

(Standard and Advanced versions together)

Name (printed): _____

Username (login id): _____

My signature below certifies that I have complied with the University of Pennsylvania’s Code of Academic Integrity in completing this examination.

Signature: _____ Date: _____

Directions: This exam booklet contains both the standard and advanced track questions. Questions with no annotation are for *both* tracks. Other questions are marked “Standard Only” or “Advanced Only”. *Do not do the questions intended for the other track.*

Mark the box of the track you wish to follow.

<input type="checkbox"/> Standard	
1	/12
2	ADVANCED ONLY/–
3	/18
4	/16
5	/18
6	ADVANCED ONLY/–
7	/20
8	/16
Total	/100

<input type="checkbox"/> Advanced	
1	STANDARD ONLY/–
2	/8
3	/18
4	/16
5	STANDARD ONLY/–
6	/22
7	/20
8	/16
Total	/100

1. [Standard Only] Coq Programming: Types (12 points)

For each of the following Coq terms, write its type or write “ill-typed” if it is not well typed.

(a) `fun (n:nat) => n + 1`

(b) `fun (n:nat) => fun (P:prop) => P n`

(c) `fun (P:nat -> Prop) => forall (m:nat), P m`

The typing questions below use these definitions taken from `Maps.v`:

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

```
Definition total_map (A:Type) := id -> A.
```

```
Definition partial_map (A:Type) := total_map (option A).
```

```
Definition empty {A:Type} : partial_map A :=
  t_empty None.
```

```
Definition t_update {A:Type} (m : total_map A) (x : id) (v : A) :=
  fun x' => if beq_id x x' then v else m x'.
```

(d) `partial_map total_map`

(e) `partial_map (nat -> nat)`

(f) `t_update empty (Id 0) (Some Id)`

2. [Advanced Only] Coq Programming (8 points)

(a) What is the type of the following Coq term? Write “ill typed” if it is not well typed.

```
fun n => (exists x, x < n)
```

(b) What is the type of the following Coq term? Write “ill typed” if it is not well typed.

```
fun P => fun n => fun (x:P(n+1)) => (P n)
```

Consider the following (well-typed) Coq program:

```
Fixpoint foo (n:nat) :=
  match n with
  | 0 => fun (y:nat) => y = 0
  | S m => fun (y:nat) =>
      match y with
      | 0 => True
      | S l => foo m l
      end
  end.
```

(c) What is the type of foo?

(d) Which of the following lemmas can you prove to characterize foo? (Choose one.)

- forall n m, n = m <-> foo m n
- forall n m, n < m <-> foo m n
- forall n m, m < n <-> foo m n
- forall n m, n <= m <-> foo m n
- forall n m, m <= n <-> foo m n

3. Imp Semantics (18 points)

The Appendix contains the definitions of the syntax, large-step operational semantics, and *program equivalence* as defined by the `cequiv` relation for Imp programs. Multiple choice: mark *all* correct answers. There may be zero or more than one!

(a) (3 pts.) Consider the Imp program:

```
Y ::= 0;;
WHILE X > 0 DO
  Y ::= Y + 2;;
  X ::= X - 1;;
DONE
```

Which of the following programs are equivalent to it, according to `cequiv`?

- | | | |
|-----------------------------|---|---|
| i. Y ::= X * 2;;
X := 0; | ii. IF X = 2 THEN
Y ::= 4;;
ELSE
Y ::= 2 * X;;
FI | iii. WHILE Y <> (2 * X) DO
Y := Y + 1;;
DONE;;
X ::= 0 |
|-----------------------------|---|---|

(b) (3 pts.) Consider the Imp program:

```
IF X > Y THEN
  WHILE True DO SKIP DONE
ELSE
  X ::= X - Y;;
FI
```

Which of the following programs are equivalent to it, according to `cequiv`?

- | | | |
|------------|--|--|
| i. X ::= 0 | ii. IF X <= Y THEN
X ::= 0
ELSE
WHILE True DO SKIP DONE
FI | iii. WHILE (X - Y) > 0 DO
SKIP
DONE;;
X ::= 0 |
|------------|--|--|

(c) (4 pts.) Which of the following choices of commands `c` are such that `c;;c` is equivalent to just `c` (according to `cequiv`)?

