CIS 500 — Software Foundations

Final
(Advanced version)

April 30, 2013

Name: _____________________________________________________________

Pennkey (e.g. bcspierc) : ____________________________________________

Scores:

| 1 | 12 |
| 2 | 12 |
| 3 | 10 |
| 4 | 12 |
| 5 | 10 |
| 6 | 14 |
| 7 | 15 |
| 8 | 10 |
| 9 | 20 |
| 10 | 5 |
| Total: | 120 |
1. (12 points) Recall the definition of the fold function on lists (given in the appendix, page 1). Use it to define a function sum_list_list that, given a list (list nat), returns the sum of all the nats in it. E.g.

\[
\text{sum_list_list }[[1,3],[5,6]] = 15
\]

(Note that the first line we've given you starts with Definition, not Fixpoint: your solution should be non-recursive.)

Definition sum_list_list (l : list (list nat)) : nat :=
2. (12 points) Complete the definition at the bottom of the page of an inductive proposition `count` that counts the number of elements of a list satisfying some predicate `P`. For example, if we define

\[
\text{Definition iszero } (n : \text{nat}) : \text{Prop} := \\
n = 0.
\]

then the propositions

\[
\begin{align*}
\text{count iszero } [] & \text{ 0} \\
\text{count iszero } [1,2,3] & \text{ 0} \\
\text{count iszero } [0,1,2,3] & \text{ 1} \\
\text{count iszero } [1,0,0,2,3,0] & \text{ 3}
\end{align*}
\]

should all be provable, whereas the propositions

\[
\begin{align*}
\text{count iszero } [1,2,3] & \text{ 3} \\
\text{count iszero } [0,0] & \text{ 4} \\
\text{count iszero } [] & \text{ 1}
\end{align*}
\]

should not be provable.

\[
\text{Inductive count } (X : \text{Type}) (P : X \to \text{Prop}) : \text{list } X \to \text{nat} \to \text{Prop} :=
\]
3. (10 points) For each of the types below, write a Coq expression that has that type. (For example, if the type given were \texttt{nat->nat}, you might write \texttt{fun x:nat, x+5.}) The definitions of \texttt{\&\&} and \texttt{\|/} are given in the appendix (page 2) for reference.

(a) \texttt{forall X Y, (X -> Y) -> X -> list Y}

(b) \texttt{forall (X : Type), (X -> Prop) -> Prop}

(c) \texttt{forall (X Y : Prop), X -> Y -> (X \|/ Y) \&/ Y}

(d) \texttt{forall (X Y W Z : Prop), (X \&/ Y) \|/ (W \&/ Z) -> (X \|/ Z)}
4. (12 points) The following Imp program sets \( R \) to the sum of the initial values of \( X \) and \( Y \) modulo the initial value of \( Z \):

\[
R ::= X + Y;
\]

\[
\text{WHILE } Z \leq R \text{ DO}
\]

\[
R ::= R - Z
\]

\[
\text{END}
\]

Your task is to fill in assertions to make this a well-decorated program (see page 3 of the appendix), relative to an appropriate and post-condition. Please fill in annotations in the spaces provided below.

\[
\begin{align*}
\{\text{ True} \} &\implies \{\} \\
\{\} &\implies \{\} \\
R ::= X + Y; &\implies \{\} \\
\{\} &\implies \{\} \\
\text{WHILE } Z \leq R \text{ DO} &\implies \{\} \\
\{\} &\implies \{\} \\
R ::= R - Z &\implies \{\} \\
\{\} &\implies \{\} \\
\text{END} &\implies \{\} \\
\{\} &\implies \{\} \\
\{ R = (X + Y) \mod Z \} &\implies \{\}
\end{align*}
\]

Finally, please mark (by circling the associated arrow \(\implies\) and writing either 1 or 2 or both, as appropriate) any places where the following facts about modular arithmetic are needed to validate the correctness of the decorations:

\[
\begin{align*}
(1) \quad a \mod z &= (a + z) \mod z \\
(2) \quad a < z \implies (a \mod z) &= a
\end{align*}
\]
5. (10 points) A triangular number is one that can be expressed as a sum of the form

\[ 1 + 2 + 3 + \ldots + n \]

for some \( n \). In Coq notation:

\[
\text{Fixpoint tri (n : nat) : nat :=} \\
\text{match n with} \\
\text{ 0 => 0} \\
\text{  S n' => n + (tri n')} \\
\text{end.}
\]

\[
\text{Definition triangular (t : nat) : Prop :=} \\
\text{ exists n, t = tri n.}
\]

