CIS 500: Software Foundations **Final Exam**

May 3, 2016

(Standard and Advanced versions together)

Solutions

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`
   Answer: `nat -> nat`

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

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

   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`
   Answer: Ill typed

   (e) `partial_map (nat -> nat)`
   Answer: `Type`

   (f) `t_update empty (Id 0) (Some Id)`
   Answer: `id -> option (nat -> 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)
```

Answer: `nat -> Prop`

(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)
```

Answer: `forall (P : nat -> Type) (n : nat), P (n + 1) -> Type`

Consider the following (well-typed) Coq program:

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

(c) What is the type of `foo`?
Answer: `nat -> nat -> Prop`

(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
```

Answer: Just `i.`

(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
```

Answer: `ii.` and `iii.`

(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
```

Answer: `i.` and `ii.` and `iii.`

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.

Answer: No, it is not possible. Such a program would have to assign to all identifiers in the state, but no finite Imp program can do that (even using a `WHILE` loop).

(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).

Answer: `c = WHILE True DO SKIP DONE` (or any looping program)

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

Consider the following Imp program that computes `m*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 `->>`. 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 } ->>
  { 0 = (m-X) * n /\ X<=m }
Z ::= 0;;
  { Z = (m-X) * n /\ X<=m }
WHILE X <> 0 DO
    { Z = (m-X) * n /\ X<=m /\ X<>0 } ->>
    { Z = (m-X) * n + (n-n) /\ X<=m }
  W ::= n;;
    { Z = (m-X) * n + (n-W) /\ X<=m /\ W<=n }
  WHILE W <> 0 DO
      { Z = (m-X) * n + (n-W) /\ X<=m /\ W<=n /\ W<>0 } ->>
    Z ::= Z + 1;;
      { Z = (m-X) * n + (n-(W-1)) /\ X<=m /\ W-1<=n }
    W ::= W - 1
      { Z = (m-X) * n + (n-W) /\ X<=m /\ W<=n }
  END;;
    { Z = (m-X) * n + (n-W) /\ X<=m /\ W<=n /\ ~(W<>0) } ->>
    { Z = (m-(X-1)) * n /\ X<=m }
  X ::= X - 1
    { Z = (m-X) * n /\ X<=m }
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:

```
------------- (ST_Loop)        ---------------- (T_Loop)
loop ==> loop                    Γ ⊢ loop ∈ T
```

- Determinism of `==>`
  Remains true. Each term has at most one rule that applies.
- Progress
  Remains true. Every well-typed term can still take a step, as can `loop`
- Preservation
  Remains true. `loop` can have any type.

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

```
Γ ⊢ t1 ∈ U -> T
--------------------  (T_App')
Γ ⊢ t1 t2 ∈ T
```

- Determinism of `==>`
  Remains true.
- Progress
  Remains truel.
- Preservation
  Becomes false: `(\x:(Bool->Bool).x true) true` can be given the type `Bool` but it steps to `true true` which is ill-typed. (The substitution lemma does not apply.)

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

```
--------------------------------     (ST_GArr)
guess (T -> U) ==> \x:T. guess U
```

and the following typing rule:

```
-------------------- (T_Guess)
  Γ ⊢ guess T ∈ T
```

- Determinism of `==>`
  Remains true.
- Progress
  Becomes false. The term `guess Bool` is stuck.
- Preservation
  Remains true.

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 $\Gamma \vdash [x := v]t \in U$.

Answer: See the solution in the Software Foundations text.

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 ::= []                    (* hole *)       s ::= (\x:T.t) v
    | E t                   (* ST_App1 *)        | if true then t1 else t2
    | v E                   (* ST_App2 *)        | 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.

Answer: The context `E` itself might contain `x` as a free variable and so it won't be well-typed when we remove `x` from the context.

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

```
Lemma ectx_substitution:  forall (E:ectx) (T U:ty) (s t:tm) Γ,
  Γ ⊢ (fill s E) ∈ U ->   (* Hyp1 *)
  ⊢ s ∈ T ->             (* Hyp2 *)
  ⊢ t ∈ T ->             (* Hyp3 *)
  Γ ⊢ (fill t E) ∈ U.
```

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

☒ E	☐ T	☐ U	☐ s
☐ t	☐ Hyp1	☐ Hyp2	☐ 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 ::= ...
    | (E, t)
    | (v, E)
    | E.fst
    | E.snd
```

(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 (`==>`)?