- | | | | |
|---------|--|-----------------------------|--|
| i. SKIP | ii. WHILE True DO
X ::= X+1
DONE | iii. X ::= Y;;
Y ::= X;; | iv. IF X = 0 THEN
SKIP
ELSE
X ::= X - 1;;
FI |
|---------|--|-----------------------------|--|

Suppose we extend Imp with the command `CALL` that lets us call a Coq-level function from within an Imp program:

```
Inductive com : Type := ...
  | CCall : (state -> state) -> com.      (* <---- new *)
```

```
Notation "'CALL' f" := (CCall f) (at level 60).
```

```
Inductive ceval : com -> state -> state -> Prop := ...
  | E_Call : forall (st : state) (f:state -> state),
    (CALL f) / st \\ (f st)      (* <---- run f on st *)
```

Here is a simple example of how it can be used to implement a function that zeros-out the state, and a lemma that shows what the `reset` function does:

```
Definition reset (st:state) : state := empty_state.
```

```
Lemma call_example : forall st,
  (X ::= 1;; Y ::= 2;; Z ::= 3;; CALL reset) / st \\ empty_state.
```

(d) (4 pts.) Is it possible to create an Imp command `c`, *that does not use CALL*, but such that `c` is equivalent to the program `CALL reset`? Briefly justify your answer.

(e) (4 pts.) Give an example Imp command `c` that *cannot* be implemented as a Coq function `f : state -> state` (i.e. such that `c` and `CALL f` are equivalent).
`c =`

4. **Hoare Logic: Decorated Programs** (16 points)

Consider the following Imp program that computes $m \cdot n$ and places the answer in Z

Add appropriate annotations to the program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid. 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 . The Appendix contains a list of all the Hoare rules and the rules for decorated programs.

We have given you the outer-loop's invariant.

```

    {{ X = m }}  $\rightarrow$ 
    {{
    Z ::= 0;;
    {{
    WHILE X <> 0 DO
    {{
    {{
    W ::= n;;
    {{
    WHILE W <> 0 DO
    {{
    {{
    Z ::= Z + 1;;
    {{
    W ::= W - 1
    {{
    END;
    {{
    {{
    X ::= X - 1
    {{
    END
    {{ Z = (m-X) * n /\ X<=m /\ ~(X<>0)
    {{ Z = m * n }}
  
```

5. [Standard Only] Simply-typed Lambda Calculus (18 points)

The syntax, operational semantics, and typing rules for the simply-typed lambda calculus with booleans is given in the Appendix. Here we *do not* consider products or subtyping.

For each variant below, indicate which of the properties of the STLC remain true in the presence of this rule? For each one, write either "remains true" or else "becomes false." If a property becomes false, give a counterexample.

- (a) Consider a variant of the STLC in which we add a new term `loop` with the following reduction and typing rules:

$$\frac{}{\text{loop} \Rightarrow \text{loop}} \quad (\text{ST_Loop})$$

$$\frac{}{\Gamma \vdash \text{loop} \in T} \quad (\text{T_Loop})$$

- Determinism of \Rightarrow

- Progress

- Preservation

- (b) Suppose instead that we add the following typing rule:

$$\frac{\Gamma \vdash t_1 \in U \rightarrow T}{\Gamma \vdash t_1 t_2 \in T} \quad (\text{T_App}')$$

- Determinism of \Rightarrow

- Progress

- Preservation

(c) Instead, suppose that we add a new term `guess T` (where `T` is a type) with the following reduction rule:

$$\frac{}{\text{guess } (T \rightarrow U) \Rightarrow \lambda x:T. \text{ guess } U} \quad (\text{ST_GArr})$$

and the following typing rule:

$$\frac{}{\Gamma \vdash \text{guess } T \in T} \quad (\text{T_Guess})$$

- Determinism of \Rightarrow
- Progress
- Preservation

6. **[Advanced Only] Informal Proof for STLC** (22 points)

In this problem we consider only the simply typed lambda calculus with booleans (in particular there is *no subtyping* and you can *ignore product types*). Give an informal proof of the substitution lemma (stated further down).

Let us write $FV(t)$ for the set of *free variables* that occur in t . You will need to use the following lemma (which you may assume as given) in the proof:

Context invariance If $\Gamma \vdash t \in T$ and $(\forall x, x \in FV(t) \rightarrow \Gamma(x) = \Gamma'(x))$ then $\Gamma' \vdash t \in T$.

Substitution If $\Gamma, x : T \vdash t \in U$ and $\vdash v \in T$ then $\Gamma \vdash [x := v]t \in U$.

7. Simply-typed Lambda Calculus (20 points)

In this problem, we consider an alternate formulation of the small-step operational semantics for the simply-typed lambda calculus with booleans (without subtyping and no products until part (d)).