The following Imp program will terminate only when the initial value of the variable \( T \) is a triangular number.

\[
\begin{align*}
\text{{{ True }}} \\
\text{X ::= 0;} \\
\text{A ::= 0;} \\
\text{WHILE A <> T DO} \\
\text{X ::= X + 1;} \\
\text{A ::= A + X;} \\
\text{{{ I }}} \\
\text{END} \\
\text{{{ triangular T }}}
\end{align*}
\]

In the space below, write a suitable invariant for this loop — i.e., an assertion \( I \) such that, if we add \( I \) as the annotation at the end of the loop as shown, we can fill in annotations on the rest of the program to make it well decorated. (The spaces for these additional annotations are not shown, and you do not need to provide them.)

\[ I = \]
6. (14 points) Consider the following two programs:

\[
\text{P1} = \text{WHILE } b \text{ DO } c \text{ END} \\
\text{P2} = \text{IFB } b \text{ THEN } c \text{ ELSE SKIP FI; P1}
\]

Give a careful informal proof that P1 and P2 are equivalent.
7. (15 points) Suppose \( L \) is some variant of STLC with the same syntax but in which the typing and single-step reduction rules have been changed in some way (by adding, removing, or changing rules). Consider the following schematic statement:

“If the (A) theorem (B) for \( L \), then changing \( L \)’s (C) relation by (D) a rule might cause it to (E).”

If we substitute “preservation” for (A), “holds” for (B), “typing” for (C), “adding” for (D), and “fail” for (E), we obtain the statement “If the preservation theorem holds for \( L \), then changing \( L \)’s typing relation by adding a rule might cause it to fail,” which happens to be true: adding a rule to the typing relation of a language like the STLC can sometimes destroy the preservation property.

Fill in the last column of the following table to indicate whether the statement obtained by similarly substituting the values in columns (A) to (E) is true or false. (We’ve done the first one for you.)

<table>
<thead>
<tr>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
<th>(E)</th>
<th>T/F</th>
</tr>
</thead>
<tbody>
<tr>
<td>preservation holds typing</td>
<td>adding</td>
<td>fail</td>
<td></td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>preservation holds single-step reduction removing fail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>preservation holds typing removing fail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>preservation holds single-step reduction adding fail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress holds typing removing fail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress holds single-step reduction removing fail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress holds typing adding fail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress holds single-step reduction adding fail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>preservation fails typing removing hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>preservation fails single-step reduction removing hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>preservation fails typing adding hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>preservation fails single-step reduction adding hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress fails typing removing hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress fails single-step reduction removing hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress fails typing adding hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>progress fails single-step reduction adding hold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7
8. (10 points) Consider the following propositions about typing in STLC. For each one, either give values for the existentially quantified variables to make the term typeable, or write “not typeable” and briefly explain why not.

(a) There exist types $T$ and $T_1$ such that

$$\text{empty } \vdash (\lambda x : T. \text{if } x \text{ then true else } x) \in T_1$$

$T =$

$T_1 =$

(b) For some types $T_2$ and $T_1$,

$$y : T_2 \vdash (\lambda x : T_1. x (y (y x) y)) \in \text{Bool}$$

$T_2 =$

(c) For all types $T$, there exist types $T_1$ and $T_2$ such that

$$y : T_2, x : T_1 \vdash (y (x \text{ true}) x) \in T$$

$T_1 =$

$T_2 =$

(d) For all types $T_2$ and $T_1$, there exists a type $T$ such that

$$y : T_2 \vdash (\lambda x : T_1. \text{if } y x \text{ then true else } x) \in T$$

$T =$

(e) There exist types $T_1$ and $T_2$ such that

$$\text{empty } \vdash (\lambda y : T_2. \lambda x : T_1. \text{if } x \text{ then } y (y x) \text{ else } x) \in \text{Bool}$$

$T_1 =$

$T_2 =$
9. (20 points) Your job in this problem is to fill in part of the proof of a lemma needed for the Preservation theorem in the STLC. (Note that the language we are considering here is the standard SLTC, without subtyping—see page 8 of the appendix.)

