CIS 500 — Software Foundations

Final Exam

(Advanced version)

December 18, 2014

Name: 

Pennkey (e.g. sweirich): 

Scores:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>16</td>
</tr>
<tr>
<td>Total:</td>
<td>110</td>
</tr>
</tbody>
</table>
1. **Properties of Imp Relations**

The propositions below concern basic properties of the Imp language. For each proposition, indicate whether it is true or false by circling either True or False. For reference, the definition of Imp, its evaluation semantics, and program equivalence ($cequiv$) starts on page 13.

(a) The evaluation relation for Imp is deterministic.
   
   True  False

(b) The $cequiv$ relation is symmetric.

   True  False

(c) The command `WHILE BFalse DO SKIP` is not equivalent to any other command.

   True  False

(d) There is an Imp command $c$ that terminates for some input states and diverges for others.

   True  False

(e) If $cequiv c_1 c_2$ then $cequiv (SKIP ;; c_1) (SKIP ;; c_2)$.

   True  False

(f) For all arithmetic expressions $a_1$ and $a_2$, we can show

   $cequiv (X ::= a_1 ;; Y ::= a_2) (Y ::= a_2 ;; X ::= a_1)$

   True  False

(g) If $SKIP / st || st'$ then we know that $st = st'$.

   True  False
2. Hoare Logic

The following Imp program (slowly) computes $X + Y$, placing the answer into $Z$.

\[
\begin{align*}
Z & ::= Y ; ; \\
\text{WHILE } X <> 0 \text{ DO} & \\
Z & ::= Z + 1 ; ; \\
X & ::= X - 1 \\
\text{END}
\end{align*}
\]

Below, add appropriate annotations in the provided spaces. You will need to give the outermost pre- and post-conditions; these assertions should show that the program works as described above. Use informal notations for mathematical formulae and assertions, but be completely precise in the way you apply the Hoare rules — i.e., write out assertions in \emph{exactly} the form given by the rules (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with $\Rightarrow$.

Mark the implication step(s) in your decoration (by circling the $\Rightarrow$) that rely on the following fact. You may use other arithmetic facts silently.

- $\forall a \ b \ c, \ b <> 0 \Rightarrow (a - b) + 1 = a - (b - 1)$

The Hoare rules and the rules for well-formed decorated programs are provided on pages 15 and 16.
3. Coq programming - Small-step semantics

This problem refers to the Coq version of the small-step relation (\texttt{step}) for the Simply-typed Lambda Calculus with booleans, shown on page 18.

Because the \texttt{step} relation is deterministic, we can write a Coq function, called \texttt{next_step}, that computes what each term steps to (if any). The next page shows (part of) the definition of this function; you will need to complete the definition. Your implementation should satisfy the following correctness lemmas that state that it exactly corresponds to the \texttt{step} relation.

\textbf{Lemma next_step_correct1} : \forall t t',
\texttt{step t t'} \iff \texttt{next_step t} = \text{Some } t'.

\textbf{Lemma next_step_correct2} : \forall t,
\texttt{normal_form step t} \iff \texttt{next_step t} = \text{None}.

(a) Fill in the blanks for the following \textit{examples} that demonstrate the evaluation of \texttt{next_step}. Your answers should be consistent with the correctness lemmas shown above.

Several of these examples make use of the following definition:

\texttt{(* Identity function for booleans *)
Definition idB := tabs x TBool (tvar x).}

The first one has been done for you.

\textbf{Example ex0} : \texttt{next_step (tapp idB ttrue)} =

\texttt{____________Some ttrue________________________}.

\textbf{Example ex1} : \texttt{next_step ttrue} =

\texttt{-----------------------------------------------------------------------}.

\textbf{Example ex2} : \texttt{next_step (tapp ttrue tfalse)} =

\texttt{-----------------------------------------------------------------------}.

\textbf{Example ex3} : \texttt{next_step (tapp idB (tif ttrue tfalse ttrue))} =

\texttt{-----------------------------------------------------------------------}.