One annoying thing about the operational semantics is the number of “structural” rules (ST_App1, ST_App2, ST_If) that we have to deal with. (It’s even worse when we consider products, but we’re leaving them out for simplicity here.)

An alternate formulation of the operational semantics is to give a syntax of “evaluation contexts” E (of type `ectx`) and “primitive steps” s (which are just particular terms) like this:

```
(* Evaluation contexts  E : ectx *)          (* prim_step : tm -> Prop *)
E ::= []                                     s ::= (\x:T.t) v
    | E t                                     | if true then t1 else t2
    | v E                                     | if false then t1 else t2
    | if E then t1 else t2 (* ST_If *)
```

Here we use “informal” syntax rather than Coq constructors to make it simpler to write examples. We also use the convention the v stands for a term that is a value. The idea is that E describes a term with a single “hole” `[]` in it, into which we can place an arbitrary term. We define the function that fills the hole by pattern matching on the E like this:

```
Fixpoint fill (t:tm) (E:ectx) : tm :=
  match E with
  | [] => t
  | E t1 => ((fill t E) t1)
  | v E => (v (fill t E))
  | if E then t1 else t2 => if (fill t E) then t1 else t2
  end.
```

Each non-hole E corresponds to one of the structural rules, which lets us use one evaluation rule `E_Hole` for all of them. We also include one rule for each primitive step, like this:

```

          s ==> t                                (\x:T.t) v ==> [x:=v]t      (ST_AppAbs)
----- (ST_Hole)
fill s E ==> fill t E                          if true then t1 else t2 ==> t1      (ST_IfTrue)
                                               if false then t1 else t2 ==> t2      (ST_IfFalse)
```

These rules replace the old definition of the small-step semantics.

There are no questions on this page.

If we use these evaluation contexts to prove soundness, we need a couple different helper lemmas.

- (a) (3 pts.) The first lemma says that we can always decompose a well-typed term if it is not a value:

```
Lemma decompose : forall (t:tm) (T:ty) ,
  ⊢ t : T ->
  value t ∨
  exists (E:ectx), exists (s:tm), (prim_step s) /\ t = fill s E.
```

The proofs of which of the following would *directly* require this `decompose` lemma? (If A needs B and B needs the lemma, mark only B.)

- canonical forms
- preservation
- progress
- context invariance
- substitution

- (b) (4 pts.) We also need a kind of substitution lemma that relates to `fill`. A bad attempt at stating it might be something like this:

```
Lemma ectx_substitution: forall (E:ectx) (x:id) (T U:ty) (t:tm) Γ,
  Γ, x:T ⊢ (fill x E) ∈ U ->
  ⊢ t ∈ T ->
  Γ ⊢ (fill t E) ∈ U.
```

Unfortunately, the lemma above is not provable (indeed it is false!). Briefly explain why.

(c) (3 pts.) A better way to state the substitution principal is:

```
Lemma ectx_substitution: forall (E:ectx) (T U:ty) (s t:tm)  $\Gamma$ ,
   $\Gamma \vdash (\text{fill } s \ E) \in U \rightarrow$  (* Hyp1 *)
   $\vdash s \in T \rightarrow$  (* Hyp2 *)
   $\vdash t \in T \rightarrow$  (* Hyp3 *)
   $\Gamma \vdash (\text{fill } t \ E) \in U$ .
```

It would be easiest to prove this fact by induction on which of the following? (Choose one.)

- | | | | |
|----------------------------|-------------------------------|-------------------------------|-------------------------------|
| <input type="checkbox"/> E | <input type="checkbox"/> T | <input type="checkbox"/> U | <input type="checkbox"/> s |
| <input type="checkbox"/> t | <input type="checkbox"/> Hyp1 | <input type="checkbox"/> Hyp2 | <input type="checkbox"/> Hyp3 |

(d) (4 pts.) Following the development above, what would you add to the definition of **E** to support products? (The usual rules for products are given in the appendix.) (You may need to add more than one clause.)

```
E ::= ...
    |
```

(e) It is also possible to use evaluation contexts to implement other language features. Here we add a new expression **halt**, which halts the program.

```
      E <> []
----- (ST_Halt)
fill halt E ==> halt
```

i. (2 pts.) What does the following term step to in one step (\Rightarrow)?

```
if (\x:Bool.x) halt then false else true
```

- true
- false
- halt
- $(\lambda x:\text{Bool}.x) \text{halt}$
- `if halt then false else true`

ii. (4 pts.) What should the typing rule for **halt** be to ensure type safety?

