U* at this environment. Only such indices are allowed in rules *SA_Refl_TVar* and *SA_Trans_TVar*.

Inductive *sub*:*type_env* → *type* → *type* → *Prop*:=
| *SA_Top*:
    ∀ *e T*, *wfl e* → *wft e T* → *e* ⊢ *T* <: *top*

| *SA_Refl_TVar*:
    ∀ *e e0 U*, *wfl* (*e* ++ *U* :: *e0*) → (*e* ++ *U* :: *e0*) ⊢ (*len e*) <: (*len e*)

| *SA_Trans_TVar*:
    ∀ *e e0 T U*, (*e* ++ *U* :: *e0*) ⊢ (*lift* (*S* (*len e*)) *U*) <: *T* → (*e* ++ *U* :: *e0*) ⊢ (*len e*) <: *T*

| *SA_Arrow*:
    ∀ *e T1 T2 S1 S2*, *e* ⊢ *T1* <: *S1* → *e* ⊢ *S2* <: *T2* → *e* ⊢ (*S1* –> *S2*) <: (*T1* –> *T2*)

| *SA_All*:
    ∀ *e T1 T2 S1 S2*, *e* ⊢ *T1* <: *S1* → (*T1* :: *e*) ⊢ *S2* <: *T2* → *e* ⊢ (*S1* . *S2*) <: (*T1* . *T2*).

Notation "*e* ⊢ *x* <: *y*" := (*sub e x y*).

Subtyping implies well-formedness.

Lemma *sub_wf0*: ∀ *T U e*, *e* ⊢ *T* <: *U* → *wfl e*.
Proof.
*induction* 1;*simpl*;*intros*;*auto*.
Qed.

Lemma *sub_wf12*: ∀ *T U e*, *e* ⊢ *T* <: *U* → *wft e T* && *wft e U*.
Proof.
*induction* 1;*simpl*;*intros*;*b_auto*.
*apply wfi_app_cons*.
*apply wfi_app_cons*.
*apply wfi_app_cons*.
*apply lel_wft with* (*T1* :: *e*);*simpl*;*auto*.
Qed.

Lemma *sub_wf1*: ∀ *T U e*, *e* ⊢ *T* <: *U* → *wft e T*.
Proof.
*intros T U e H*;*cut* (*TRUE* (*wft e T* && *wft e U*)).
*intros*;*b_auto*.
*apply sub_wf12* ;*auto*.
Qed.

Lemma *sub_wf2*: $\forall$ *T U e*, *e* $\vdash$ *T* <: *U* $\rightarrow$ *wft e U*.
Proof.
*intros T U e H*;*cut* (*TRUE* (*wft e T* && *wft e U*)).
*intros*;*b_auto*.
*apply sub_wf12*;*auto*.
Qed.

Hint *Resolve sub_wf0 sub_wf1 sub_wf2*.

Another form of the rule *SA_Refl_TVar*.

Lemma *SA_Refl_TVar'*:
$\forall$ *e X*, *wfl e* $\rightarrow$ *wfi e X* $\rightarrow$ *sub e X X*.
Proof.
*intros e X H1 H2*.
*elim* (*inl_ex H2*);*intros V HV*.
*replace X with* (*len* (*head e X*)).
*rewrite decomp_list with type e X V*;*auto* .
*replace* (*len* (*head* (*head e X* ++ *V* :: *tail e X*) *X*)) *with* (*len* (*head e X*)).
*apply SA_Refl_TVar*;*auto*.
*rewrite* $\leftarrow$*decomp_list with type e X V*;*auto* .
*rewrite* (*head_len* (*inl_wf HV*));*auto*.
*rewrite* $\leftarrow$*decomp_list with type e X V*;*auto* .
*rewrite* (*head_len* (*inl_wf HV*));*auto*.
*rewrite* (*head_len* (*inl_wf HV*));*auto*.
Qed.

Another form of the rule *SA_Trans_TVar*.

Lemma *SA_Trans_TVar'*:
$\forall$ *e T X U*, *inl e X U* $\rightarrow$ *e* $\vdash$ (*lift* (*S X*) *U*) <: *T* $\rightarrow$ *e* $\vdash$ *X* <: *T*.
Proof.
*intros e T X U H*.
*rewrite decomp_list with type e X U*;*auto* .
*replace* (*S X*) *with* (*S* (*len* (*head e X*))).
*replace* (*ref X*) *with* (*ref* (*len* (*head e X*))).
*intros*;*apply SA_Trans_TVar*;*auto*.
*rewrite* (*head_len* (*inl_wf H*));*auto*.
*rewrite* (*head_len* (*inl_wf H*));*auto*.
Qed.

To enable induction on environments we are to introduce shifting of environments. See library for definition of mapping for lists.

Definition *list_tshift*:=*list_map tshift*.

Lemma *app_tshift_wft*: $\forall$ *T T0 e e0*,
    *wft (e ++ e0) T* $\rightarrow$ *wft (list_tshift e ++ T0 :: e0) (tshift (len e) T)*.
Proof.
*induction T;simpl;intros;b_auto.*
*apply lel_wfi with (e ++ T0 :: e0);auto.*
*apply app_lel_2;auto.*
*apply lel_list_map with (A:=type)(f:=tshift).*
*apply wfi_app;auto.*
*apply IHT2 with (e:=T1 :: e)(e0:=e0)(T0:=T0);auto.*
Qed.

Lemma *wft_cons_tshift*: $\forall$ *e T T0, wft e T* $\rightarrow$ *wft (T0 :: e) (+ T)*.
Proof.
*intros e T T0;intros.*
*apply app_tshift_wft with (e0:=e)(e:=Nil)(T0:=T0);auto.*
Qed.

Lemma *app_wft*:
    $\forall$ *e e0 N, wft e0 N* $\rightarrow$ *wfl (e ++ e0)* $\rightarrow$ *wft (e ++ e0) (lift (len e) N)*.
Proof.
*induction e;simpl;intros;b_auto.*
*apply wft_cons_tshift;b_auto.*
Qed.

Lemma *wfl_app_list_shift*: $\forall$ *e T e0, wft e0 T* $\rightarrow$ *wfl (e ++ e0)->*
    *wfl (list_tshift e ++ T :: e0)*.
Proof.
*induction e; simpl;intros;b_auto.*
*apply app_tshift_wft;auto.*
Qed.

Lemma A.1.

Lemma *sub_reflexivity*: $\forall$ *T e, wfl e* $\rightarrow$ *wft e T* $\rightarrow$ $e \vdash T <: T$.
Proof.
*induction T;simpl;intros.*
*apply SA_Top; auto.*
*apply SA_Refl_TVar';auto.*
*apply SA_Arrow;b_auto.*
*apply SA_All;b_auto.*
*apply IHT2;simpl;b_auto.*
Qed.

Some kind of replacement for Lemma A.2. Permutations are difficult to introduce within the nameless framework.

Lemma *sub_weakening_ind*:

∀ *e U V*, *e* ⊢ *U* <: *V* →
∀ *e0 e1 T0* , *wft e0 T0* → *e* = *e1* ++ *e0* →
(*list_tshift e1* ++ *T0* :: *e0*) ⊢ (*tshift* (*len e1*) *U*) <: (*tshift* (*len e1*) *V*).

Proof.

Induction on derivation of e |- U <: V.
*induction* 1; *intros*;*auto.*
SA_Top case
*apply SA_Top.*
*apply wfl_app_list_shift*;*auto*;*rewrite* ← *H2*;*auto.*
*apply app_tshift_wft*;*rewrite* ← *H2*;*auto.*
SA_Refl_TVar case
*simpl*;*apply SA_Refl_TVar'.*
*apply wfl_app_list_shift*;*auto*;*rewrite* ← *H1*;*auto.*
*apply lel_wfi with* (*e2* ++ *T0* :: *e1*);*auto.*
*apply app_lel_2*;*auto.*
*apply lel_list_map with* (*A*:=*type*)(*f*:=*tshift*).
*apply wfi_app*;*auto*;*rewrite* ← *H1*;*auto.*
*apply wfi_app_cons.*
SA_Trans_TVar case
*cut* (*TRUE* (*wfi* (*list_tshift e2* ++ *T0* :: *e1*) (*S_Next* (*len e2*) (*len e*)))).
*intro HX*;*elim* (*inl_ex HX*);*intros V HV.*
*simpl*;*apply SA_Trans_TVar' with V*;*auto.*
*rewrite lift_tshift.*
*apply N_Decide with* (*len e*) (*len e2*);*intro.*
case *S* (*len e2*) ≤ (*len e1*)
*rewrite S_Next_gt*;*auto.*
*rewrite inl_map_cons_gt with* (*A*:=*type*)(*f*:=*tshift*) (*e2*:=*e2*)(*e1*:= *e1*) (*X*:= *len e*) (*T1*:= *U*)(*T2*:= *V*) (*T0*:=*T0*);*auto.*
*rewrite* ← *lift_tshift.*
*rewrite* ←*tshift_lift*;*auto.*
*rewrite* ←*H1*;*auto.*
*apply inl_app_cons.*
*rewrite S_Next_gt in HV*;*auto.*
case *len e2* ≤ *len e*
*rewrite S_Next_le*;*auto.*
*rewrite* ←*inl_map_cons_le with* (*A*:=*type*)(*f*:=*tshift*)(*e1*:=*e1*)(*e2*:=*e2*)(*T1*:=*U*)(*T2*:=*V*)(*T0*:=*T0*)(
*e*) ;*auto.*
*rewrite* ←*tshift_lift_le with* (*len e2*) (*len e*) ( + *U*);*auto.*
*rewrite* ←*lift_tshift*;*auto.*
*rewrite* ←*H1*;*eauto.*
*apply inl_app_cons.*

$rewrite\ S\_Next\_le\ in\ HV;auto.$
$apply\ lel\_wfi\ with\ (e2\ {+}{+}\ T0\ ::\ e1).$
$apply\ app\_lel\_2.$
$apply\ lel\_list\_map\ with\ (A{:=}type)(f{:=}tshift).$
$apply\ wfi\_app;auto.$
$rewrite\ \leftarrow H1;apply\ wfi\_app\_cons;auto.$
SA_Arrow case
$simpl;apply\ SA\_Arrow;\ auto.$
SA_All case
$simpl;apply\ SA\_All;\ auto.$
$apply\ IHsub2\ with\ (e1{:=}T1\ ::\ e1)(e0{:=}e0)(T0{:=}T0);auto;rewrite\ H2;auto.$
Qed.