The lemma in question is the one stating that substitution preserves typing in STLC:

Theorem substitution_preserves_typing :
  forall (t t' : tm) (x : id) (Γ : context) (T T' : ty),
  Γ, x:T' |- t ∈ T ->
  empty |- t' ∈ T' ->
  Γ |- [x := t'] t ∈ T.

On the next two pages, you will find an informal proof of this theorem, with two cases on the second page omitted. Fill in the argument for the missing cases.

Your may find it useful at some point(s) to invoke the following lemma (which you do not need to prove):

Lemma context_invariance : forall Γ Γ' t T,
  Γ |- t ∈ T ->
  (forall x, appears_free_in x t -> Γ x = Γ' x) ->
  Γ' |- t ∈ T.
Proof: By induction on $t$. We have the following cases to consider:

- **$t = \text{true}$**. In this case, we have $[x := t'] \text{true} = \text{true}$ by the definition of substitution. By the rule $T_\text{True}$ the conclusion follows.

- **$t = \text{false}$**. Similar.

- **$t = t_1 \ t_2$**. In this case, we have

  \[
  [x := t'] (t_1 \ t_2) \\
  = ([x := t'] t_1) ([x := t'] t_2)
  \]

  by the definition of substitution. By inversion on $\Gamma, x:T' \vdash t \in T$ we know that there exists a $T_1$ such that

  \[
  \Gamma, x:T' \vdash t_1 \in T_1 \rightarrow T \quad (1) \\
  \Gamma, x:T' \vdash t_2 \in T_1 \quad (2)
  \]

  In order to apply rule $T_\text{App}$ to build the desired derivation, we have to show the following conditions:

  - $\Gamma \vdash [x := t'] t_1 \in T_1 \rightarrow T$. This follows from (1) by the IH.
  - $\Gamma \vdash [x := t'] t_2 \in T_1$. This follows from (2) by the IH.

- **$t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$**. This case is similar to the one for application.

  *See next page for remaining cases...*
• $t = y$.
  
  *Fill in the rest...*

• $t = \lambda y : T_1. t_1$.
  
  *Fill in the rest...*
10. (5 points) The STLC with records and subtyping is summarized in the appendix for reference (page 10). Indicate whether each of the following claims is true or false for this language.

(a) If $T <: S$, then $(T -> U) <: (S -> U)$.

   T   F

(b) The subtype relation contains an infinite descending chain — that is, there is an infinite sequence of types $T_1, T_2, T_3, ...$ such that, for each $i$, we have $T_{i+1} <: T_i$ but not $T_i <: T_{i+1}$.

   T   F

(c) There is a record type $T$ that is a supertype of every other record type (that is, $S <: T$ for every record type $S$).

   T   F

(d) There is a type $T$ that is a subtype of every other type (that is, $T <: S$ for every type $S$).

   T   F

(e) There is a record type $T$ that is a subtype of every other record type (that is, $T <: S$ for every record type $S$).

   T   F
For Reference...

Some functions on lists

Fixpoint map \{X Y:Type\} (f:X\rightarrow Y) (l:list X) : list Y :=
  match l with
  | [] => []
  | h :: t => (f h) :: (map f t)
end.

Fixpoint fold \{X Y:Type\} (f:X\rightarrow Y\rightarrow Y) (y:Y) (l:list X) : Y :=
  match l with
  | [] => y
  | h :: t => f h (fold f y t)
end.
Definitions of logical connectives in Coq

Inductive and \((P \land Q : \text{Prop}) : \text{Prop} :=
\begin{align*}
\text{conj} & : P \to Q \to (P \land Q) . \\
\end{align*}

Inductive or \((P \lor Q : \text{Prop}) : \text{Prop} :=
\begin{align*}
| \text{or_introl} & : P \to P \lor Q \\
| \text{or_intror} & : Q \to P \lor Q . \\
\end{align*}

Notation "\(P \land Q" := (P \land Q) : \text{type_scope}.
Notation "\(P \lor Q" := (P \lor Q) : \text{type_scope}.
Formal definitions for Imp

Syntax

Inductive aexp : Type :=
    | ANum : nat -> aexp
    | AId : id -> aexp
    | APlus : aexp -> aexp -> aexp
    | AMinus : aexp -> aexp -> aexp
    | AMult : aexp -> aexp -> aexp.