8. **Subtyping** (16 points)

In this problem we consider the simply typed lambda calculus with booleans, pairs, and *subtyping*. The syntax, operational semantics, and typing rules are given in the Appendix.

Recall that this language already includes the type `Top`, which is a supertype of all other types, as indicated by this subtyping rule:

$$\frac{}{S <: \text{Top}} \quad (\text{S_Top})$$

In this problem we consider the implications of adding a new type `Bot` (for “bottom”), which is a subtype of all others:

$$\frac{}{\text{Bot} <: S} \quad (\text{S_Bot})$$

Just as when we added `Top`, we leave the operational semantics and the typing rules unchanged.

(a) (3 pts.) For each of the following lemmas, indicate whether it is provable with the addition of `Bot` as described above:

• Lemma `sub_inversion_Bot` : forall U, U <: TBot -> U = TBot.

Provable Not provable

• Lemma `sub_inversion_Bool` : forall U, U <: TBool -> U = TBool.

Provable Not provable

• Lemma `canonical_forms_of_Bool` : forall s,
empty ⊢ s ∈ TBool ->
value s ->
(s = ttrue ∨ s = tfalse).

Provable Not provable

(b) (3 pts.) Recall that a term is *closed* if it has no free variables. Are there any closed values of type `Bot`? That is, can you find a value `v` such that:

empty ⊢ v : Bot

If so, give an example. If not, briefly explain.

(c) (5 pts.) For each pair of types T and S given below, indicate whether $T <: S$, $S <: T$, or T and S are *incomparable* (that is, not related by $<:$).

- $T = (\text{Bot} * \text{Top})$ $S = (\text{Bot} * \text{Bool})$
 $T <: S$ $S <: T$ incomparable

- $T = \text{Bool} \rightarrow \text{Bool}$ $S = \text{Top} \rightarrow \text{Bot}$
 $T <: S$ $S <: T$ incomparable

- $T = \text{Bool} \rightarrow \text{Top}$ $S = \text{Bot} \rightarrow \text{Bot}$
 $T <: S$ $S <: T$ incomparable

- $T = (\text{Bot} \rightarrow \text{Top}) \rightarrow \text{Bool}$ $S = (\text{Top} \rightarrow \text{Bot}) \rightarrow \text{Top}$
 $T <: S$ $S <: T$ incomparable

- $T = \text{Bool} \rightarrow (\text{Top} * \text{Bot})$ $S = \text{Bot} \rightarrow (\text{Bot} * \text{Top})$
 $T <: S$ $S <: T$ incomparable

(d) (5 pts.) Consider the following program:

$\text{empty} \vdash (\lambda x:T. x x) : T \rightarrow \text{Bot}$

Which of the following types T allow the above program to be well-typed? That is, for which of the following choices of T does there exist a typing derivation with the conclusion above?

- $T = \text{Bool} \rightarrow \text{Bool}$
- $T = \text{Bot}$
- $T = \text{Top}$
- $T = \text{Top} \rightarrow \text{Bot}$
- $T = (\text{Bot} \rightarrow \text{Bool}) \rightarrow \text{Bot}$

For Reference

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 "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

```
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).
```


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

$$\frac{}{\{\text{assn_sub } X \ a \ Q\} X := a \ \{\!\! \{ Q \}\!\! \}} \quad (\text{hoare_asgn})$$

$$\frac{}{\{\!\! \{ P \}\!\! \} \text{ SKIP } \{\!\! \{ P \}\!\! \}} \quad (\text{hoare_skip})$$

$$\frac{\{\!\! \{ P \}\!\! \} c1 \ \{\!\! \{ Q \}\!\! \} \quad \{\!\! \{ Q \}\!\! \} c2 \ \{\!\! \{ R \}\!\! \}}{\{\!\! \{ P \}\!\! \} c1;; c2 \ \{\!\! \{ R \}\!\! \}} \quad (\text{hoare_seq})$$

$$\frac{\{\!\! \{ P \wedge b \}\!\! \} c1 \ \{\!\! \{ Q \}\!\! \} \quad \{\!\! \{ P \wedge \sim b \}\!\! \} c2 \ \{\!\! \{ Q \}\!\! \}}{\{\!\! \{ P \}\!\! \} \text{ IFB } b \ \text{ THEN } c1 \ \text{ ELSE } c2 \ \text{ FI } \ \{\!\! \{ Q \}\!\! \}} \quad (\text{hoare_if})$$