\textbf{Example ex4} : \texttt{next_step (tif ttrue (tapp idB ttrue) tfalse)} =

\texttt{-----------------------------------------------------------------------}.
(b) Now complete the implementation of the `next_step` function. The first few cases of this implementation have been given for you.

Your code may use the following helper function in your answer.

(* Determine whether the given term is a value *)

Fixpoint is_value (t : tm) : bool :=
    match t with
    | tabs x T u => true
    | ttrue => true
    | tfalse => true
    | _ => false
end.

(* Calculate the next (small-)step for this term, if one exists *)

Fixpoint next_step (t : tm) : option tm :=
    match t with
    | tif ttrue t2 t3 => Some t2
    | tif tfalse t2 t3 => Some t3
    | tif t1 t2 t3 => match next_step t1 with
        | Some t1' => Some (tif t1' t2 t3)
        | None => None
    end

(* Fill in remaining cases here, and on the next page if necessary *)

|
(Extra space for the implementation of next_step, if necessary.)
4. Inductive Definitions and Scoping

Consider the following Coq definitions for a simple language of expressions with constants, variables, and options.

Definition id := nat.
Inductive tm : Type :=
| tnum : nat -> tm (* Constants 0, 1, 2, ... *)
| tvar : id -> tm (* Variables X Y Z ... *)
| tsome : tm -> tm (* Some t1 *)
| tnone : tm (* None *)
| tmatch : tm -> tm -> id -> tm -> tm. (* match t1 with
  | None => t2
  | Some x => t3 *)

For example, we might encode the Coq expression

match x with
| None => 0
| Some y => y
end

as tmatch (tvar X) (tnum 0) Y (tvar Y).

The tmatch construct follows the usual variable scoping rules. That is, in the expression tmatch t1 t2 X t3 the variable X is bound in t3.

Note that a variable X appears free in a term t if there is an occurrence of X that is not bound by a corresponding tmatch. Complete the following Coq definition of afi as an inductively defined relation such that afi X t is provable if and only if X appears free in t. You may use the next page if you need more space.

Inductive afi : id -> tm -> Prop :=
(Space for the definition of afi, if necessary).
5. **Subtyping**

The subtyping rules for STLC extended with pairs and records are given on page 22 for your reference. The subtyping relations among a collection of types can be visualized compactly in picture form: we draw a graph so that \( S <: T \) iff we can get from \( S \) to \( T \) by following arrows in the graph (either directly or indirectly). For example, a picture for the types \( \text{Top}*\text{Top} \), \( \text{A}*\text{Top} \), \( \text{Top}*(\text{Top}*\text{Top}) \), and \( \text{Top}*(\text{A}*\text{A}) \) would look like this (it happens to form a tree, but that is not necessary in general):

\[
\begin{array}{c}
\text{Top}*\text{Top} \\
\text{A}*\text{Top} & \text{Top}*(\text{Top}*\text{Top}) \\
& \text{Top}*(\text{A}*\text{A})
\end{array}
\]

Suppose we have defined types \( \text{A} \) and \( \text{B} \) so that \( \text{A} <: \text{B} \). Draw a picture for the following seven types.

\[
\begin{align*}
\{\} \\
\{ \text{m} : \text{A} \} \\
\{ \text{m} : \text{B} \} \\
\{ \text{k} : \text{B} \} \\
\{ \text{m} : \text{Top} \} \\
\{ \text{m} : \text{A}, \text{k} : \text{B} \} \\
\text{Top} \\
\{ \text{m} : \text{A} \} \rightarrow \text{Top}
\end{align*}
\]
6. Informal Proofs - Subtyping

Give a detailed proof of the following inversion lemma for typing abstractions in STLC with subtyping (see the rules on page 21, you should the language with booleans and functions only, not the extension with product and record types).