Lemma $sub\_weakening$: $\forall\ e\ V\ T\ U,$
$e \vdash T <: U \rightarrow wft\ e\ V \rightarrow (V\ ::\ e) \vdash (+\ T) <: (+\ U).$
Proof.
$intros\ e\ V;intros.$
$apply\ sub\_weakening\_ind\ with\ (e{:=}e)(e0{:=}e)(e1{:=}Nil)(T0{:=}V);auto.$
Qed.

Iterative application of previous lemma, provides weakening for extra concatenation.

Lemma $sub\_weak\_app\_cons$: $\forall\ e\ e0\ U\ T\ V,$
$e0 \vdash T <: U \rightarrow wfl\ (e\ {+}{+}\ V\ ::\ e0) \rightarrow$
$(e\ {+}{+}\ V\ ::\ e0) \vdash (lift\ (S\ (len\ e))\ T) <: (lift\ (S\ (len\ e))\ U).$
Proof.
$induction\ e;simpl;intros;auto.$
$apply\ sub\_weakening;b\_auto.$
$simpl\ in\ \times;simpl;apply\ sub\_weakening;b\_auto.$
Qed.

# 7 Transitivity and Narrowing for Subtyping Relation

We shall use induction on structural depth of a type.

```
Fixpoint max (n m:nat) {struct n}:nat:=
    match n, m with
    | 0 , _ ⇒ m
    | S n', 0 ⇒ n
    | S n', S m' ⇒ S (max n' m')
    end.
```

```
Lemma le_max_l: ∀ n m :nat, le n (max n m).
Proof.
induction n; auto with arith.
simpl;induction m; auto with arith.
Qed.
```

```
Lemma le_max_r: ∀ n m, le m (max n m).
Proof.
induction n; auto with arith.
simpl;induction m; auto with arith.
Qed.
```

```
Fixpoint depth (T:type):nat:=
    match T with
    | ref X ⇒ 0
    | top ⇒ 0
    | T1 -> T2 ⇒ S (max (depth T1) (depth T2))
    | T1 . T2 ⇒ S (max (depth T1) (depth T2))
    end.
```

```
Lemma tshift_depth: ∀ T e, depth (tshift e T) = depth T.
Proof.
induction T;auto; simpl;intros n; rewrite IHT1; rewrite IHT2; auto.
Qed.
```

```
Lemma lift_depth: ∀ (e:type_env) T, depth (lift (len e) T) = depth T.
Proof.
induction e;simpl;intros;auto;rewrite tshift_depth;auto.
Qed.
```

Subtyping narrowing under assumption of transitivity.

Lemma $sub\_narrowing0$: $\forall$ $T1$,
$(\forall\ Q, depth\ Q = depth\ T1 \rightarrow$
$\forall\ e\ X\ T,$
$e \vdash X <: Q \rightarrow e \vdash Q <: T \rightarrow e \vdash X <: T) \rightarrow$
$\forall\ e\ M\ N,\ e \vdash M <: N \rightarrow$
$\forall\ e1\ e0\ T2,$
$e0 \vdash T2 <: T1 \rightarrow e = e1 ++ T1 :: e0 \rightarrow (e1 ++ T2 :: e0) \vdash M <: N.$
Proof.
Induction on derivation of e |- M <: N.
*induction* 2;*intros*;*auto.*
SA_Top case
*apply SA_Top*;*auto.*
*apply list_app_wfl with T1*;*eauto*;*rewrite* ←*H3*;*eauto.*
*apply lel_wft with e*;*auto*;*rewrite H3*;*auto.*
*apply app_lel*;*simpl*;*auto.*
SA_Refl_TVar case
*apply SA_Refl_TVar'*;*simpl*;*auto.*
*apply list_app_wfl with T1*;*eauto*;*rewrite* ←*H2*;*eauto.*
*apply lel_wfi with* (*e1* ++ *T1*::*e2*);*auto.*
*apply app_lel*;*simpl*;*auto.*
*rewrite* ←*H2*;*auto.*
*apply wfi_app_cons.*
SA_Trans_TVar case
*apply Decide with* (*len e* == *len e1*);*intro HA.*
Case (len e == (len e1).
*simpl*;*apply SA_Trans_TVar' with T2*;*auto.*
*rewrite* (*n_eq_1 HA*).
*apply inl_app_cons.*
*apply H with* (*lift* (*S* (*len e1*)) *T1*).
*simpl*;*rewrite tshift_depth*;*apply lift_depth.*
*rewrite* (*n_eq_1 HA*).
*apply sub_weak_app_cons*;*auto.*
*apply list_app_wfl with T1*;*eauto*;*rewrite* ←*H2*;*eauto.*
*rewrite inl_inv with type* (*e1* ++ *T1* :: *e2*) (*len e1*) *T1 U*;*auto.*
*rewrite* (*n_eq_2 HA*);*auto.*
*apply inl_app_cons.*
*rewrite* ←*H2*;*rewrite* (*n_eq_2 HA*);*apply inl_app_cons.*
Case ˜ (len e == len e1.
*simpl*;*apply SA_Trans_TVar' with U*;*auto.*
*apply inl_app_neq with T1*;*auto.*
*rewrite* ←*H2*;*auto.*
*apply inl_app_cons.*

SA_Arrow case
*apply SA_Arrow; auto.*
SA_All case
*apply SA_All; auto.*
*apply IHsub2 with (e1:=T0 ::e1)(e0:=e0)(T3:=T3);simpl;b_auto;rewrite H1;auto.*
Qed.

Special case of subtyping narrowing required for the proof of transitivity.

Lemma *sub_narrowing1*: $\forall$ *e T1 T2*,
$e \vdash T2 <: T1 \rightarrow$
$(\forall\ Q,\ depth\ Q = depth\ T1 \rightarrow$
$\forall\ e\ X\ T,\ e \vdash X <: Q \rightarrow e \vdash Q <: T \rightarrow e \vdash X <: T) \rightarrow$
$\forall\ M\ N,\ (T1 :: e) \vdash M <: N \rightarrow (T2 :: e) \vdash M <: N.$
Proof.
*intros e T1 T2;intros.*
*apply sub_narrowing0 with (T1:=T1)(T2:=T2)(e:= T1 :: e)(e0:=e)(e1:=Nil);simpl;b_auto*
.
Qed.

Theorem *sub_transitivity_step*:
$\forall\ n,$
$(\forall\ Q,\ depth\ Q < n \rightarrow \forall\ e\ U\ T,$
$e \vdash U <: Q \rightarrow e \vdash Q <: T \rightarrow e \vdash U <: T) \rightarrow$
$\forall\ Q,\ depth\ Q < S\ n \rightarrow \forall\ e\ U\ T,$
$e \vdash U <: Q \rightarrow e \vdash Q <: T \rightarrow e \vdash U <: T.$
Proof.
*intros n H0 Q H e U T H1 H2.*
Induction on derivation of $e \vdash U <: Q$ with case analysis of $e \vdash Q <: T$.
*induction H1;simpl;intros;auto.*
Case 1
*inversion_clear H2;auto.*
*apply SA_Top; auto.*
Case 2
*simpl;apply SA_Trans_TVar;auto.*
Case 3
*inversion_clear H2;simpl;intros;auto.*
*apply SA_Top;simpl;b_auto.*
*apply lel_wft with (T1 :: e);simpl;eauto.*
*apply SA_Arrow.*
*apply H0 with T1;auto;apply le_lt_trans with (max (depth T1) (depth T2));auto with arith.*
*apply le_max_l.*
*apply H0 with T2;auto;apply le_lt_trans with (max (depth T1) (depth T2));auto with arith.*
*apply le_max_r.*
Case 4

*inversion_clear H2*;*intros*;*auto*.
*apply SA_Top*;*simpl*;*b_auto*.
*apply SA_All*.
*apply H0 with T1*;*auto*;*apply le_lt_trans with* (*max* (*depth T1*) (*depth T2*));*auto with arith*.
*apply le_max_l*.
*apply H0 with T2*;*auto*.
*apply le_lt_trans with* (*max* (*depth T1*) (*depth T2*));*auto with arith*.
*apply le_max_r*.
*apply sub_narrowing1 with T1*;*auto*.
*intros*;*apply H0 with Q* ;*auto*;*rewrite H2*.
*apply le_lt_trans with* (*max* (*depth T1*) (*depth T2*));*auto with arith*.
*apply le_max_l*.
Qed.

Main result of the part 1.(Lemma 3.1)

Theorem *sub_transitivity*: $\forall$ *e U Q T*,
$e \vdash U <: Q \rightarrow e \vdash Q <: T \rightarrow e \vdash U <: T$ .
Proof.
*cut* ($\forall$ *n Q*, *lt* (*depth Q*) *n* $\rightarrow$
$\forall$ *e U T*, $e \vdash U <: Q \rightarrow e \vdash Q <: T \rightarrow e \vdash U <: T$).
*intros H e U Q T* ; *apply* (*H* (*S* (*depth Q*)));*auto with arith*.
*induction n*.
*intros*; *absurd* (*depth Q* $<$ 0);*auto with arith*.
*apply sub_transitivity_step*;*auto*.
Qed.