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

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

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

```
-----------------
Γ ⊢ halt ∈ T
```

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:

```
--------                                   (S_Top)
S <: Top
```

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

```
--------                                   (S_Bot)
Bot <: S
```

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.

Answer: No. By induction on the possible typing derivation: there are no base-case values of type Bot, and subsumption would require a `T` such that `T <: Bot`, but (again by induction) in that case `T = Bot`.
```

(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 = (Bot * Top)` `S = (Bot * Bool)`

  ☐ `T <: S` ☒ `S <: T` ☐ incomparable

- `T = Bool -> Bool` `S = Top -> Bot`

  ☐ `T <: S` ☒ `S <: T` ☐ incomparable

- `T = Bool -> Top` `S = Bot -> Bot`

  ☐ `T <: S` ☐ `S <: T` ☒ incomparable

- `T = (Bot -> Top) -> Bool` `S = (Top -> Bot) -> Top`

  ☒ `T <: S` ☐ `S <: T` ☐ incomparable

- `T = Bool -> (Top * Bot)` `S = Bot -> (Bot * Top)`

  ☐ `T <: S` ☐ `S <: T` ☒ incomparable

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

`empty ⊢ (\x:T. x x) : T -> 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 exists a typing derivation with the conclusion above?

☐ `T = Bool -> Bool`
☒ `T = Bot`
☐ `T = Top`
☒ `T = Top -> Bot`
☒ `T = (Bot -> Bool) -> 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

$$\frac{}{\{\!\{\texttt{assn\_sub X a } Q\}\!\}\ \texttt{X := a}\ \{\!\{Q\}\!\}}\ (\texttt{hoare\_asgn})$$

$$\frac{}{\{\!\{P\}\!\}\ \texttt{SKIP}\ \{\!\{P\}\!\}}\ (\texttt{hoare\_skip})$$

$$\frac{\{\!\{P\}\!\}\ \texttt{c1}\ \{\!\{Q\}\!\} \quad \{\!\{Q\}\!\}\ \texttt{c2}\ \{\!\{R\}\!\}}{\{\!\{P\}\!\}\ \texttt{c1;; c2}\ \{\!\{R\}\!\}}\ (\texttt{hoare\_seq})$$

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

$$\frac{\{\!\{P \wedge b\}\!\}\ \texttt{c}\ \{\!\{P\}\!\}}{\{\!\{P\}\!\}\ \texttt{WHILE b DO c END}\ \{\!\{P \wedge \sim b\}\!\}}\ (\texttt{hoare\_while})$$

$$\frac{\{\!\{P'\}\!\}\ \texttt{c}\ \{\!\{Q'\}\!\} \quad P \twoheadrightarrow P' \quad Q' \twoheadrightarrow Q}{\{\!\{P\}\!\}\ \texttt{c}\ \{\!\{Q\}\!\}}\ (\texttt{hoare\_consequence})$$

$$\frac{\{\!\{P'\}\!\}\ \texttt{c}\ \{\!\{Q\}\!\} \quad P \twoheadrightarrow P'}{\{\!\{P\}\!\}\ \texttt{c}\ \{\!\{Q\}\!\}}\ (\texttt{hoare\_consequence\_pre})$$

$$\frac{\{\!\{P\}\!\}\ \texttt{c}\ \{\!\{Q'\}\!\} \quad Q' \twoheadrightarrow Q}{\{\!\{P\}\!\}\ \texttt{c}\ \{\!\{Q\}\!\}}\ (\texttt{hoare\_consequence\_post})$$

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 /\ b` and `P /\ ~b` and if its "then" branch is locally consistent (with respect to `P /\ b` and `Q`) and its "else" branch is locally consistent (with respect to `P /\ ~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 /\ ~b` and if the pre- and postconditions of its body are exactly `P /\ b` and `P`:

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

6. A pair of assertions separated by `->>` 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 ::= Bool              t ::= x                   v ::= true
    | T -> T                | t t                     | false
                            | \x:T. t                 | \x:T. t
                            | true
                            | false
                            | if t then t else t
```