Inductive bexp : Type :=
    | BTrue : bexp
    | BFalse : bexp
    | BEq : aexp -> aexp -> bexp
    | BLe : aexp -> aexp -> bexp
    | BNot : bexp -> bexp
    | BAnd : bexp -> bexp -> bexp.

Inductive com : Type :=
    | CSkip : com
    | CAss : id -> aexp -> com
    | CSeq : com -> com -> com
    | CIf : bexp -> com -> com -> com
    | CWhile : bexp -> com -> com.

Notation "'SKIP'" :=
    CSkip.
Notation "X '::=' a" :=
    (CAss X a) (at level 60).
Notation "c1 ; c2" :=
    (CSeq c1 c2) (at level 80, right associativity).
Notation "'WHILE' b 'DO' c 'END'" :=
    (CWhile b c) (at level 80, right associativity).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
    (CIf e1 e2 e3) (at level 80, right associativity).
Evaluation relation

Inductive ceval : com -> state -> state -> Prop :=
  | E_Skip : forall st,
    SKIP / st || st
  | E_Ass : forall st a1 n X,
    aeval st a1 = n ->
    (X ::= a1) / st || (update st X n)
  | E_Seq : forall c1 c2 st st' st'',
    c1 / st || st' ->
    c2 / st' || st'' ->
    (c1 ; c2) / st || st''
  | E_IfTrue : forall st st' b1 c1 c2,
    beval st b1 = true ->
    c1 / st || st' ->
    (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_IfFalse : forall st st' b1 c1 c2,
    beval st b1 = false ->
    c2 / st || st' ->
    (IFB b1 THEN c1 ELSE c2 FI) / st || st'
  | E_WhileEnd : forall b1 st c1,
    beval st b1 = false ->
    (WHILE b1 DO c1 END) / st || st
  | E_WhileLoop : forall st st' st'' b1 c1,
    beval st b1 = true ->
    c1 / st || st' ->
    (WHILE b1 DO c1 END) / st' || st'' ->
    (WHILE b1 DO c1 END) / st || st''

where "c1 '/' st '||' st'" := (ceval c1 st st').

Program equivalence

Definition bequiv (b1 b2 : bexp) : Prop :=
  forall (st:state), beval st b1 = beval st b2.

Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
  (c1 / st || st') <-> (c2 / st || st').
Hoare triples

Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
forall st st', c / st || st' -> P st -> Q st'.

Notation "{{ P }} c {{ Q }}" := (hoare_triple P c Q)
(at level 90, c at next level)
: hoare_spec_scope.

Implication on assertions

Definition assert_implies (P Q : Assertion) : Prop :=
forall st, P st -> Q st.

Notation "P => Q" := (assert_implies P Q) (at level 80).

Hoare logic rules

\[
\begin{align*}
\text{\{ assn_sub X a Q \} X := a & Q \} & \quad \text{(hoare_asgn)} \\
\text{\{ P \} SKIP \{ P \} & \quad \text{(hoare_skip)} \\
\text{\{ P \} c1 \{ Q \} & \text{\{ Q \} c2 \{ R \} \quad \text{(hoare_seq)} \\
\text{\{ P \} c1; c2 \{ R \} & \\
\text{\{ P ∧ b \} c1 \{ Q \} & \text{\{ P ∧ ∼b \} c2 \{ Q \} \quad \text{(hoare_if)} \\
\text{\{ P ∧ b \} c \{ P \} & \text{\{ P \} WHILE b DO c END \{ P ∧ ∼b \} \quad \text{(hoare_while)} \\
\text{\{ P' \} c \{ Q' \} & \quad \text{\{ P \} c \{ Q \} \quad \text{(hoare_consequence)} \\
\end{align*}
\]

Decorated programs

A decorated program consists of the program text interleaved with assertions. To check that a decorated program represents a valid proof, we check that each individual command is locally consistent with its accompanying assertions in the following sense:

- SKIP is locally consistent if its precondition and postcondition are the same:
• The sequential composition of commands \( c_1 \) and \( c_2 \) is locally consistent (with respect to assertions \( P \) and \( R \)) if \( c_1 \) is locally consistent (with respect to \( P \) and \( Q \)) and \( c_2 \) is locally consistent (with respect to \( Q \) and \( R \)):

\[
\begin{align*}
\{ \ P \ \} \\
c_1; \\
\{ \ Q \ \} \\
c_2 \\
\{ \ R \ \}
\end{align*}
\]

• An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

\[
\begin{align*}
\{ \ P \ \} \\
x ::= a \\
\{ \ P \ \}
\end{align*}
\]

• A conditional is locally consistent (with respect to assertions \( P \) and \( Q \)) if the assertions at the top of its “then” and “else” branches are exactly \( P \land b \) and \( P \land \neg b \) and if its “then” branch is locally consistent (with respect to \( P \land b \) and \( Q \)) and its “else” branch is locally consistent (with respect to \( P \land \neg b \) and \( Q \)):

\[
\begin{align*}
\{ \ P \ \} \\
\text{IF} \ b \ \text{THEN} \\
\{ \ P \land b \ \} \\
c_1 \\
\{ \ Q \ \} \\
\text{ELSE} \\
\{ \ P \land \neg b \ \} \\
c_2 \\
\{ \ Q \ \} \\
\text{FI} \\
\{ \ Q \ \}
\end{align*}
\]

• A while loop is locally consistent if its postcondition is \( P \land \neg b \) (where \( P \) is its precondition) and if the pre- and postconditions of its body are exactly \( P \land b \) and \( P \) :

\[
\begin{align*}
\{ \ P \ \} \\
\text{WHILE} \ b \ \text{DO} \\
\{ \ P \land b \ \} \\
c_1 \\
\{ \ P \ \} \\
\text{END} \\
\{ \ P \land \neg b \ \}
\end{align*}
\]
- A pair of assertions separated by $\Rightarrow$ is locally consistent if the first implies the second (in all states):

$\{\{ P \} \} \Rightarrow \{\{ Q \} \}$
STLC with booleans

Syntax

\[
T ::= \text{Bool} \\
| T \rightarrow T \\
| t \ t \\
| \ x : T . \ t \\
| \text{true} \\
| \text{false} \\
| \text{if } t \text{ then } t \text{ else } t
\]

Small-step operational semantics

\[
\text{value } v_2 \\
\text{------------------------ (ST_AppAbs)} \\
(\ x : T . t_1 2 ) \ v_2 \Rightarrow [ x := v_2 ] t_1 2
\]

\[
t_1 \Rightarrow t_1' \\
\text{---------------- (ST_App1)} \\
t_1 \ t_2 \Rightarrow t_1' \ t_2
\]

\[
\text{value } v_1 \\
v_1 \ t_2 \Rightarrow v_1 \ t_2' \\
\text{---------------- (ST_App2)}
\]

\[
\text{------------------------ (ST_IfTrue)} \\
(\text{if true then } t_1 \text{ else } t_2) \Rightarrow t_1
\]

\[
\text{------------------------ (ST_IfFalse)} \\
(\text{if false then } t_1 \text{ else } t_2) \Rightarrow t_2
\]

\[
t_1 \Rightarrow t_1' \\
\text{------------------------ (ST_If)} \\
(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) \Rightarrow (\text{if } t_1' \text{ then } t_2 \text{ else } t_3)
\]
Typing

\[ \Gamma \vdash x = T \]
\[ \begin{array}{c}
\hline
\Gamma, x : \text{T11} \vdash \text{t12} \in \text{T12} \\
\hline
\end{array} \]
\[ \begin{array}{c}
\hline
\Gamma \vdash \lambda x : \text{T11}. \text{t12} \in \text{T11} \rightarrow \text{T12} \\
\hline
\end{array} \]
\[ \begin{array}{c}
\hline
\Gamma \vdash \text{t1} \in \text{T11} \rightarrow \text{T12} \\
\Gamma \vdash \text{t2} \in \text{T11} \\
\hline
\end{array} \]
\[ \begin{array}{c}
\hline
\Gamma \vdash \text{t1} \text{ t2} \in \text{T12} \\
\hline
\end{array} \]
\[ \begin{array}{c}
\hline
\Gamma \vdash \text{true} \in \text{Bool} \\
\hline
\end{array} \]
\[ \begin{array}{c}
\hline
\Gamma \vdash \text{false} \in \text{Bool} \\
\hline
\end{array} \]
\[ \begin{array}{c}
\hline
\Gamma \vdash \text{t1} \in \text{Bool} \\
\Gamma \vdash \text{t2} \in \text{T} \\
\Gamma \vdash \text{t3} \in \text{T} \\
\hline
\end{array} \]
\[ \begin{array}{c}
\hline
\Gamma \vdash \text{if t1 then t2 else t3} \in \text{T} \\
\hline
\end{array} \]