Narrowing lemma (Lemma 3.2)

Lemma *sub_narrowing*: $\forall$ *e1 e P Q M N*,
*wfl* (*e1* ++ *P* :: *e*) $\rightarrow e \vdash P <: Q \rightarrow$
(*e1* ++ *Q* :: *e*) $\vdash M <: N \rightarrow$ (*e1* ++ *P* :: *e*) $\vdash M <: N$.
Proof.
*intros e1 e P Q*;*intros*.
*apply sub_narrowing0 with Q* (*e1* ++ (*Q* :: *e*));*auto*.
*intros Q0 H2 e0 U T*;*apply sub_transitivity*;*auto*.
Qed.

# 8 Type substitution preserves subtyping (Lemma A.10)

To make preparation for results on typing being stable under type substitutions (*lemma subst_type_preserves_typ*), we are to consider type substitutions for environments to formulate and prove *tsubst_preserves_subtyp*.

Note that the definition of *tsubst* perfectly matches wfi_app_2 in the following lemma.

Definition *list_tsubst* (*T0*:*type*):=*list_map* (*f_tsubst T0*).

Lemma *app_tsubst_wft_0*:
    ∀ *U e*, *wft e U* →
    ∀ *e1 e0 M*, *e* = *e1* ++ *M* :: *e0* → *wft e0 M* →
    ∀ *N*, *wft e0 N* → *wfl* (*list_tsubst N e1* ++ *e0*) →
    *wft* (*list_tsubst N e1* ++ *e0*) ((*len e1* [=>] *N*) *U*).
Proof.
*unfold f_tsubst*;*induction U*;*simpl*;*intros*;*b_auto*.
Case for the first line of the definition of tsubst
*apply Decide with* (*n* == (*len e1*)).
*intros*;*b_rewrite*.
Case (n == (len e1).
*intros*;*rewrite* ←*len_list_map with* (*A*:=*type*)(*f*:=*f_tsubst N*).
*apply app_wft*;*auto*.
Case ˜(n == (len e1).
*rewrite* ←*len_list_map with* (*A*:=*type*)(*f*:=*f_tsubst N*).
*intros*;*b_rewrite*;*simpl*;*apply wfi_app_2 with M*;*auto*.
*apply lel_wfi with* (*e*);*auto*;*rewrite H0*;*auto*;*apply app_lel_2*.
*apply lel_list_map with* (*A*:=*type*)(*f*:=*f_tsubst N*).
Induction step
*apply IHU2 with* (*e*:=*U1* :: *e*)
(*e1*:=*U1* :: *e1*)(*e0*:=*e0*)(*M*:=*M*)(*N*:=*N*); *simpl*;*unfold f_tsubst*;*b_auto*.
*congruence*.
Qed.

Lemma *app_tsubst_wfl*:
    ∀ *e e0 M*, *wfl* (*e* ++ *M* ::*e0*) →
    ∀ *N*, *wft e0 N* → *wfl* (*list_tsubst N e* ++ *e0*).
Proof.
*induction e*;*simpl*;*intros*;*b_auto*.
*apply app_tsubst_wft_0 with* (*e* ++ *M* :: *e0*) *M*;*b_auto*.
*cut* (*TRUE* (*wfl* (*M* :: *e0*))).
*simpl*;*intro*;*b_auto*.
*apply wfl_app with e*;*auto*.
Qed.

Lemma *app_tsubst_wft*:
$\forall$ *e e0 M T1*, *wfl* (*e* ++ *M* ::*e0*) $\rightarrow$ *wft* (*e* ++ *M* :: *e0*) *T1* $\rightarrow$
$\forall$ *T*, *wft e0 T* $\rightarrow$ *wft* (*list_tsubst T e* ++ *e0*)((*len e* [=>] *T*) *T1*).
Proof.
*intros e0 e M T T1*;*intros*;*simpl.*
*apply app_tsubst_wft_0 with* (*e0* ++ *M* :: *e*) *M*;*auto.*
*cut* (*TRUE* (*wfl* (*M* :: *e*))).
*simpl*;*intro*;*b_auto.*
*apply wfl_app with e0*;*auto.*
*apply app_tsubst_wfl with M*;*auto.*
Qed.

Some kind of subtyping weakening for substitution of environments.

Lemma *app_tsubst_sub*:
$\forall$ *e e0 M N*, *wfl* (*e* ++ *M* :: *e0*) $\rightarrow$ *e0* $\vdash$ *N* <: *M* $\rightarrow$
(*list_tsubst N e* ++ *e0*) $\vdash$ (*lift* (*len e*) *N*) <: (*lift* (*len e*) *M*).
Proof.
*induction e*;*simpl*;*intros*;*auto.*
*apply sub_weakening*;*b_auto.*
*apply app_tsubst_wft with M*;*b_auto.*
Qed.

Equivalent for Lemma A.10.

Lemma *tsubst_preserves_subtyp*:
$\forall$ *e U V*, *e* $\vdash$ *U* <: *V* $\rightarrow$
$\forall$ *e1 e0 M N*, *e* = *e1* ++ *M* :: *e0* $\rightarrow$ *e0* $\vdash$ *N* <: *M* $\rightarrow$
(*list_tsubst N e1* ++ *e0*) $\vdash$ ((*len e1* [=>] *N*) *U*) <: ((*len e1* [=>] *N*) *V*).
Proof.
Induction on derivation of e |- U <: V.
*induction* 1;*intros*;*auto.*
SA_Top case
*apply SA_Top.*
*apply app_tsubst_wfl with M*;*eauto*;*rewrite* $\leftarrow$*H1*;*eauto.*
*apply app_tsubst_wft with M*;*eauto*;*rewrite* $\leftarrow$*H1*;*eauto.*
SA_Refl_TVar case
*apply sub_reflexivity*;*auto.*
*apply app_tsubst_wfl with M*;*eauto*;*rewrite* $\leftarrow$*H0*;*eauto.*
*apply app_tsubst_wft with M*;*eauto*;*rewrite* $\leftarrow$*H0*;*auto.*
*simpl*;*apply wfi_app_cons.*
SA_Trans_TVar case
*unfold f_tsubst*;*intros*;*simpl.*
*apply Decide with* (*len e* == *len e1*);*intros*;*b_rewrite.*
Case *len e* == *len e1*.

```
apply sub_transitivity with (lift (len e1) M);eauto.
apply app_tsubst_sub;auto.
rewrite ←H0;eauto.
rewrite ←tsubst_lift_le with (len e1) (len e1) M N;auto.
replace (S (len e1)) with (S (len e));auto.
unfold f_tsubst in ×;eauto.
rewrite inl_inv with type (e1 ++ M :: e2) (len e1) M U;eauto.
apply inl_app_cons.
rewrite ←H0;rewrite (n_eq_2 H2).
apply inl_app_cons.
rewrite (n_eq_2 H2);auto.
Case ˜ (len e == len e1.
apply N_Decide with (len e) (len e1);auto;intro.
case S (len e) ≤ len e1
rewrite S_Pred_gt;auto.
cut (TRUE (wfi (list_tsubst N e1 ++ M :: e2) (S_Next (len (list_tsubst N e1)) (len
e)))).
intro HX;elim (inl_ex HX);intros V HV.
simpl;apply SA_Trans_TVar' with V;auto.
apply inl_app_1 with M;auto.
replace (tsubst (lift (len e1) N) (len e1) T) with (f_tsubst N (len e1) T);auto.
rewrite inl_map_cons_gt with (A:=type)(f:=f_tsubst N) (e2:=e1)(e1:= e2) (X:= len e)
(T1:= U)(T2:= V) (T0:=M);auto.
rewrite tsubst_lift_gt;eauto.
apply inl_app_1 with M;auto.
rewrite ←H0;eauto.
rewrite S_Next_gt;auto.
apply inl_app_cons.
rewrite S_Next_gt in HV;auto.
unfold list_tsubst;rewrite len_list_map with (A:=type)(f:=f_tsubst N)(e:=e1);auto.
rewrite S_Next_gt;auto.
apply lel_wfi with (e1 ++ M::e2).
apply app_lel_2;apply lel_list_map with (A:=type)(f:=f_tsubst N)(e:=e1).
rewrite ← H0;auto;apply wfi_app_cons.
unfold list_tsubst;rewrite len_list_map with (A:=type)(f:=f_tsubst N)(e:=e1);auto.
case len e1 ≤ len e
simpl;apply SA_Trans_TVar' with U;eauto.
replace (len e1) with (len (list_map (f_tsubst N) e1)).
apply inl_app_2 with M;auto.
rewrite len_list_map with (A:=type)(f:=f_tsubst N);auto.
apply inl_map_app_le;auto.
rewrite ← H0;auto.
```

*apply inl_app_cons*;*auto*.
*rewrite len_list_map with* $(A{:=}type)(f{:=}f\_tsubst\ N)$;*auto*.
*rewrite S_Pred_le*;*auto*.
*rewrite* ←*tsubst_lift_le with* $(len\ e1)\ (len\ e)\ U\ N$;*auto*.
*unfold f_tsubst in* ×;*apply IHsub with* $M$;*auto*.
SA_Arrow case
*unfold f_tsubst in* ×;*simpl*;*apply SA_Arrow*; *eauto*.
SA_All case
*unfold f_tsubst in* ×;*simpl*;*apply SA_All*; *eauto*.
*apply IHsub2 with* $(e1{:=}T1 :: e1)(e0{:=}e0)(M{:=}M)(N{:=}N)$;*simpl*;*b_auto*.
*congruence*.
Qed.

# 9 Terms

Inductive $term$:$Set$:=
 | $var$: $nat \rightarrow term$
 | $abs$: $type \rightarrow term \rightarrow term$
 | $apl$: $term \rightarrow term \rightarrow term$
 | $tabs$:$type \rightarrow term \rightarrow term$
 | $tapl$:$term \rightarrow type \rightarrow term$.

By using coercion $var$:$nat$ >-> $term$ we identify $X$ with $var$ $X$.

*Coercion var:nat >-> term.*

Notation "$x$ $y$" := $apl$ $x$ $y$.
Infix ".$\rightarrow$" := $abs$.
Infix "$x$ [y] := $tapl$ $x$ $y$.
Infix ".$\Rightarrow$" := $tabs$.