Small-step operational semantics

$$\frac{\text{value } v2}{(\lambda x{:}T.\, t12)\ v2 \Longrightarrow [x:=v2]t12}\ (\text{ST\_AppAbs})$$

$$\frac{t1 \Longrightarrow t1'}{t1\ t2 \Longrightarrow t1'\ t2}\ (\text{ST\_App1})$$

$$\frac{\text{value } v1 \quad t2 \Longrightarrow t2'}{v1\ t2 \Longrightarrow v1\ t2'}\ (\text{ST\_App2})$$

$$\frac{}{(\text{if true then } t1 \text{ else } t2) \Longrightarrow t1}\ (\text{ST\_IfTrue})$$

$$\frac{}{(\text{if false then } t1 \text{ else } t2) \Longrightarrow t2}\ (\text{ST\_IfFalse})$$

$$\frac{t1 \Longrightarrow t1'}{(\text{if } t1 \text{ then } t2 \text{ else } t3) \Longrightarrow (\text{if } t1' \text{ then } t2 \text{ else } t3)}\ (\text{ST\_If})$$

Typing

$$\frac{\Gamma\ \texttt{x} = \texttt{T}}{\Gamma \vdash \texttt{x} \in \texttt{T}} \quad \text{(T\_Var)}$$

$$\frac{\Gamma,\ \texttt{x:T11} \vdash \texttt{t12} \in \texttt{T12}}{\Gamma \vdash \texttt{\textbackslash x:T11.t12} \in \texttt{T11->T12}} \quad \text{(T\_Abs)}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{T11->T12} \qquad \Gamma \vdash \texttt{t2} \in \texttt{T11}}{\Gamma \vdash \texttt{t1 t2} \in \texttt{T12}} \quad \text{(T\_App)}$$

$$\frac{}{\Gamma \vdash \texttt{true} \in \texttt{Bool}} \quad \text{(T\_True)}$$

$$\frac{}{\Gamma \vdash \texttt{false} \in \texttt{Bool}} \quad \text{(T\_False)}$$

$$\frac{\Gamma \vdash \texttt{t1} \in \texttt{Bool} \qquad \Gamma \vdash \texttt{t2} \in \texttt{T} \qquad \Gamma \vdash \texttt{t3} \in \texttt{T}}{\Gamma \vdash \texttt{if t1 then t2 else t3} \in \texttt{T}} \quad \text{(T\_If)}$$

Properties of STLC

```
Theorem preservation : forall t t' T,
     empty ⊢ t ∈ T  ->
     t ==> t'  ->
     empty ⊢ t' ∈ T.

Theorem progress : forall t T,
     empty ⊢ t ∈ T ->
     value t \/ exists t', t ==> t'.
```

STLC with products

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

```
T ::= ...                t ::=  ...                  v ::= ...
    | T * T                   | (t,t)                     | (v, v)
                              | t.fst
                              | t.snd
```

Small-step operational semantics (added to STLC rules)

$$\frac{t1 \Longrightarrow t1'}{(t1,t2) \Longrightarrow (t1',t2)} \text{(ST\_Pair1)}$$

$$\frac{t2 \Longrightarrow t2'}{(v1,t2) \Longrightarrow (v1,t2')} \text{(ST\_Pair2)}$$

$$\frac{t1 \Longrightarrow t1'}{t1.fst \Longrightarrow t1'.fst} \text{(ST\_Fst1)}$$

$$\frac{}{(v1,v2).fst \Longrightarrow v1} \text{(ST\_FstPair)}$$

$$\frac{t1 \Longrightarrow t1'}{t1.snd \Longrightarrow t1'.snd} \text{(ST\_Snd1)}$$

$$\frac{}{(v1,v2).snd \Longrightarrow v2} \text{(ST\_SndPair)}$$

Typing (added to STLC rules)

$$\frac{\Gamma \vdash t1 \in T1 \qquad \Gamma \vdash t2 \in T2}{\Gamma \vdash (t1,t2) \in T1*T2} \text{(T\_Pair)}$$

$$\frac{\Gamma \vdash t1 \in T11*T12}{\Gamma \vdash t1.fst \in T11} \text{(T\_Fst)}$$

$$\frac{\Gamma \vdash t1 \in T11*T12}{\Gamma \vdash t1.snd \in T12} \text{(T\_Snd)}$$

Subtyping

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

```
T ::= ...
    | Top
```

Add these rules that characterize the subtyping relation:

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

$$\frac{}{T <: T} \text{(S\_Refl)}$$

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

$$\frac{S_1 <: T_1 \quad S_2 <: T_2}{S_1 * S_2 <: T_1 * T_2} \text{(S\_Prod)}$$

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \text{(S\_Arrow)}$$

Typing (added to STLC with products)

All of the ordinary typing rules, plus:

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