Properties of STLC

Theorem preservation:
\[ \text{forall } t, t', T, \]
\[ \text{empty } \vdash t \in T \rightarrow \\
\text{t } \Rightarrow \text{t'} \rightarrow \\
\text{empty } \vdash t' \in T. \]

Theorem progress:
\[ \text{forall } t, T, \]
\[ \text{empty } \vdash t \in T \rightarrow \\
\text{value t \lor exists t', t } \Rightarrow \text{t'}. \]
STLC with booleans, records and subtyping

Syntax

\[
T ::= \ldots \quad t ::= \ldots \quad v ::= \ldots \\
\mid \{i_1: T_1, \ldots, i_n: T_n\} \quad \mid \{i_1= t_1, \ldots, i_n= t_n\} \quad \mid \{i_1= v_1, \ldots, i_n= v_n\} \\
\mid \text{Top} \quad \mid t.i
\]

Small-step operational semantics

\[
\begin{align*}
& \quad ti \Rightarrow ti' \\
& \quad \vdash \{i_1=v_1, \ldots, i_m=v_m, i_n=ti, \ldots\} \Rightarrow \{i_1=v_1, \ldots, i_m=v_m, i_n=ti', \ldots\} \\
& \quad t_1 \Rightarrow t_1' \\
& \quad t_1.i \Rightarrow t_1'.i \\
& \quad \vdash \{\ldots, i=v_i, \ldots\}.i \Rightarrow v_i
\end{align*}
\]

Typing

\[
\begin{align*}
& \quad \Gamma \vdash t_1 \in T_1 \ldots \quad \Gamma \vdash t_n \in T_n \\
& \quad \vdash \{i_1=t_1, \ldots, i_n=t_n\} \in \{i_1:T_1, \ldots, i_n:T_n\} \\
& \quad \Gamma \vdash t \in \{\ldots, i:T, \ldots\} \\
& \quad \vdash t.i \in T \\
& \quad \vdash t \in S \quad S <: T \\
& \quad \vdash t \in T
\end{align*}
\]
Subtyping

\[
S <: U \quad U <: T \\
\quad \quad \quad \quad \quad (S_{\text{Trans}}) \\
\quad \quad S <: T \\
\quad \quad \quad \quad \quad (S_{\text{Refl}}) \\
\quad \quad T <: T \\
\quad \quad \quad \quad \quad (S_{\text{Top}}) \\
\quad \quad S <: \text{Top} \\
\quad \quad T_1 <: S_1 \quad S_2 <: T_2 \\
\quad \quad \quad \quad \quad (S_{\text{Arrow}}) \\
\quad \quad S_1 \rightarrow S_2 <: T_1 \rightarrow T_2 \\
\quad \quad \quad \quad \quad (S_{\text{Arrow}}) \\
\quad \quad n > m \\
\quad \quad \quad \quad \quad (S_{\text{RcdWidth}}) \\
\quad \quad \{i_1:T_1...i_n:T_n\} <: \{i_1:T_1...i_m:T_m\} \\
\quad \quad S_1 <: T_1 \ldots S_n <: T_n \\
\quad \quad \quad \quad \quad (S_{\text{RcdDepth}}) \\
\quad \quad \{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\} \\
\quad \quad \quad \quad \quad (S_{\text{RcdPerm}}) \\
\quad \quad \quad \{i_1:S_1...i_n:S_n\} \text{ is a permutation of } \{i_1:T_1...i_n:T_n\} \\
\quad \quad \quad \quad \quad (S_{\text{RcdPerm}}) \\
\quad \quad \{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\} \\
\quad \quad \quad \quad \quad (S_{\text{RcdPerm}})
\]