Shifting of terms
Fixpoint $shift$ ($x$:$nat$) ($t$:$term$) {$struct$ $t$}:$term$:=
 $match$ $t$ $with$
 | $var$ $y \Rightarrow S\_Next$ $x$ $y$
 | $t1.\rightarrow$ $t2 \Rightarrow t1.\rightarrow$ $shift$ ($S$ $x$) $t2$
 | $t1$ $t2 \Rightarrow$ ($shift$ $x$ $t1$) ($shift$ $x$ $t2$)
 | $t1.\Rightarrow$ $t2 \Rightarrow t1.\Rightarrow$ ($shift$ $x$ $t2$)
 | $t1$ [t2] $\Rightarrow$ ($shift$ $x$ $t1$) [t2]
 $end$.

Notation "+" := ($shift$ 0).

Fixpoint $shift\_type$ ($X$:$nat$) ($t$:$term$) {$struct$ $t$}:$term$:=
 $match$ $t$ $with$
 | $var$ $y \Rightarrow y$
 | $t1.\rightarrow$ $t2 \Rightarrow$ ($tshift$ $X$ $t1$).$\rightarrow$ ($shift\_type$ $X$ $t2$)
 | $t1$ $t2 \Rightarrow$ ($shift\_type$ $X$ $t1$) ($shift\_type$ $X$ $t2$)
 | $t1.\Rightarrow$ $t2 \Rightarrow$ ($tshift$ $X$ $t1$) .$\Rightarrow$ ($shift\_type$ ($S$ $X$) $t2$)
 | $t1$ [t2] $\Rightarrow$ ($shift\_type$ $X$ $t1$) [tshift X t2]
 $end$.

Notation "+" := ($shift\_type$ 0).

Substitution of types in terms.

Fixpoint $subst\_type$ $(T{:}type)$ $(X{:}nat)$ $(t{:}term)$ $\{struct\ t\}{:}term{:=}$

$match\ t\ with$
$\mid var\ y \Rightarrow y$
$\mid t1.\!\rightarrow t2 \Rightarrow ((X\ [{=}{>}]\ T)\ t1)\ .\!\rightarrow (subst\_type\ T\ X\ t2)$
$\mid t1\ t2 \Rightarrow (subst\_type\ T\ X\ t1)\ (subst\_type\ T\ X\ t2)$
$\mid t1.\!\Rightarrow t2 \Rightarrow ((X\ [{=}{>}]\ T)\ t1)\ .\!\Rightarrow (subst\_type\ T\ (S\ X)\ t2)$
$\mid t1$ [t2] $\Rightarrow (subst\_type\ T\ X\ t1)\ [(X\ [{=}{>}]\ T)\ t2]$
$end.$

Notation " $x\ [{-}{>}]\ T$" := $(subst\_type\ T\ x)$.

Substitution of terms.

Fixpoint $subst$ $(s{:}term)$ $(x{:}nat)$ $(t{:}term)$ $\{struct\ t\}{:}term{:=}$

$match\ t\ with$
$\mid var\ y \Rightarrow if\ (y == x)\ then\ s\ else\ (S\_Pred\ y\ x)$
$\mid t1.\!\rightarrow t2 \Rightarrow t1\ .\!\rightarrow (subst\ (+\ s)\ (S\ x)\ t2)$
$\mid t1\ t2 \Rightarrow (subst\ s\ x\ t1)\ (subst\ s\ x\ t2)$
$\mid t1.\!\Rightarrow t2 \Rightarrow t1\ .\!\Rightarrow (subst\ (+\ s)\ x\ t2)$
$\mid t1$ [t2] $\Rightarrow (subst\ s\ x\ t1)$ [t2]
$end.$

Notation " $x$ [->] $t$" := $(subst\ t\ x)$.

# 10 Type Enviroments for Part2a.

The structure *list2* and operations with such lists are defined in the library file.

```
Definition env:=list2 type.
Definition empty:= nil2 type.
```

```
Infix ":;" := cons1.
Infix ";:" := cons2.
Notation "[ x ]" := (lst x).
```

Well-formed environments.

```
Fixpoint wfe (e:env){struct e}:bool:=
      match e with
      | nil2 ⇒ true
      | T :; e ⇒ wft ([e]) T && wfe e
      | T ;: e ⇒ wft ([e]) T && wfe e
      end.
```

```
Lemma wfe_cons1: ∀ e T, wfe (T :; e) → wft ([e]) T.
Proof.
simpl;intros;b_auto.
Qed.
```

```
Lemma wfe_cons2: ∀ e T, wfe (T ;: e) → wft ([e]) T.
Proof.
simpl;intros;b_auto.
Qed.
```

Down to part 1 environments.

```
Lemma wf_lst: ∀ e, wfe e → wfl ([e]).
Proof.
induction e;simpl;intros;b_auto.
Qed.
```

```
Infix "++" := list2_app.
```

```
Lemma env_app_cons_wf:
      ∀ e e0 T, wfe (e ++ T :; e0) → wfe (e ++ e0).
Proof.
induction e; simpl;intros;b_auto.
rewrite list2list_app.
replace ([e0]) with ([T :; e0]);auto.
rewrite ←list2list_app;auto.
rewrite list2list_app.
replace ([e0]) with ([T :; e0]);auto.
rewrite ←list2list_app;auto.
```

Qed.

Lemma *env_app_cons_wf_2*:

$\forall$ *e e0 T*, *wft* ([*e0*]) *T* $\rightarrow$ *wfe* (*e* ++ *e0*) $\rightarrow$ *wfe* (*e* ++ *T* :; *e0*).

Proof.

*induction e*; *simpl*;*intros*;*b_auto*.
*rewrite list2list_app*.
*simpl*.
*rewrite* $\leftarrow$*list2list_app*;*auto*.
*rewrite list2list_app*.
*simpl*.
*rewrite* $\leftarrow$*list2list_app*;*auto*.
Qed.

Lemma *env_app_wfe*:

$\forall$ *e e0 T1 T2*, *wft* ([*e0*]) *T2* $\rightarrow$ *wfe* (*e* ++ *T1* ;: *e0*) $\rightarrow$ *wfe* (*e* ++ *T2* ;: *e0*).

Proof.

*induction e*;*simpl*;*intros*;*b_auto*.
*rewrite list2list_app*.
*rewrite list2list_app in H1*.
*apply lel_wft with* ([*e*] ++ *T1* :: [e0]);*simpl*;*auto*.
*apply app_lel*;*simpl*;*auto*.
*rewrite list2list_app*.
*rewrite list2list_app in H1*.
*apply lel_wft with* ([*e*] ++ *T1* :: [e0]);*simpl*;*auto*.
*apply app_lel*;*simpl*;*auto*.
Qed.

# 11 Typing Relations

For environments of the form *e ++ t :; e0* it is ensured that *len1 e* is well-formed index, positioning *t* at this environment. Only such indices are allowed in rules *T_Var*.

Inductive $typ : env \to term \to type \to Prop$:=
  | $T\_Var$: $\forall$ $e$ $e0$ $t$,
    $wfe\ (e\ ++\ t\ :;\ e0) \to (e\ ++\ t\ :;\ e0) \vdash (len1\ \ e) <: (lift\ (len\ ([e]))\ t)$

  | $T\_Abs$: $\forall$ $e$ $t$ $t1$ $t2$,
    $(t1\ :;\ e) \vdash t <:\ t2 \to e \vdash (t1.\!\to t) <: (t1\ -\!\!>\ t2)$

  | $T\_App$: $\forall$ $e$ $t1$ $t2$ $t11$ $t12$,
    $e \vdash t1\ <:\ (t11\ -\!\!>\ t12) \to e \vdash t2\ <:\ t11 \to e \vdash (t1\ \ t2)\ <:\ t12$

  | $T\_Tabs$: $\forall$ $e$ $t$ $t1$ $t2$,
    $(t1\ ;:\ e) \vdash t <:\ t2 \to e \vdash (t1.\!\Rightarrow t) <: (t1\ .\ t2)$

  | $T\_Tapp$: $\forall$ $e$ $t$ $t11$ $t12$ $t0$,
    $e \vdash t <:\ (t11\ .\ t12) \to [\mathrm{e}] \vdash t0\ <:\ t11 \to e \vdash (t\ [\mathrm{t0}])\ <:\ ((0\ [=>]\ t0)\ t12)$

  | $T\_Sub$: $\forall$ $e$ $t$ $t1$ $t2$,
    $e \vdash t <:\ t1 \to [\mathrm{e}] \vdash t1\ <:\ t2 \to e \vdash t <:\ t2$.

Notation "$e \vdash x : y$" := $(typ\ e\ x\ y)$.

Another form of the rule *T_Var*. Operator *head1* introduced in the library, so that *head1* $(e1\ ++\ t\ :;\ e2)\ (len1\ \ e1) = e1$.

Lemma $T\_Var'$: $\forall$ $e$ $X$ $t$,
    $wfe\ e \to ine\ e\ X\ t \to typ\ e\ X\ (lift\ (len\ ([head1\ \ e\ X]))\ t)$.

Proof.
*intros e X t H1 H2.*
*cut (typ (head1 e X ++ (t :; tail1 e X)) (len1 (head1 e X)) (lift (len ([head1 e X])) t)).*
*replace (head1 e X ++ (t :; tail1 e X)) with e;auto.*
*replace (len1 (head1 e X)) with X;auto.*
*rewrite (head_len1 H2);auto.*
*apply (decomp_list2 H2);auto.*
*apply T_Var;auto.*
*rewrite <-(decomp_list2 H2);auto.*
Qed.

# 12 Type Substitution Preserves Typing (Lemma A.11)

Typing implies well-formedness.

Lemma $typ\_wf1$: $\forall\ e\ t\ U,\ e \vdash t\ :\ U \rightarrow wfe\ e.$
Proof.
$intros\ e\ t\ U\ H;\ induction\ H; simpl\ in\ \times; intros; b\_auto.$
Qed.