**Lemma** (Typing Inversion for Abstractions): If $\Gamma \vdash \lambda x : S_1 . t_2 \in T$, then there is a type $S_2$ such that $\Gamma, x : S_1 \vdash t_2 \in S_2$ and $S_1 \rightarrow S_2 \sqsubseteq T$. 
7. Informal Proofs — Progress with null

In this problem we will extend STLC with null. Your job will be to state and prove a progress lemma for this extension. Although we will work with informal notation for this problem, the base language is the one specified by the Coq formalization, starting on page .

Informally, the only change we will make to the syntax of the language is to add null,

\[ t ::= x \mid \lambda x:T . t \mid t_1 t_2 \mid \text{true} \mid \text{false} \mid \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \mid \text{null} \]

which is a new form of value.

\[ \text{----------- (v_null)} \]
\[ \text{value null} \]

We also add one new typing rule, shown below, that allows the null value to have any type.

\[ \text{----------- (T_Null)} \]
\[ \Gamma \vdash \text{null} \in T \]

(The call-by-value operational semantics for this language is unchanged.)

The standard progress lemma doesn’t hold for STLC with this extension. The problem is that a term could get stuck when trying to use a null value. For example the term \( \text{if null then t1 else t2} \) is well typed, but isn’t a value and doesn’t step. We say that stuck terms like this one throw null pointer exceptions. In this problem, we won’t model those exceptions directly. Instead, we define a relation, called npe, that describes where they should occur. We can use this relation to modify the Progress lemma (as shown on the next page) so that it is true.

The npe relation (shown on the next page) characterizes those terms that should throw null pointer exceptions. This relation includes terms that try to use null as another sort of value. It does not include any other stuck terms.

Your task for this problem is to complete the proof of this revised progress lemma.

*Hint: This question is asking whether you understand the proof of the progress lemma for STLC. You will receive half credit for recreating the appropriate parts of the standard proof here, without the extension to null.*
The npe relation

\[
\begin{align*}
\text{NPE}_\text{If} & \quad \text{npe} \ (\text{if null then } t_1 \text{ else } t_2) \\
\text{npe} & \quad t_1 \\
\text{NPE}_\text{If1} & \quad \text{npe} \ (\text{if } t_1 \text{ then } t_2 \text{ else } t_3) \\
\text{NPE}_\text{App} & \quad \text{npe} \ (\text{null } t_1) \\
\text{npe} & \quad t_1 \\
\text{NPE}_\text{App1} & \quad \text{npe} \ (t_1 \ t_2) \\
\text{NPE}_\text{App2} & \quad \text{npe} \ (t_1 \ t_2)
\end{align*}
\]

**Lemma** (Progress): For all \( t \in T \), if \( \emptyset \vdash t \in T \) then either \( t \) is a value, throws a null pointer exception, or steps. (i.e. either \text{value} \( t \) or \text{npe} \( t \) holds or there exists some \( t' \) such that \( t \Rightarrow t' \)).

**Proof**: by induction on the derivation of \( \emptyset \vdash t \in T \).

- The last rule of the derivation cannot be \text{T_VAR}, since a variable is never well typed in an empty context.

- The \text{T_Null}, \text{T_True}, \text{T_False}, and \text{T_Abs} cases are trivial, because in each of these cases we know immediately that \( t \) is a value.

- If the last rule of the derivation was \text{T_If}, then ... (This case is omitted for brevity.)

- If the last rule of the derivation was \text{T_App}, then ... (Complete this case of the proof on the next page.)
(T_App case of the revised progress proof).
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 -> com.

Notation "'SKIP'" :=
CSkip.
Notation "l '::=' a" :=
(CAss l 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