$$\frac{\{\!\! \{ P \wedge b \}\!\! \} c \ \{\!\! \{ P \}\!\! \}}{\{\!\! \{ P \}\!\! \} \text{ WHILE } b \ \text{ DO } c \ \text{ END } \ \{\!\! \{ P \wedge \sim b \}\!\! \}} \quad (\text{hoare_while})$$

$$\frac{\{\!\! \{ P' \}\!\! \} c \ \{\!\! \{ Q' \}\!\! \} \quad P \rightarrow P' \quad Q' \rightarrow Q}{\{\!\! \{ P \}\!\! \} c \ \{\!\! \{ Q \}\!\! \}} \quad (\text{hoare_consequence})$$

$$\frac{\{\!\! \{ P' \}\!\! \} c \ \{\!\! \{ Q \}\!\! \} \quad P \rightarrow P'}{\{\!\! \{ P \}\!\! \} c \ \{\!\! \{ Q \}\!\! \}} \quad (\text{hoare_consequence_pre})$$

$$\frac{\{\!\! \{ P \}\!\! \} c \ \{\!\! \{ Q' \}\!\! \} \quad Q' \rightarrow Q}{\{\!\! \{ P \}\!\! \} c \ \{\!\! \{ Q \}\!\! \}} \quad (\text{hoare_consequence_post})$$

Decorated programs

1. SKIP is locally consistent if its precondition and postcondition are the same:

```
  {{ P }}
  SKIP
  {{ P }}
```

2. The sequential composition of $c1$ and $c2$ is locally consistent (with respect to assertions P and R) if $c1$ is locally consistent (with respect to P and Q) and $c2$ is locally consistent (with respect to Q and R):

```
  {{ P }}
  c1;
  {{ Q }}
  c2
  {{ R }}
```

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

```
  {{ P [X |-> a] }}
  X ::= a
  {{ P }}
```

4. 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 $P \wedge b$ and $P \wedge \sim b$ and if its "then" branch is locally consistent (with respect to $P \wedge b$ and Q) and its "else" branch is locally consistent (with respect to $P \wedge \sim b$ and Q):

```
  {{ P }}
  IFB b THEN
    {{ P /\ b }}
    c1
    {{ Q }}
  ELSE
    {{ P /\ ~b }}
    c2
    {{ Q }}
  FI
  {{ Q }}
```

5. A while loop with precondition P is locally consistent if its postcondition is $P \wedge \sim b$ and if the pre- and postconditions of its body are exactly $P \wedge b$ and P :

```

  {{ P }}
  WHILE b DO
    {{ P /\ b }}
    c1
    {{ P }}
  END
  {{ P /\ ~b }}

```

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

```

  {{ P }} ->
  {{ P' }}

```

Relations

Definition `relation (X: Type) := X->X->Prop`.

```

Inductive multi {X:Type} (R: relation X) : relation X :=
| multi_refl  : forall (x : X), multi R x x
| multi_step  : forall (x y z : X),
  R x y ->
  multi R y z ->
  multi R x z.

```

Notation "`t '==>*' t'`" := `(multi step t t')` (at level 40).

STLC with booleans

Syntax

$T ::= \text{Bool}$	$t ::= x$	$v ::= \text{true}$
$\quad T \rightarrow T$	$\quad t \ t$	$\quad \text{false}$
	$\quad \lambda x:T. t$	$\quad \lambda x:T. t$
	$\quad \text{true}$	
	$\quad \text{false}$	
	$\quad \text{if } t \text{ then } t \text{ else } t$	

Small-step operational semantics

$\frac{\text{value } v2}{\text{-----}}(\lambda x:T.t12) \ v2 \ ==> \ [x:=v2]t12$	(ST_AppAbs)
$\frac{t1 \ ==> \ t1'}{\text{-----}}t1 \ t2 \ ==> \ t1' \ t2$	(ST_App1)
$\frac{\text{value } v1 \quad t2 \ ==> \ t2'}{\text{-----}}v1 \ t2 \ ==> \ v1 \ t2'$	(ST_App2)
$\text{-----}(\text{if true then } t1 \ \text{else } t2) \ ==> \ t1$	(ST_IfTrue)
$\text{-----}(\text{if false then } t1 \ \text{else } t2) \ ==> \ t2$	(ST_IfFalse)
$\frac{t1 \ ==> \ t1'}{\text{-----}}(\text{if } t1 \ \text{then } t2 \ \text{else } t3) \ ==> \ (\text{if } t1' \ \text{then } t2 \ \text{else } t3)$	(ST_If)