Hint $Resolve\ typ\_wf1.$

Lemma $typ\_wf2$: $\forall\ e\ t\ U,\ e \vdash t\ :\ U \rightarrow wft\ ([e])\ U.$
Proof.
$induction\ 1; simpl\ in\ \times; intros; b\_auto.$
$induction\ e; simpl\ in\ \times; intros; b\_auto.$
$apply\ wft\_cons\_tshift; auto.$
$apply\ wfe\_cons1; eauto.$
$apply\ wfe\_cons2; eauto.$
$apply\ app\_tsubst\_wft\ with\ (e0{:=}[\mathrm{e}])(M{:=}t11)(T{:=}t0)(T1{:=}t12)(e{:=}Nil); simpl; b\_auto.$
Qed.

Hint $Resolve\ typ\_wf2.$

To enable induction to prove typing stabilty under type substitutions

(lemma $subst\_type\_preserves\_typ$), we are to consider substitutions for environments.

Definition $env\_tsubst\ (T0{:}type){:=}list2\_map\ (f\_tsubst\ T0).$

Lemma $app\_tsubst\_wfe$:
$\forall\ e\ e0\ M, wfe\ (e\ {+}{+}\ M\ ::\ e0) \rightarrow$
$\forall\ N,\ wft\ ([e0])\ N \rightarrow wfe\ (env\_tsubst\ N\ e\ {+}{+}\ e0).$
Proof.
$induction\ e; simpl; intros; b\_auto.$
$rewrite\ list2\_apps\ with\ (A{:=}type)(f{:=}(f\_tsubst\ N))(e{:=}e); auto.$
$apply\ app\_tsubst\_wft\ with\ (M{:=}M)(T1{:=}a)(T{:=}N)(e{:=}[e])(e0{:=}[e0]); auto.$
$rewrite \leftarrow list2list\_cons; rewrite \leftarrow list2list\_app; apply\ wf\_lst; auto.$
$rewrite\ list2list\_app\ in\ H1; auto.$
$rewrite\ list2\_apps\ with\ (A{:=}type)(f{:=}(f\_tsubst\ N))(e{:=}e); auto.$
$apply\ app\_tsubst\_wft\ with\ (M{:=}M)(T1{:=}a)(T{:=}N)(e{:=}[e])(e0{:=}[e0]); auto.$
$rewrite \leftarrow list2list\_cons; rewrite \leftarrow list2list\_app; apply\ wf\_lst; auto.$
$rewrite\ list2list\_app\ in\ H1; auto.$
Qed.

Equivalent for Lemma A.11.

Lemma *subst_type_preserves_typ_ind*:
$\forall$ *e U V*, $e \vdash U : V \rightarrow$
$\forall$ *e1 e0 M N*, *e* = *e1* ++ *M* ;: *e0* $\rightarrow$ [e0]$\vdash N$ <: $M \rightarrow$
(*env_tsubst N e1* ++ *e0*) $\vdash$ ((*len* ([*e1*]) [->] *N*) *U*) : ((*len* ([*e1*]) [=>] *N*) *V*).
Proof.
Induction on derivation of e |- U : V.
*induction* 1.
T_Var case
*simpl*;*intros.*
*rewrite* $\leftarrow$ *head1_prop_0 with type e e0 t.*
*rewrite H0.*
*apply N_Decide with* (*len1 e*) (*len1 e1*);*intro.*
case *S len1 e* $\leq$ *len1 e1*
*rewrite* (*head1_cons2_gt* (*A*:=*type*) ) *with e1 e2 M* (*len1 e*);*auto.*
*rewrite* <-(*head1_map* (*A*:=*type*)) *with* (*f_tsubst N*) *e1 e2* (*len1 e*).
*rewrite* $\leftarrow$ *tsubst_lift_gt*;*auto.*
*apply T_Var'*;*auto.*
*apply app_tsubst_wfe with* (*N*:=*N*)(*e*:=*e1*)(*e0*:=*e2*)(*M*:=*M*);*eauto.*
*rewrite* $\leftarrow$*H0*;*eauto.*
*rewrite* (*head1_map* (*A*:=*type*)) *with* (*f_tsubst N*) *e1 e2* (*len1 e*).
*apply ine_app_gt*;*auto.*
*apply ine_app_cons_2 with M*;*auto.*
*rewrite* $\leftarrow$*H0*;*eauto.*
*apply ine_app_cons*;*auto.*
*rewrite head1_map.*
*apply head1_prop_2* ;*auto.*
case *len1 e1* $\leq$ *len1 e*
*rewrite* (*head1_cons2_le* (*A*:=*type*) ) *with e1 e2 M* (*len1 e*);*auto.*
*rewrite* <-(*head1_map* (*A*:=*type*)) *with* (*f_tsubst N*) *e1 e2* (*len1 e*).
*rewrite tsubst_lift_le*;*auto.*
*apply T_Var'*;*auto.*
*apply app_tsubst_wfe with* (*N*:=*N*)(*e*:=*e1*)(*e0*:=*e2*)(*M*:=*M*);*eauto.*
*rewrite* $\leftarrow$*H0*;*eauto.*
*apply ine_app_cons_2 with M*;*auto.*
*apply ine_app_le*;*auto.*
*rewrite* $\leftarrow$*H0*;*auto.*
*apply ine_app_cons*;*auto.*
*rewrite head1_map.*
*apply head1_prop_1*;*auto.*
T_Abs case
*unfold f_tsubst*;*simpl*;*intros*;*apply T_Abs*;*auto.*

*apply IHtyp with* $(e1{:=}(t1\ {:};\ e1))(e0{:=}\ e0)(M{:=}\ M)(\ N{:=}N)$;*auto.*
*simpl*;*rewrite* $\leftarrow$*H0*;*auto.*
T_App case
*simpl*;*intros*;*apply T_App with* (*f_tsubst N* (*len* ([*e1*])) *t11*);
*unfold f_tsubst in* $\times$;*simpl in* $\times$;*eauto.*
*unfold f_tsubst*;*simpl*;*intros*;*apply T_Tabs.*
*apply IHtyp with* $(e1{:=}(t1\ {;:}\ e1))(e0{:=}\ e0)(M{:=}\ M)(\ N{:=}N)$;*auto.*
*simpl*;*congruence.*
T_Tapp case
*intros*;*rewrite tsubst_tsubst_prop.*
*simpl*;*apply T_Tapp with* (*f_tsubst N* (*len* ([*e1*])) *t11*);*auto.*
*unfold f_tsubst in* $\times$;*simpl in* $\times$;*eauto.*
*rewrite list2_apps with* (*A*:=*type*)(*f*:=(*f_tsubst N*))(*e*:=*e1*);*auto.*
*apply tsubst_preserves_subtyp with*
(*e*:= [e1] ++ (*M* :: [e0]))(*e1*:=[*e1*])(*e0*:=[*e0*]) (*M*:=*M*)(*N*:=*N*)(*U*:=*t0*)(*V*:=*t11*);*auto.*
*rewrite H1 in H0.*
*rewrite list2list_app in H0.*
*rewrite list2list_cons in H0*;*auto.*
T_Sub case
*simpl*;*intros*;*apply T_Sub with* (*f_tsubst N* (*len* ([*e1*])) *t1*);*eauto.*
*rewrite list2_apps with* (*A*:=*type*)(*f*:=(*f_tsubst N*))(*e*:=*e1*);*auto.*
*apply tsubst_preserves_subtyp with*
(*e*:= [e1] ++ (*M* :: [e0]))(*e1*:=[*e1*])(*e0*:=[*e0*]) (*M*:=*M*)(*N*:=*N*);*auto.*
*rewrite H1 in H0.*
*rewrite list2list_app in H0.*
*rewrite list2list_cons in H0*;*auto.*
Qed.

Theorem *subst_type_preserves_typ*: $\forall\ e\ t\ U\ P\ Q$,
$Q\ {;:}\ e \vdash t : U \rightarrow [e] \vdash P <: Q \rightarrow e \vdash ((0\ [->]\ P)\ t) : ((0\ [=>]\ P)\ U)$.
Proof.
*intros e t U P Q H1 H2*;*intros.*
*apply subst_type_preserves_typ_ind with*
(*e*:=(*Q* ;: *e*))(*e0*:=*e*)(*M*:=*Q*)(*N*:=*P*)(*V*:=*U*)(*e1*:=*empty*);*b_auto.*
Qed.

# 13 Weakening Lemmas (A.4, A.5, A.6)

Lemma *typ_weakening_1_ind*:
$\forall$ *e t U*, $e \vdash t : U \rightarrow$
$\forall$ *e0 e1 T0*, *wft* ([*e0*]) *T0* $\rightarrow e = e1 ++ e0 \rightarrow$
$(e1 ++ T0 :; e0) \vdash$ (*shift* (*len1 e1*) *t*) : *U*.
Proof.
Induction on derivation of e |- t : U.
*induction* 1;*simpl*;*intros*;*auto*.
T_Var case
*rewrite* ←*head1_prop_0 with type e e0 t*.
*repeat* (*rewrite H1*).
*rewrite* <-(*head1_cons1* (*A*:=*type*)) *with e2* (*len1 e*) *e1 T0*;*auto*.
*apply T_Var'*; *auto*.
*apply env_app_cons_wf_2*;*auto*.
*rewrite* ←*H1*;*auto*.
*apply ine_app*;*auto*.
*rewrite* ←*H1*;*auto*.
*apply ine_app_cons*;*auto*.
T_Abs case
*apply T_Abs*.
*apply IHtyp with* (*e1*:=(*t1* :; *e1*))(*e0*:=*e0*)(*T0*:=*T0*);*simpl*;*b_auto*.
*rewrite H1*;*auto*.
T_App case
*apply T_App with t11*;*auto*.
T_Tabs case
*apply T_Tabs*.
*apply IHtyp with* (*e1*:=(*t1* ;: *e1*))(*e0*:=*e0*)(*T0*:=*T0*);*simpl*;*b_auto*.
*congruence*.
T_Tapp case
*apply T_Tapp with t11*;*auto*.
*rewrite list2list_app*;*simpl*;*rewrite* ←*list2list_app*;*rewrite* ← *H2*;*auto*.
T_Sub case
*apply T_Sub with t1*;*auto*.
*rewrite list2list_app*;*simpl*;*rewrite* ←*list2list_app*;*rewrite* ← *H2*;*auto*.
Qed.