\[
\text{Inductive ceval : com \to state \to state \to Prop := }\\
\begin{align*}
| \text{E.Skip} & : \forall \text{st}, \\
& \quad \text{SKIP} / \text{st} \parallel \text{st} \\
| \text{E.Ass} & : \forall \text{st} \text{ a1 n X}, \\
& \quad \text{aeval st a1 = n} \rightarrow \\
& \quad (X := a1) / \text{st} \parallel (\text{update st X n}) \\
| \text{E.Seq} & : \forall \text{c1 c2 st st'} \text{ st''}, \\
& \quad \text{c1 / st} \parallel \text{st'} \rightarrow \\
& \quad \text{c2 / st'} \parallel \text{st''} \rightarrow \\
& \quad (\text{c1 ; c2}) / \text{st} \parallel \text{st''} \\
| \text{E.IfTrue} & : \forall \text{st st'} \text{ b1 c1 c2}, \\
& \quad \text{beval st b1 = true} \rightarrow \\
& \quad \text{c1 / st} \parallel \text{st'} \rightarrow \\
& \quad (\text{IFB b1 THEN c1 ELSE c2 FI}) / \text{st} \parallel \text{st'} \\
| \text{E.IfFalse} & : \forall \text{st st'} \text{ b1 c1 c2}, \\
& \quad \text{beval st b1 = false} \rightarrow \\
& \quad \text{c2 / st} \parallel \text{st'} \rightarrow \\
& \quad (\text{IFB b1 THEN c1 ELSE c2 FI}) / \text{st} \parallel \text{st'} \\
| \text{E.WhileEnd} & : \forall \text{b1 st c1}, \\
& \quad \text{beval st b1 = false} \rightarrow \\
& \quad (\text{WHILE b1 DO c1 END}) / \text{st} \parallel \text{st} \\
| \text{E.WhileLoop} & : \forall \text{st st'} \text{ st'' b1 c1}, \\
& \quad \text{beval st b1 = true} \rightarrow \\
& \quad \text{c1 / st} \parallel \text{st'} \rightarrow \\
& \quad (\text{WHILE b1 DO c1 END}) / \text{st'} \parallel \text{st''} \rightarrow \\
& \quad (\text{WHILE b1 DO c1 END}) / \text{st} \parallel \text{st''} \\
\end{align*}
\]

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

Program equivalence

\[
\text{Definition bequiv (b1 b2 : bexp) : Prop := } \\
\quad \forall (\text{st:state}), \text{beval st b1 = beval st b2}. \\
\]

\[
\text{Definition cequiv (c1 c2 : com) : Prop := } \\
\quad \forall (\text{st st'} : \text{state}), \\
\quad (\text{c1 / st} \parallel \text{st'}) \leftrightarrow (\text{c2 / st} \parallel \text{st'}). \\
\]

Hoare triples

\[
\text{Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop := } \\
\quad \forall \text{st st'}, c / \text{st} \parallel \text{st'} \rightarrow P \text{ st} \rightarrow Q \text{ st'}. \\
\]

\[
\text{Notation } "\{\{ P \}\} c \{\{ Q \}\}" := (\text{hoare_triple P c Q}).
\]
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).

(ASCII ->> is typeset as a hollow arrow in the rules below.)

Hoare logic rules

\[ \{ \text{assn_sub \ } X a Q \} X := a \{ Q \} \]  \quad \text{(hoare_assgn)}

\[ \{ P \} \text{SKIP} \{ P \} \]  \quad \text{(hoare_skip)}

\[ \{ P \} c1 \{ Q \} \]
\[ \{ Q \} c2 \{ R \} \]
\[ \{ P \} c1; c2 \{ R \} \]  \quad \text{(hoare_seq)}

\[ \{ P \land b \} c1 \{ Q \} \]
\[ \{ P \land \neg b \} c2 \{ Q \} \]
\[ \{ P \} \text{IFB \ } b \text{ THEN } c1 \text{ ELSE } c2 \text{ FI } \{ Q \} \]  \quad \text{(hoare_if)}

\[ \{ P \land b \} c \{ P \} \]
\[ \{ P \} \text{WHILE \ } b \text{ DO \ } c \text{ END } \{ P \land \neg b \} \]  \quad \text{(hoare_while)}

\[ \{ P' \} c \{ Q' \} \]
\[ P \rightarrow P' \]
\[ Q' \rightarrow Q \]
\[ \{ P \} c \{ Q \} \]  \quad \text{(hoare_consequence)}

\[ \{ P' \} c \{ Q \} \]
\[ P \rightarrow P' \]
\[ \{ P \} c \{ Q \} \]  \quad \text{(hoare_consequence_pre)}

\[ \{ P \} c \{ Q' \} \]
\[ Q' \rightarrow Q \]
\[ \{ P \} c \{ Q \} \]  \quad \text{(hoare_consequence_post)}
Decorated programs

