CIS 500: Software Foundations **Final Exam**

December 16, 2021

Solutions

1 [Standard Track Only] Miscellaneous (16 points)

1.1 The type `True` in Coq is inhabited by the single value `true`.

☐ True ☒ False

(`True` is inhabited by `I`; `true` has type `bool`.)

1.2 The type `bool->False` in Coq is uninhabited.

☒ True ☐ False

1.3 The term `(fun P => P \/ ~P) False` in Coq has type `Prop`.

☒ True ☐ False

1.4 Authors of custom Ltac scripts for Coq need to be careful that their scripts do not diverge, as this would create an inconsistency in Coq's logic.

☐ True ☒ False

(A diverging tactic script cannot cause Coq to believe that it has a proof of something false. Rather, the job of a tactic script is to *build* a proof object; if it diverges, we just don't get any proof at all.)

1.5 If two Imp commands `c1` and `c2` are equivalent (that is, `st =[c1]=> st'` iff `st =[c2]=> st'` for all `st` and `st'`), then they also validate the same Hoare triples (that is, `{{P}}c1{{Q}}` iff `{{P}}c2{{Q}}`, for all `P` and `Q`).

☒ True ☐ False

1.6 Conversely, if two Imp commands validate the same Hoare triples, then they are equivalent.

☒ True ☐ False

1.7 For every `b : bexp` and `c1, c2 : com`, either the command `if b then c1 else c2` is equivalent to `c1` or it is equivalent to `c2`.

☐ True ☒ False

(Counterexample: Let `b` be `X <= Y`, let `c1` be `Z := 0`, and let `c2` be `Z := 1`.)

1.8 The big-step evaluation of programs in STLC + Fix can naturally be expressed in Coq as either an `Inductive` relation or a `Fixpoint`.

☐ True ☒ False

(Since PCF programs may not terminate, defining evaluation using `Fixpoint` is awkward.)

$\boxed{2}$ **Inductive relations** (12 points)

Two lists are "equivalent modulo stuttering" if compressing sequences of repeated elements into a single element (e.g., compressing `[1;1;2;3;3;3]` into `[1;2;3]`) makes them identical.

For example:

```
equiv_mod_stuttering [1;2;2;2] [1;1;1;2;2].

equiv_mod_stuttering ([] : list nat) ([] : list nat).

~ (equiv_mod_stuttering [1] [2]).

~ (equiv_mod_stuttering [1] [1;2]).
```

(The `list nat` annotations in the second example are there to help type inference.) *Update: Another good example we noticed during the exam:*

And we should have included an example underscoring the fact that ordering is still important.

Complete the inductive definition of `equiv_modulo_stuttering`:

```
Inductive equiv_mod_stuttering {X : Type} : list X -> list X -> Prop :=
```

Answer:

```
| Done : equiv_mod_stuttering [] []
| SameHead : forall x l1 l2,
    equiv_mod_stuttering l1 l2 ->
    equiv_mod_stuttering (x::l1) (x::l2)
| StutterLeft : forall x l1 l2,
    equiv_mod_stuttering (x::l1) l2 ->
    equiv_mod_stuttering (x::x::l1) l2
| StutterRight : forall x l1 l2,
    equiv_mod_stuttering l1 (x::l2) ->
    equiv_mod_stuttering l1 (x::x::l2).
```

$\boxed{3}$ **Program equivalence in Imp** (14 points)

Recall that two Imp commands `c1` and `c2` are said to be *equivalent* when `st =[c1]=> st'` iff `st =[c2]=> st'`, for all `st` and `st'`.

Choose True or False for the following claims (and give counterexamples as appropriates).

$\boxed{3.1}$ If `c` always diverges (that is, there are no `st` and `st'` such that `st =[c]=> st'`), then `c` is equivalent to `c;c`.

☒ True ☐ False

If you chose False, give a counterexample (a command `c` that always diverges but such that `c` is not equivalent to `c;c`):

$\boxed{3.2}$ Conversely, if `c` is equivalent to `c;c`, then `c` always diverges.

☐ True ☒ False

If you chose False, give a counterexample:

```
c = skip
```

$\boxed{3.3}$ If then `c` is constant (i.e., it always leaves the state unchanged), then `c` is equivalent to `c;c`.

☒ True ☐ False

If you chose False, give a counterexample (a command `c` that is constant but such that `c` is not equivalent to `c;c`):

$\boxed{3.4}$ Conversely, if `c` is equivalent to `c;c`, then `c` is constant.