Lemma *typ_weakening_1*: $\forall$ *e t U V*,
*wft* ([*e*]) $V \rightarrow e \vdash t : U \rightarrow (V :; e) \vdash (+\ t) : U$.
Proof.
*intros*;*apply typ_weakening_1_ind with* (*e*:=*e*)(*e0*:=*e*)(*e1*:=*empty*)(*T0*:=*V*);*auto*.
Qed.

To prove typing weakening lemma *typ_weakening_2*, we are to consider shifting for environments.

Definition $env\_tshift$:=$list2\_map\ tshift$.

Lemma $env\_app\_wf$: $\forall\ e\ T\ e0$,
$wft\ ([e0])\ T \rightarrow wfe\ (e\ ++\ e0) \rightarrow wfe\ (env\_tshift\ e\ ++\ T\ ;:\ e0)$.
Proof.
$induction\ e$; $simpl$;$intros$;$b\_auto$.
$rewrite\ list2\_apps\ with\ (A{:=}type)(f{:=}tshift)$;$auto$.
$simpl$;$apply\ app\_tshift\_wft$.
$rewrite \leftarrow list2list\_app$;$auto$.
$rewrite\ list2\_apps\ with\ (A{:=}type)(f{:=}tshift)$;$auto$.
$rewrite\ list2list\_cons$.
$apply\ app\_tshift\_wft$;$auto$.
$rewrite \leftarrow list2list\_app$;$auto$.
Qed.

Lemma $typ\_weakening\_2\_ind$:
$\forall\ e\ t\ U$,
$e \vdash t : U \rightarrow$
$\forall\ e0\ e1\ T0$,
$wft\ ([e0])\ T0 \rightarrow e = e1\ ++\ e0 \rightarrow$
$(env\_tshift\ e1\ ++\ T0\ ;:\ e0) \vdash (shift\_type\ (len\ ([e1]))\ t) : (tshift\ (len\ ([e1]))\ U)$.
Proof.
Induction on derivation of e |- t : U.
$induction$ 1;$simpl$;$intros$;$auto$.
T_Var case
$rewrite \leftarrow head1\_prop\_0\ with\ type\ e\ e0\ t$.
$rewrite\ H1$.
$apply\ N\_Decide\ with\ (len1\ e)\ (len1\ e2)$;$intro$.
case $S\ len1\ e \leq len1\ e2$
$rewrite \leftarrow (head1\_cons2\_gt\ (A{:=}type)\ )\ with\ e2\ e1\ T0\ (len1\ e)$;$auto$.
$rewrite$ <-$(head1\_map\ (A{:=}type))\ with\ tshift\ e2\ (T0\ ;:\ e1)\ (len1\ e)$.
$rewrite\ tshift\_lift$;$auto$.
$apply\ T\_Var'$;$simpl$;$auto$.
$apply\ env\_app\_wf$;$auto$;$rewrite \leftarrow H1$;$auto$.
$rewrite\ (head1\_map\ (A{:=}type))\ with\ tshift\ e2\ (T0\ ;:\ e1)\ (len1\ e)$.
$apply\ ine\_app\_gt$;$auto$.
$apply\ ine\_cons$;$rewrite \leftarrow H1$;$auto$.
$apply\ ine\_app\_cons$;$auto$.
$rewrite\ head1\_map$.
$apply\ head1\_prop\_2$ ;$auto$.
case $len1\ e2 \leq len1\ e$

*rewrite tshift_lift_le;auto.*
*rewrite ← (head1_cons2_le (A:=type) ) with e2 e1 T0 (len1 e);auto.*
*rewrite <-(head1_map (A:=type)) with tshift e2 (T0 ;: e1) (len1 e).*
*simpl;apply T_Var';simpl;b_auto.*
*apply env_app_wf;auto;rewrite ← H1;auto.*
*apply ine_app_le;auto.*
*apply ine_cons;rewrite ←H1;auto.*
*apply ine_app_cons;auto.*
*apply head1_prop_1;auto.*
T_Abs case
*apply T_Abs.*
*apply IHtyp with (e1:=(t1 :; e1))(e0:=e0)(T0:=T0);auto;rewrite H1;auto.*
T_App case
*apply T_App with (tshift (len ([e1])) t11);simpl in ×;auto.*
T_Tabs case
*apply T_Tabs;auto.*
*apply IHtyp with (e1:=(t1 ;: e1))(e0:=e0)(T0:=T0);auto;rewrite H1;auto.*
T_Tapp case
*simpl in ×;*
*rewrite tshift_tsubst_prop_2.*
*apply T_Tapp with (tshift (len ([e1])) t11);auto.*
*rewrite list2_apps with (A:=type)(f:=tshift)(e:=e1);auto.*
*rewrite list2list_cons.*
*apply sub_weakening_ind with ([e]);auto;rewrite H2;eauto.*
*apply list2list_app with (e:= e1) (e0:= e0).*
T_Sub case
*apply T_Sub with (tshift (len ([e1])) t1);auto.*
*rewrite list2_apps with (A:=type)(f:=tshift)(e:=e1);auto.*
*rewrite list2list_cons;apply sub_weakening_ind with ([e]);auto;rewrite H2;auto.*
*apply list2list_app with (e:= e1) (e0:= e0).*
Qed.

Lemma *typ_weakening_2*:
  ∀ *e t U V*,
  *wft* ([*e*]) *V* → *e* ⊢ *t* : *U* → (*V* ;: *e*) ⊢ (+ *t*) : + *U*.
Proof.
*intros e t U V H0 H.*
*apply typ_weakening_2_ind with (e:=e)(e0:=e)(e1:=empty)(T0:=V);auto.*
Qed.

# 14 Substitution Preserves Typing (Lemma A.8)

To prove *subst_preserves_typing* we are to consider more general case with additional concatenations of lists,by using extra environment *e1*.

Lemma *subst_preserves_typing0*:
  $\forall$ *e t U*, *e* $\vdash$ *t* : *U* $\rightarrow$
  $\forall$ *e0 e1 T u*,
  *e* = *e1* ++ *T* :; *e0* $\rightarrow$ *wft* ([*e0*]) *T* $\rightarrow$
  (*e1* ++ *e0*) $\vdash$ *u* : (*lift* (*len* ([*head1 e* (*len1 e1*)])) *T*) $\rightarrow$
  (*e1* ++ *e0*) $\vdash$ ((*len1 e1* [->] *u*) *t*) : *U*.

Proof.
*Opaque wfe.*
Induction on derivation of e |- t : U.
*induction* 1;*simpl*;*intros.*
T_Var case
*rewrite* $\leftarrow$*head1_prop_0 with type e e0 t.*
*apply Decide with* (*len1 e* == *len1 e2*);*intro HX.*
case *len1 e* == *len1 e2*.
*b_rewrite.*
*rewrite* (*n_eq_1 HX*);*auto.*
*rewrite* <-(*app2_inv H0*) *in H2*;*auto.*
case ¬ *len1 e* == *len1 e2*.
*b_rewrite.*
*rewrite H0*;*auto.*
*rewrite* (*head1_cons1_neq* (*A*:=*type*)) *with e2* (*len1 e*) *e1 T*;*auto.*
*apply T_Var'*;*eauto.*
*apply ine_app_2 with T*;*auto.*
*rewrite* $\leftarrow$*H0*;*auto.*
*apply ine_app_cons*;*auto.*
T_Abs case
*apply T_Abs.*
*apply IHtyp with* (*e1*:=(*t1* :; *e1*))(*e0*:=*e0*)(*T*:=*T*);*auto.*
*simpl*;*rewrite H0*;*simpl*;*auto.*
*simpl*;*apply typ_weakening_1*;*auto*;*apply wfe_cons1.*
*apply env_app_cons_wf with* (*e*:=(*t1* :; *e1*))(*e0*:=*e0*)(*T*:=*T*);*auto.*
*simpl*;*rewrite* $\leftarrow$*H0*;*eauto.*
T_App case
*apply T_App with t11*;*eauto.*
T_Tabs case
*apply T_Tabs*;*auto.*
*apply IHtyp with* (*e1*:=(*t1* ;: *e1*))(*e0*:=*e0*)(*T*:=*T*);*auto*;*rewrite H0*;*auto.*
*apply typ_weakening_2 with*

$(e{:=}e1 \; {+}{+} \; e0)(t{:=}u)(U{:=}lift \; (len \; ([head1 \; (e1 \; {+}{+} \; (T \; {:;} \; e0)) \; (len1 \; e1)])) \; T)(V{:=}t1)$;*auto.*
*apply wfe_cons2*;*apply env_app_cons_wf with* $(e{:=}(t1 \; {;:} \; e1))(e0{:=}e0)(T{:=}T)$;*auto.*
*simpl*;*rewrite* $\leftarrow$*H0*;*eauto.*
*rewrite H0 in H2*;*auto.*
T_Tapp case
*simpl*;*apply T_Tapp with t11*;*eauto.*
*rewrite list2list_app*;*replace* $([e0])$ *with* $([T \; {:;} \; e0])$;*auto.*
*rewrite* $\leftarrow$*list2list_app*;*rewrite* $\leftarrow$ *H1*;*auto.*
T_Sub case
*simpl*;*apply T_Sub with t1*;*eauto.*
*rewrite list2list_app*;*replace* $([e0])$ *with* $([T \; {:;} \; e0])$;*auto.*
*rewrite* $\leftarrow$*list2list_app*;*rewrite* $\leftarrow$ *H1*;*auto.*
*Transparent wfe.*
Qed.