Typing

$$\frac{\Gamma \ x = T}{\Gamma \vdash x \in T} \quad (\text{T_Var})$$
$$\frac{\Gamma, x:T_{11} \vdash t_{12} \in T_{12}}{\Gamma \vdash \lambda x:T_{11}.t_{12} \in T_{11} \rightarrow T_{12}} \quad (\text{T_Abs})$$
$$\frac{\Gamma \vdash t_1 \in T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 \in T_{11}}{\Gamma \vdash t_1 \ t_2 \in T_{12}} \quad (\text{T_App})$$
$$\frac{}{\Gamma \vdash \text{true} \in \text{Bool}} \quad (\text{T_True})$$
$$\frac{}{\Gamma \vdash \text{false} \in \text{Bool}} \quad (\text{T_False})$$
$$\frac{\Gamma \vdash t_1 \in \text{Bool} \quad \Gamma \vdash t_2 \in T \quad \Gamma \vdash t_3 \in T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in T} \quad (\text{T_If})$$

Properties of STLC

Theorem preservation : forall t t' T,
empty $\vdash t \in T \rightarrow$
t \Rightarrow t' \rightarrow
empty $\vdash t' \in T$.

Theorem progress : forall t T,
empty $\vdash t \in T \rightarrow$
value t \vee exists t', t \Rightarrow t'.

STLC with products

Extend the STLC with product types, terms, projections, and pair values:

$$\begin{array}{lll}
 T ::= \dots & t ::= \dots & v ::= \dots \\
 | T * T & | (t, t) & | (v, v) \\
 & | t.fst & \\
 & | t.snd &
 \end{array}$$

Small-step operational semantics (added to STLC rules)

$$\begin{array}{l}
 \begin{array}{c}
 t1 ==> t1' \\
 \hline
 (t1, t2) ==> (t1', t2)
 \end{array}
 \quad (ST_Pair1) \\
 \\
 \begin{array}{c}
 t2 ==> t2' \\
 \hline
 (v1, t2) ==> (v1, t2')
 \end{array}
 \quad (ST_Pair2) \\
 \\
 \begin{array}{c}
 t1 ==> t1' \\
 \hline
 t1.fst ==> t1'.fst
 \end{array}
 \quad (ST_Fst1) \\
 \\
 \begin{array}{c}
 \hline
 (v1, v2).fst ==> v1
 \end{array}
 \quad (ST_FstPair) \\
 \\
 \begin{array}{c}
 t1 ==> t1' \\
 \hline
 t1.snd ==> t1'.snd
 \end{array}
 \quad (ST_Snd1) \\
 \\
 \begin{array}{c}
 \hline
 (v1, v2).snd ==> v2
 \end{array}
 \quad (ST_SndPair)
 \end{array}$$

Typing (added to STLC rules)

$$\begin{array}{l}
 \begin{array}{c}
 \Gamma \vdash t1 \in T1 \quad \Gamma \vdash t2 \in T2 \\
 \hline
 \Gamma \vdash (t1, t2) \in T1 * T2
 \end{array}
 \quad (T_Pair) \\
 \\
 \begin{array}{c}
 \Gamma \vdash t1 \in T11 * T12 \\
 \hline
 \Gamma \vdash t1.fst \in T11
 \end{array}
 \quad (T_Fst) \\
 \\
 \begin{array}{c}
 \Gamma \vdash t1 \in T11 * T12 \\
 \hline
 \Gamma \vdash t1.snd \in T12
 \end{array}
 \quad (T_Snd)
 \end{array}$$

Subtyping

Extend the language above with the type Top (terms and values remain unchanged):

$T ::= \dots$
| Top

Add these rules that characterize the subtyping relation:

$$\frac{S <: U \quad U <: T}{S <: T} \quad (\text{S_Trans})$$

$$\frac{}{T <: T} \quad (\text{S_Ref1})$$

$$\frac{}{S <: \text{Top}} \quad (\text{S_Top})$$

$$\frac{S1 <: T1 \quad S2 <: T2}{S1 * S2 <: T1 * T2} \quad (\text{S_Prod})$$

$$\frac{T1 <: S1 \quad S2 <: T2}{S1 \rightarrow S2 <: T1 \rightarrow T2} \quad (\text{S_Arrow})$$

Typing (added to STLC with products)

All of the ordinary typing rules, plus:

$$\frac{\Gamma \vdash t \in S \quad S <: T}{\Gamma \vdash t \in T} \quad (\text{T_Sub})$$