☐ True ☒ False

If you chose False, give a counterexample:

```
c = if X=0 then X:=1 else skip fi
```

$\boxed{3.5}$ If there is some state `st'` in the *range* of `c` such that `c` fails to terminate when started in state `st'` (that is, `st =[c]=> st'` for some starting state `st` but there is no `st''` such that `st' =[c]=> st''`), then `c` is *not* equivalent to `c;c`.

☒ True ☐ False

If you chose False, give a counterexample:

$\boxed{4}$ [Advanced Track Only] Program equivalence in Imp, continued (8 points)

State the conditions under which `c` is equivalent to `c;c`. That is, give *necessary and sufficient* conditions on `c` that guarantee `c` is equivalent to `c;c`.

Answer: Command `c` is equivalent to `c;c` iff `c` is constant on every state in its range — that is, if `st =[c]=> st'` implies `st' =[c]=> st'` for all `st` and `st'`.

Proof (not requested by the question and not required for full credit, but FYI):

- Suppose `st =[c]=> st'` implies `st' =[c]=> st'` for all `st` and `st'`. Then `c` and `c;c` terminate on the same set of starting states, and they yield the same final state whenever they terminate—that is, they are equivalent.
- Conversely, suppose `st =[c]=> st'` does *not* imply `st' =[c]=> st'`—that is, there are some `st` and `st'` such that `st =[c]=> st'` but not `st' =[c]=> st'` (either `c` diverges when started on `st'` or it terminates with some other state `st''`). Then clearly `c` and `c;c` are not equivalent: the former terminates in `st'` but the latter does not.

5 Stlc with iteration (14 points)

The Simply Typed Lambda-Calculus with fixpoints allows general recursion—that is, terms involving `fix` may diverge. If we want to avoid divergent terms while still expressing many computations involving numbers, we can introduce a bounded `iter` combinator.

Iter takes a function `f : T -> T`, a number `n : Nat` that controls how many times the function is executed, and an initial value for an "accumulator" `a : T`. Every step of the loop, it decrements `n` and calls `f` to update the accumulator, stopping after `n` becomes 0.

```
Inductive tm: Type :=
| tm_abs (x: string) (p: ty) (body: tm)
| tm_app (e1: tm) (e2: tm)
| tm_succ (e: tm)
| tm_const (n: nat)
| tm_var (x: string)
(* NEW *)
| tm_iter (f: tm) (n: tm) (a: tm).
```

Here is an example of using `iter` to define addition of two natural numbers.

```
Definition add_f(a b: tm) :=
  <{ iter (\acc: Nat, succ acc) a b }>.

Hint Unfold add_f: core.
Example add_ex1: add_f <{ 3 }> <{ 5 }> -->* <{ 8 }>.
```

5.1 First, let's practice using `iter`. Define an `apply_n` function that takes as argument a function `f : Nat -> Nat` and a starting value `n : Nat` and composes `f` with itself `n` times. For example, `apply_n f 4 1` should yield `f (f (f (f 1)))`, while `apply_n f 0 1` should yield `1`.

```
Definition apply_n (f : tm) (n : tm) :=
```

```
Definition apply_n (f n : tm) :=
  <{
     iter (\acc:Nat, \a:Nat, f (acc a)) n (\a:Nat, a)
  }>.

Example apply_n_ex1: tm_app (apply_n <{ \i: Nat, succ i }> <{ 2 }>) <{ 0 }>  -->* <{ 2 }>.
```

(N.b.: This part was a bit confusing, technically (it's defining an STLC function as if it were a Coq function). We decided during the exam to just skip it.)

$\boxed{5.2}$ Now that you've got the hang of it, let's extend the call-by-value operational semantics of STLC with appropriate rules for `iter`. Note that the evaluation order of the arguments to `iter` are from left to right, ie: `f` evaluates first, then `n` and finally `a`.

```
Inductive step : tm -> tm -> Prop :=
| ST_AppAbs : forall x T2 t1 v2,
    value v2 ->
    <{(\x:T2, t1) v2}> --> <{ [x:=v2]t1 }>
| ST_App1 : forall t1 t1' t2,
    t1 --> t1' ->
    <{t1 t2}> --> <{t1' t2}>
| ST_App2 : forall v1 t2 t2',
    value v1 ->
    t2 --> t2' ->
    <{v1 t2}> --> <{v1  t2'}>
| ST_Succ1: forall e e',
    e --> e' ->
    <{ succ e }> --> <{ succ e' }>
| ST_Succ2: forall (n: nat),
    <{ succ n }> --> <{ {S n} }>
(* FILL IN HERE *)

| ST_IterStep: forall f a (n: nat),
    value f ->
    value a ->
    <{ iter f {S n} a }> --> <{ f (iter f n a) }>
| ST_IterEnd: forall f a,
    value f ->
    value a ->
    <{ iter f 0 a }> --> <{ a }>
| ST_Iter1: forall f f' n a,
    f --> f' ->
    <{ iter f n a }> --> <{ iter f' n a }>
| ST_Iter2: forall f n n' a,
    value f ->
    n --> n' ->
    <{ iter f n a }> --> <{ iter f n' a }>
| ST_Iter3: forall f n a a',
    value f ->
    value n ->
    a --> a' ->
    <{ iter f n a }> --> <{ iter f n a' }>
```