Lemma *subst_preserves_typing*:
$\forall \; e \; t \; V \; t1, \; t1 \; {:;} \; e \vdash t : V \rightarrow$
$\forall \; u, \; e \vdash u : t1 \rightarrow e \vdash ((0 \; [\text{->}] \; u) \; t) : V.$
Proof.
*intros e t V t1 H1 u H2.*
*apply subst_preserves_typing0 with*
$(e{:=}t1 \; {:;} \; e)(e0{:=}e)(e1{:=}empty)(T{:=}t1)$;*simpl*;*auto.*
*apply wfe_cons1*;*eauto.*
Qed.

# 15 Narrowing for the Typing (Lemma A.7)

To prove *typ_narrowing* we are to consider more general case with additional concatenations of lists, by using extra environment *e1*.

Lemma *typ_narrowing_ind*:
$\forall$ *e t U*, *e* $\vdash$ *t* : *U* $\rightarrow$
$\forall$ *e1 e0 M N*, *e* = *e1* ++ *M* ;: *e0* $\rightarrow$ [e0]$\vdash$ *N* <: *M* $\rightarrow$ (*e1* ++ *N* ;: *e0*) $\vdash$ *t* : *U*.
Proof.
Induction on derivation of e |- t : U.
*induction* 1;*intros*.
T_Var case
*rewrite* $\leftarrow$*head1_prop_0 with type e e0 t*.
*rewrite H0*;*auto*.
*rewrite head1_cons2 with* (*A*:=*type*) (*e*:=*e1*)(*e0*:= *e2*)(*M*:= *M*)(*N*:= *N*)(*X*:= (*len1 e*)).
*apply T_Var'*;*auto*.
*apply env_app_wfe with M*;*eauto*.
*rewrite* $\leftarrow$*H0*;*auto*.
*apply ine_cons*;*apply ine_app_cons_2 with M*;*auto*;*rewrite* $\leftarrow$ *H0*;*auto*.
*apply ine_app_cons*;*auto*.
T_Abs case
*apply T_Abs*.
*apply IHtyp with* (*e1*:=(*t1* :; *e1*))(*e0*:= *e0*)(*M*:= *M*)( *N*:=*N*);*auto*.
*simpl*;*rewrite H0*;*auto*.
T_App case
*eapply T_App*; *eauto*.
T_Tabs case
*apply T_Tabs*.
*apply IHtyp with* (*e1*:=(*t1* ;: *e1*))(*e0*:= *e0*)(*M*:= *M*)( *N*:=*N*);*auto*.
*simpl*;*rewrite H0*;*auto*.
T_Tapp case
*eapply T_Tapp*; *eauto*.
*rewrite list2list_app with* (*e*:= *e1*) (*e0*:= (*N* ;: *e0*));*rewrite list2list_cons*;*auto*.
*apply sub_narrowing with M*;*auto*.
*apply list_app_wfl with M*;*eauto*.
*rewrite* $\leftarrow$*list2list_cons*;*rewrite* $\leftarrow$*list2list_app with* (*e*:= *e1*) (*e0*:= (*M* ;: *e0*));*auto*.
*apply wf_lst*.
*rewrite* $\leftarrow$*H1*;*eauto*.
*rewrite H1 in H0*;
*rewrite list2list_app with* (*e*:= *e1*) (*e0*:= (*M* ;: *e0*)) *in H0*;*auto*.
T_Sub case
*apply T_Sub with t1*;*eauto*.
*rewrite list2list_app with* (*e*:= *e1*) (*e0*:= (*N* ;: *e0*));*rewrite list2list_cons*;*auto*.

*apply sub_narrowing with M;auto.*
*apply list_app_wfl with M;eauto.*
*rewrite* $\leftarrow$*list2list_cons;rewrite* $\leftarrow$*list2list_app with (e:= e1) (e0:= (M ;: e0));auto.*
*apply wf_lst.*
*rewrite* $\leftarrow$*H1;eauto.*
*rewrite* $\leftarrow$*list2list_cons;rewrite* $\leftarrow$*list2list_app with (e:= e1) (e0:= (M ;: e0));auto.*
*rewrite* $\leftarrow$*H1;eauto.*
Qed.

Lemma *typ_narrowing*: $\forall$ *e M N t U*,
[e]$\vdash$ *N <: M ->(M ;: e)* $\vdash$ *t : U* $\rightarrow$ *(N ;: e)* $\vdash$ *t : U.*
Proof.
*intros;apply typ_narrowing_ind with (e:=(M ;: e))(e1:=empty)(e0:=e)(M:=M)(N:=N);*
*simpl;b_auto.*
Qed.

# 16 Inversions of Typing Rules

Following results looks to be new, at least no analogs given in the paper proof.

Lemma *typ_inv1*: $\forall$ *e T t0*,
   $e \vdash t0 : T \to$
   $\forall$ *t1 t2 T1 T2 T3*,
   $t0 = (T1.\!\to t1) \to [\mathrm{e}] \vdash T <: (T2 -> T3) \to$
   $e \vdash t2 : T2 \to e \vdash ((0 \,[\text{->}]\, t2)\ t1) : T3.$
Proof.
*induction* 1;*intros*; *try discriminate.*
*inversion_clear H1*;*auto.*
*apply T_Sub with t2*;*auto.*
*apply subst_preserves_typing with t1*;*auto.*
*injection H0*; *intros E2 E3*;*rewrite* $\leftarrow$ *E2*;*auto.*
*apply T_Sub with T2*;*auto.*
*apply IHtyp with T1 T2*;*auto*;*apply sub_transitivity with t2*;*auto.*
Qed.

More general form of the typing rule for term substitutions.

Lemma *typ_inversion1*: $\forall$ *e t1 t2 T1 T2 T3*,
   $e \vdash (T1.\!\to t1) : (T2 -> T3) \to$
   $e \vdash t2 : T2 \to e \vdash ((0 \,[\text{->}]\, t2)\ t1) : T3.$
Proof.
*intros e t1 t2 T1 T2 T3*;*intros.*
*apply typ_inv1 with* $(T2 -> T3)$ $(T1.\!\to t1)$ *T1 T2*;*auto.*
*apply sub_reflexivity*;*eauto.*
*apply wf_lst*;*eauto.*
Qed.

To prove *typ_inversion3*, we are to consider more general result. The usage of subtyping is essential to enable induction, while only special case formulated in *typ_inversion3* will be required.

Lemma *typ_inv2*: $\forall$ *e T t0*,
   $e \vdash t0 : T \to$
   $\forall$ *t1 T1 T2 T3*,
   $t0 = (T1.\!\Rightarrow t1) \to [\mathrm{e}] \vdash T <: (T2 -> T3) \to \mathit{false}.$
Proof.
*induction* 1;*intros*; *try discriminate.*
*inversion_clear H1*;*auto.*
*apply IHtyp with t0 T1 T2 T3*;*auto.*
*apply sub_transitivity with t2*;*auto.*
Qed.

Lemma $typ\_inversion2$: $\forall$ $e$ $T$ $t1$ $T1$ $T2$ $T3$,
$e \vdash (T1. \Rightarrow t1) : T \rightarrow [\mathrm{e}] \vdash T <: (T2 -> T3) \rightarrow false.$
Proof.
*intros e T t1 T1 T2 T3;intros.*
*apply typ_inv2 with e T (T1.⇒ t1) t1 T1 T2 T3;auto.*
Qed.

Lemma $typ\_inversion3$: $\forall$ $e$ $t1$ $T1$ $T2$ $T3$,
$e \vdash (T1. \Rightarrow t1) : (T2 -> T3) \rightarrow false.$
Proof.
*intros e t1 T1 T2 T3;intros.*
*apply typ_inversion2 with e (T2 −> T3) t1 T1 T2 T3;auto.*
*apply sub_reflexivity;eauto.*
*apply wf_lst;eauto.*
Qed.

Lemma $T\_typ\_inv1$: $\forall$ $e$ $T$ $t0$, $e \vdash t0 : T \rightarrow$
$\forall$ $t$ $T1$ $T2$ $T3$,
$t0 = (T1. \Rightarrow t) \rightarrow [\mathrm{e}] \vdash T <: (T2 \ . \ T3) \rightarrow (T2 \ ;: \ e) \vdash t : T3.$
Proof.
*induction* 1;*intros*; *try discriminate.*
*inversion_clear H1;auto.*
*apply T_Sub with t2;auto.*
*apply typ_narrowing with t1;auto.*
*injection H0; intros E2 E3; rewrite ← E2;auto.*
*apply IHtyp with T1;auto;apply sub_transitivity with t2; auto.*
Qed.

Some kind of inversion for $T\_Tabs$ rule.

Lemma $T\_typ\_inversion1$: $\forall$ $e$ $t$ $T1$ $T2$ $T3$,
$e \vdash (T1. \Rightarrow t) : (T2 \ . \ T3) \rightarrow (T2 \ ;: \ e) \vdash t : T3.$
Proof.
*intros e t T1 T2 T3;intros.*
*apply T_typ_inv1 with (T2 . T3) (T1.⇒ t) T1;auto.*
*apply sub_reflexivity;eauto.*
*apply wf_lst;eauto.*
Qed.

To prove $T\_typ\_inversion3$, we are to consider more general result. The usage of subtyping is essential to enable induction, while only special case formulated in $T\_typ\_inversion3$ will be required.

Lemma $T\_typ\_inv2$: $\forall\ e\ T\ t0$,

$e \vdash t0\ :\ T \rightarrow$

$\forall\ t\ T1\ T2\ T3$,

$t0 = (T1.\!\rightarrow t) \rightarrow [\mathrm{e}] \vdash T <: (T2\ .\ T3) \rightarrow false.$

Proof.

*induction* 1;*intros*; *try discriminate.*

*inversion_clear H1*;*auto.*

*apply IHtyp with t0 T1 T2 T3*;*auto.*

*apply sub_transitivity with t2*;*auto.*

Qed.

Lemma $T\_typ\_inversion2$: $\forall\ e\ T\ t\ T1\ T2\ T3$,

$e \vdash (T1.\!\rightarrow t)\ :\ T \rightarrow [\mathrm{e}] \vdash T <: (T2\ .\ T3) \rightarrow false.$

Proof.