(a) SKIP is locally consistent if its precondition and postcondition are the same:

\[
\begin{align*}
\{ \{ P \} \} \\
\text{SKIP} \\
\{ \{ P \} \}
\end{align*}
\]

(b) The sequential composition of \(c_1\) 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 \(R\)):

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

(c) An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

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

(d) 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 \(Q\)):

\[
\begin{align*}
\{ \{ P \} \} \\
\text{IFB } b \text{ THEN} \\
\{ \{ P \land b \} \} \\
c_1 \\
\{ \{ Q \} \} \\
\text{ELSE} \\
\{ \{ P \land \neg b \} \} \\
c_2 \\
\{ \{ Q \} \} \\
\text{FI} \\
\{ \{ Q \} \}
\end{align*}
\]
(e) A while loop with precondition \( P \) is locally consistent if its postcondition is \( P \land \neg b \) and if the pre- and postconditions of its body are exactly \( P \land b \) and \( P \):

\[
\begin{align*}
&\{\{P\}\} \\
&\text{WHILE } b \text{ DO} \\
&\quad \{\{P \land b\}\} \\
&\quad c1 \\
&\quad \{\{P\}\} \\
&\text{END} \\
&\{\{P \land \neg b\}\}
\end{align*}
\]

(f) A pair of assertions separated by \( \rightarrow\rightarrow \) is locally consistent if the first implies the second (in all states):

\[
\begin{align*}
&\{\{P\}\} \rightarrow\rightarrow \\
&\{\{P'\}\}
\end{align*}
\]
Coq formalization of STLC with booleans

Identifiers

Inductive id : Type :=
  Id : nat -> id.

Theorem eq_id_dec : forall id1 id2 : id, {id1 = id2} + {id1 <> id2}.

Definition x := (Id 0).
Definition y := (Id 1).
Definition z := (Id 2).

Types

Inductive ty : Type :=
  | TBool : ty
  | TArrow : ty -> ty -> ty.

Terms

Inductive tm : Type :=
  | tvar : id -> tm
  | tapp : tm -> tm -> tm
  | tabs : id -> ty -> tm -> tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm -> tm -> tm -> tm.

Inductive value : tm -> Prop :=
  | v_abs : forall x T t, value (tabs x T t)
  | v_true :
    value ttrue
  | v_false :
    value tfalse.

Substitution

Reserved Notation "@[' x ':= s ']@ t" (at level 20).

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' =>
      if eq_id_dec x x' then s else t
  | tabs x' T t1 =>
      tabs x' T (if eq_id_dec x x' then t1 else ([x:=s] t1))
| tapp t1 t2 =>
  tapp ([x:=s] t1) ([x:=s] t2)
| ttrue =>
  ttrue
| tfalse =>
  tfalse
| tif t1 t2 t3 =>
  tif ([x:=s] t1) ([x:=s] t2) ([x:=s] t3)
end

where "'[ ' x ':=' s ' ] ' t" := (subst x s t).

Reduction

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm -> tm -> Prop :=
  | ST_AppAbs : forall x T t12 v2,
      value v2 ->
      (tapp (tabs x T t12) v2) ==> [x:=v2]t12
  | ST_App1 : forall t1 t1' t2,
      t1 ==> t1' ->
      tapp t1 t2 ==> tapp t1' t2
  | ST_App2 : forall v1 t2 t2',
      value v1 ->
      t2 ==> t2' ->
      tapp v1 t2 ==> tapp v1 t2'
  | ST_IfTrue : forall t1 t2,
      (tif ttrue t1 t2) ==> t1
  | ST_IfFalse : forall t1 t2,
      (tif tfalse t1 t2) ==> t2
  | ST_If : forall t1 t1' t2 t3,
      t1 ==> t1' ->
      (tif t1 t2 t3) ==> (tif t1' t2 t3)

where "t1 '==>' t2" := (step t1 t2).