$\boxed{5.3}$ Finally, give a typing rule for `iter`. Both examples above (`add_f` and `apply_n`) should be well-typed.

```
Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T1,
      Gamma x = Some T1 ->
      Gamma |- x \in T1
  | T_Abs : forall Gamma x T1 T2 t1,
      x |-> T2 ; Gamma |- t1 \in T1 ->
      Gamma |- \x:T2, t1 \in (T2 -> T1)
  | T_App : forall T1 T2 Gamma t1 t2,
      Gamma |- t1 \in (T2 -> T1) ->
      Gamma |- t2 \in T2 ->
      Gamma |- t1 t2 \in T1
  | T_Succ: forall Gamma n,
      Gamma |- succ n \in Nat
  | T_Const: forall Gamma (n: nat),
      Gamma |- n \in Nat
(* FILL IN HERE *)

  | T_Iter: forall T Gamma f n a,
      Gamma |- f \in (T -> T) ->
      Gamma |- n \in Nat ->
      Gamma |- a \in T ->
      Gamma |- iter f n a \in T
```

6 **Hoare logic** (12 points)

In this problem we'll consider several Hoare triples, `{{P}}c{{Q}}`. For each one, you are asked to choose either "Valid" or else the best description of its "degree of invalidity" from among the following:

- "Inv at least once": Invalid at least once—i.e., there exists a state satisfying `P` such that, when started from this state, the command `c` will terminate in a state *not* satisfying `Q`. **In this case, provide a pair of states, one that satisfies the triple and one that does not.**
- "Inv when terminating": Always invalid mod termination—i.e., when started from *any* state satisfying `P`, the command `c` will either diverge or terminate in a state not satisfying `Q`. **Provide a pair of states, one that diverges and one for which `Q` is not satisfied.**
- "Inv always": Always invalid—i.e., when started from any state satisfying `P`, the command `c` will *definitely terminate* in a state not satisfying `Q`.

"Best description" means the strongest description that applies—i.e., "Inv always" is better than "Inv when terminating," which is stronger than "Inv at least once".

6.1

```
{{ True }}
if X = 1 then
  Y := 0
else
  Y := 1
{{ X = Y }}
```

☐ Valid ☐ Inv at least once ☐ Inv when terminating ☒ Inv always

If necessary provide a pair of states to justify your answer:

6.2

```
{{ X=0 }}
  while X=0 do Y := Y+1
{{ False }}
```

☒ Valid ☐ Inv at least once ☐ Inv when terminating ☐ Inv always

If necessary provide a pair of states to justify your answer:

6.3

```
{{ True }}
  while X > 0 do Y := Y+1; X := X-1;
{{ X = Y }}
```

☐ Valid ☒ Inv at least once ☐ Inv when terminating ☐ Inv always

If necessary provide a pair of states to justify your answer:

`X = 1` and `Y = 0` (postcondition not satisfied)
`X = 42` (diverges)

$\boxed{6.4}$

```
{{ True }}
  while X > 10 do X := X + 1;
{{ False }}
```

☐ Valid ☐ Inv at least once ☒ Inv when terminating ☐ Inv always

If necessary provide a pair of states to justify your answer:
`X = 1` (postcondition not satisfied)
`X = 42` (diverges)

7 [Standard Track Only] Loop invariants (8 points)

For each pair of Hoare triple and proposed loop invariant `Inv`, your job is to decide whether `Inv` can be used to prove a Hoare triple of this form:

```
{{ P }} while b do c end {{ Q }}
```

Specifically, you should decide whether `Inv` satisfies each of the three specific constraints from the Hoare rule for `while`:

(1) Implied by precondition:	`P ->> Inv`
(2) Preserved by loop body (when loop guard true):	`{{ Inv /\ b }} c {{ Inv }}`
(3) Implies postcondition (when loop guard false):	`(Inv /\ ~ b) ->> Q`

We call them *"Implied by Pre," "Preserved,"* and *"Implies Post"* below, for brevity.