*intros e T t T1 T2 T3*;*intros.*

*apply T_typ_inv2 with e T (T1.→ t) t T1 T2 T3*;*auto.*

Qed.

Lemma $T\_typ\_inversion3$: $\forall\ e\ t\ T1\ T2\ T3$,

$e \vdash (T1.\!\rightarrow t)\ :\ (T2\ .\ T3) \rightarrow false.$

Proof.

*intros e t T1 T2 T3*;*intros.*

*apply T_typ_inversion2 with e (T2 . T3) t T1 T2 T3*;*auto.*

*apply sub_reflexivity*;*eauto.*

*apply wf_lst*;*eauto.*

Qed.

# 17 Typing is Preserved by Reduction (Theorems 3.3, 3.4)

Progress operator to perform reduction. Reduction is not changing arguments for abstraction values.

```
Fixpoint progr (t:term){struct t}:term:=
    match t with
    | (t1.→ t2) (s1.→ s2) ⇒ (0 [->] (s1.→ s2)) t2
    | (t1.→ t2) (s1.⇒ s2) ⇒ (0 [->] (s1.⇒ s2)) t2
    | (t1.→ t2) s ⇒ (t1.→ t2) (progr s)
    | (t1.⇒ t2) (s1.→ s2) ⇒ (0 [->] (s1.→ s2)) (t1.→ t2)
    | (t1.⇒ t2) (s1.⇒ s2) ⇒ (0 [->] (s1.⇒ s2)) (t1.→ t2)
    | (t1.⇒ t2) s ⇒ (t1.⇒ t2) (progr s)
    | t s ⇒ (progr t) s
    | (t1.→ t2) [s] ⇒ (0 [->] s) (t1.→ t2)
    | (t1.⇒ t2) [s] ⇒ (0 [->] s) t2
    | t [s] ⇒ (progr t) [s]
    | _ ⇒ t
    end.
```

Main result - operator *progr* is compatible with typing relation. In the paper proof, the progress theorem is given only for empty environments, so this result could be considered as more general. No need for separate notion of evaluation relation (and evaluation contexts), for terms *t1*, *t2* being related with such relations, one just can impose restrictions of *t1* not being a value and *t2* being equal to *progr t1*. The prove is performed directly on the form of operator *progr* without introducing any additional relations.

```
Theorem preservation:
    ∀ e t U, e ⊢ t : U →e ⊢ (progr t) : U.
Proof.
```

Induction on derivation of e |- t : U.

*induction* 1;*intros*.

T_Var case

*simpl*;*apply T_Var*;*auto*.

T_Abs case

*simpl*;*apply T_Abs*;*auto*.

T_App case

```
induction t1;simpl;intros;auto.
apply T_App with t11;auto.
induction t2;intros;auto.
apply T_App with t11;auto.
apply typ_inversion1 with t t11;auto.
apply T_App with t11;auto.
apply typ_inversion1 with t t11;auto.
apply T_App with t11;auto.
```

*apply T_App with t11*;*auto.*
*induction t2*;*intros*;*auto.*
*apply T_App with t11*;*auto.*
*apply Contradiction*;*apply typ_inversion3 with e t1 t t11 t12*;*auto.*
*apply T_App with t11*;*auto.*
*apply Contradiction*;*apply typ_inversion3 with e t1 t t11 t12*;*auto.*
*apply T_App with t11*;*auto.*
*apply T_App with t11*;*auto.*
T_Tabs case
*simpl*;*apply T_Tabs*;*auto.*
T_Tapp case
*induction t*;*simpl*;*intros*;*auto.*
*apply T_Tapp with t11*;*auto.*
*apply Contradiction*;*apply T_typ_inversion3 with e t1 t t11 t12*;*auto.*
*apply T_Tapp with t11*;*auto.*
*apply subst_type_preserves_typ with t11*;*auto.*
*apply T_typ_inversion1 with t*;*auto.*
*apply T_Tapp with t11*;*auto.*
T_Sub case
*apply T_Sub with t1*; *auto.*
Qed.

# 18 Notations

Notations are listed in the order of their first appearence in the library or main part.

$bool : Set$

$true : bool$

$false : bool$

$TRUE : bool \to Prop$

$and : bool \to bool \to bool$

$(x \&\& y) = (and\ x\ y)$

$or : bool \to bool \to bool$

$(x \mid\mid y) = (or\ x\ y)$

$nat : Set$

$0 = O : nat$

$S : nat \to nat$

$nat\_le$: $nat \to nat \to bool$

$(x \leq y) = (nat\_le\ x\ y)$ (order relation for natural numbers)

$n\_eq$: $nat \to nat \to bool$

$(x == y) = (n\_eq\ x\ y)$ (equality of natural numbers)

$max : nat \to nat \to nat$

$S\_Next$: $nat \to nat \to nat$

$S\_Pred$: $nat \to nat \to nat$

$inl : list \to nat \to A \to bool$

(relation $inl\ e\ X\ a$ means the type $a$ is in the list $e$ at position $X$)

$len : list \to nat$ (length of a list)

$head : list \to nat \to list$

$tail : list \to nat \to list$

$wfi$: $list \to nat \to bool$

(well-formed indices of lists, $wfi\ e\ X <=> X < (len\ e)$)

$lel : list \to list \to bool$ $(lel\ e1\ e2 <=> (len\ e1) \leq (len\ e2))$

$cons : A \to list \to list$ (addition of an object to a list)

$(T :: e) = cons\ T\ e$

$app$ :$list \to list \to list$ (concatenation of lists)

$(e1\ ++\ e2) = (app\ e1\ \ e2)$

$list\_map$: $(nat \to A \to A)$-> $list \to list$

$list2$ : $Set$

$nil2$ : $list2$

$cons1$:$A \to list2 \to list2$

$(x$ :; $y) = (cons1\ \ x\ \ y)$

$cons2$:$A \to list2 \to list2$

$(x$ ;: $y) = (cons2\ \ x\ \ y)$

$len1$: $list2 \to nat$ (the number of term binders within a list)

$ine$: $list2 \to nat \to A \to bool$

(relation $ine\ \ e\ \ X\ \ a$ means the type $a$ is in the list $e$ at position $X$ of term binders)

$head1$: $list2 \to nat \to list2$

$tail1$: $list2 \to nat \to list2$

$lst$: $list2 \to list$

$[e] = lst\ \ e$ (image of (e : list2) in list, after all term binders removed)

$list2\_app$ :$list2 \to list2 \to list2$ (concatenation of lists)

$x ++ y = list2\_app\ \ x\ \ y$

$list2\_map$: $(nat \to A \to A)$-> $list2 \to list2$

$type$:$Set$

$ref$ :$nat \to type$ (type of a type binder)

$top$ :$type$

$arrow$:$type \to type \to type$ (type of lambda abstracions)

$(x -> y) = arrow\ \ x\ \ y$

$all$ :$type \to type \to type$ (type of applications)

$(x\ .\ y) = all\ \ x\ \ y$

$tshift$: $nat \to type \to type$ (type shifting)

$(+\ x) = tshift\ 0\ x$

$lift$ :$nat \to type \to type$ ($lift\ \ X\ \ T$ = X times applied operator $+$)

$tsubst$ :$type \to nat \to type \to type$ (types substitution in types)

$f\_tsubst$ :$type \to nat \to type \to type$ (another form for types substitution in types)

$(x\ [=>]\ T)\ t = f\_tsubst\ T\ x\ t$

$type\_env = list\ type$

$Nil = nil\ (A{:=}type)$

$wft\ {:}type\_env \rightarrow type \rightarrow bool$ (well-formed types)

$wfl\ {:}type\_env \rightarrow bool$ (well-formed type environments)

$list\_tshift = list\_map\ tshift$

$list\_tsubst = fun\ (T{:}type) \Rightarrow list\_map\ (f\_tsubst\ T)$

$env = list2\ type$ (environments of binders for types and terms)

$empty = nil2\ type$

$wfe\ {:}env \rightarrow bool$ (well-formed type environments of types and terms binders)

$env\_tshift = list2\_map\ tshift$

$env\_tsubst = fun\ (T{:}type) \Rightarrow list2\_map\ (f\_tsubst\ T)$

$sub\ :\ type\_env \rightarrow type \rightarrow type \rightarrow Prop$ (subtyping relation)

$(e \vdash x <:\ y) = sub\ e\ x\ y$

$depth{:}\ type \rightarrow nat$ (structural depth of a type)

$term{:}Set$

$var{:}\ nat \rightarrow term$ (term of a term binder)

$abs{:}\ type \rightarrow term \rightarrow term$ (lambda abstraction of terms)

$(x\ .\!\!\rightarrow y) = abs\ x\ y$

$apl{:}\ term \rightarrow term \rightarrow term$ (application of terms)

$(x\ y) = apl\ x\ y$

$tabs{:}type \rightarrow term \rightarrow term$ (lambda abstraction of types)

$(x\ .\!\!\Rightarrow y) = tabs\ x\ y$

$tapl{:}term \rightarrow type \rightarrow term$ (application of types)

$(\ x\ [y]) = tapl\ x\ y$

$shift\ {:}nat \rightarrow term \rightarrow term$ (shifting of terms)

$(+\ x) = shift\ 0\ x$

$shift\_type{:}type\_env \rightarrow term \rightarrow term$ (shifting of types in terms)

$(+\ x) = shift\_type\ 0\ x$

$subst\_type;\ term \rightarrow type\_env \rightarrow type \rightarrow term$ (substitution of types in terms)

$(x\ [->]\ T)\ t) = subst\_type\ T\ x\ t$

$subst$:$term \to nat \to term \to term$ (substitution of terms)

$(x$ [->] $s)$ $t)$ $=subst\ s\ x\ t$

$typ$ : $env \to term \to type \to Prop$ (typing relation)

$(e \vdash x : y) = typ\ e\ x\ y$

$progr$: $term \to term$ (progress operator)