Definition normal_form {X:Type} (R:relation X) (t:X) : Prop :=
  ~ exists t', R t t'.

Contexts

Definition partial_map (A:Type) := id -> option A.

Definition empty {A:Type} : partial_map A := (fun _ => None).
Definition extend \{(A:Type) \ (Gamma : partial_map A) \ (x:id) \ (T : A) \} :=
    fun x' => if eq_id_dec x x' then Some T else Gamma x'.

Typing Relation

Reserved Notation "Gamma ’|-’ t ’\in’ T" (at level 40).

Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T,
     Gamma x = Some T ->
     Gamma |- tvar x \in T
  | T_Abs : forall Gamma x T11 T12 t12,
     extend Gamma x T11 |- t12 \in T12 ->
     Gamma |- tabs x T11 t12 \in TArrow T11 T12
  | T_App : forall T11 T12 Gamma t1 t2,
     Gamma |- t1 \in TArrow T11 T12 ->
     Gamma |- t2 \in T11 ->
     Gamma |- tapp t1 t2 \in T12
  | T_True : forall Gamma,
     Gamma |- ttrue \in TBool
  | T_False : forall Gamma,
     Gamma |- tfalse \in TBool
  | T_If : forall t1 t2 t3 T Gamma,
     Gamma |- t1 \in TBool ->
     Gamma |- t2 \in T ->
     Gamma |- t3 \in T ->
     Gamma |- tif t1 t2 t3 \in T

where "Gamma ’|-’ t ’\in’ T" := (has_type Gamma t T).
STLC with subtyping

Typing relation

\[ \Gamma \vdash x \in T \]
\[ \Gamma, x : T_1 \vdash t_2 \in T_2 \]
\[ \Gamma \vdash \lambda x : T_1 . t_2 \in T_1 \to T_2 \]
\[ \Gamma \vdash t_1 \in T_1 \to T_2 \quad \Gamma \vdash t_2 \in T_1 \]
\[ \Gamma \vdash t_1 t_2 \in T_2 \]
\[ \Gamma \vdash \text{true} \in \text{Bool} \]
\[ \Gamma \vdash \text{false} \in \text{Bool} \]
\[ \Gamma \vdash t_1 \in \text{Bool} \quad \Gamma \vdash t_2 \in T \quad \Gamma \vdash t_3 \in T \]
\[ \Gamma \vdash \text{if}~t_1 \text{ then } t_2 \text{ else } t_3 \in T \]
\[ \Delta \vdash t \in S \quad S \ll T \]
\[ \Gamma \vdash t \in T \]

Subtyping relation

\[ S \ll U \quad U \ll T \]
\[ S \ll T \]
\[ T \ll T \]
\[ S \ll \text{Top} \]
\[ T_1 \ll S_1 \quad S_2 \ll T_2 \]
\[ S_1 \to S_2 \ll T_1 \to T_2 \]
STLC subtyping, extended with pairs and records

Subtyping relation

\[
S_1 <: T_1 \quad S_2 <: T_2
\]

-------------------- (S_Prod)

\[
S_1 \times S_2 <: T_1 \times T_2
\]

\[
n > m
\]

--------------------------------- (S_RcdWidth)

\[
\{i_1:T_1...i_n:T_n\} <: \{i_1:T_1...i_m:T_m\}
\]

\[
S_1 <: T_1 \ldots S_n <: T_n
\]

---------------------------------- (S_RcdDepth)

\[
\{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\}
\]

\[
\{i_1:S_1...i_n:S_n\} \text{ is a permutation of } \{i_1:T_1...i_n:T_n\}
\]

--------------------------------------------- (S_RcdPerm)

\[
\{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\}
\]