7.1

```
{{ X=m /\ Y=n }}
  while Y<>0 do
     X ::= X + 1;
     Y ::= Y - 1
  end
{{ X = m+n }}
```

Proposed `Inv`	**Implied by Pre**	**Preserved**	**Implies Post**
`X > 0`	☐	☒	☐
`X = m+n`	☐	☐	☒
`X = m+n-Y`	☒	☒	☒

7.2

```
{{ X = Y }}
  while true do
     X ::= X * Y
  end
{{ X = Y * 37 }}
```

Proposed `Inv`	**Implied by Pre**	**Preserved**	**Implies Post**
`X <> 0`	☐	☐	☒
`exists (m :  nat), X = Y + m`	☒	☒	☒
`True`	☒	☒	☒

$\boxed{8}$ **Big-step vs. Small-step** (6 points)

Briefly explain the difference between big-step and small-step styles of operational semantics. What are the advantages of each style?

Answer: The big-step style directly relates a term to the final result of its evaluation; the small-step style relates a term to a "slightly more reduced" term in which a single subphrase has taken a single step of computation. Big-step definitions tend to be shorter and easier to read; one major disadvantage is that they conflate terms that have no result because they diverge and terms that have no result because their evaluation encounters an undefined state. Small-step definitions are sometimes preferred because they are closer to implementations. Also, concurrent execution is much easier to describe in a small-step style.

9 Observational equivalence of STLC terms (16 points)

Consider the simply typed lambda-calculus (page 1 in the handout) with booleans.

Suppose `t` is a closed term. We say that a list of closed terms `[a1; ...; an]` *saturates* `t` if `|- t a1 a2 ... an \in Bool`. *(Update: Note that, although saturating argument lists look like Coq lists, their elements do NOT need to all have the same STLC type.)*

Suppose `s` and `t` are terms of the same type. We say that `s` and `t` are *observationally equivalent* if, for every list of terms `[a1; ..., an]` that saturates both `s` and `t`, we have `s a1 ... an -->* true` iff `t a1 ... an -->* true`.

For example, `\x:Bool, x` is observationally equivalent to `\x:Bool, (\y:Bool, y) x`, because they yield the same result when applied to either of the two possible saturating argument lists, `[true]` and `[false]`.

For each of the following pairs of terms, check "Equivalent" if they are observationally equivalent and "Inequivalent" if not. In the latter case, give a saturating list of arguments on which they yield different boolean results.

9.1 `\x:Bool, true` and `\x:Bool, false`

☐ Equivalent ☒ Inequivalent

If "Inequivalent," provide a saturating list of arguments on which the terms give different results:

`[true]`

9.2 `\x:Bool, x` and `\x:Bool, true`

☐ Equivalent ☒ Inequivalent

If "Inequivalent," provide a saturating list of arguments on which the terms give different results:

`[false]`

9.3 `\x:Bool, \y:Bool, x` and `\x:Bool, \y:Bool, y`

☐ Equivalent ☒ Inequivalent

If "Inequivalent," provide a saturating list of arguments on which the terms give different results:

`[false;true]`

$\boxed{9.4}$ `\x:Bool->Bool, x` and `\x:Bool->Bool, \y:Bool, x y`

☒ Equivalent ☐ Inequivalent

If "Inequivalent," provide a saturating list of arguments on which the terms give different results:

$\boxed{9.5}$ `\x:Bool->Bool, \y:Bool, x y` and `\x:Bool->Bool, \y:Bool, x true`

☐ Equivalent ☒ Inequivalent

If "Inequivalent," provide a saturating list of arguments on which the terms give different results:

`[(\z:Bool,z); false]`

$\boxed{9.6}$ `\x:Bool->Bool, \y:Bool, x y` and `\x:Bool->Bool, \y:Bool, x (x y)`

☐ Equivalent ☒ Inequivalent

If "Inequivalent," provide a saturating list of arguments on which the terms give different results:

`[(\z:Bool, if z then false else true); false]`

$\boxed{9.7}$ `true` and `false`

☐ Equivalent ☒ Inequivalent

If "Inequivalent," provide a saturating list of arguments on which the terms give different results:

`[]`

$\boxed{10}$ **[Advanced Track Only] Preservation for STLC with sums (informal proof)** (16 points)

The definition of the STLC extended with binary sum types, booleans, and `Unit` can be found on page 3 of the accompanying reference sheet.

Fill in the missing cases below of the proof that reduction preserves types (that is, the cases for `T_Inl` and `T_Case`). Use full, grammatical sentences, and make sure to state any induction hypotheses *explicitly*.

You may refer to the usual *substitution lemma* without proof. (It is repeated on page 2 of the handout, for reference.)

Theorem (Preservation): If `|- t \in T` and `t --> t'`, then `|- t' \in T`.

Proof: By induction on a derivation of `|- t \in T`.

- We can immediately rule out `T_Var`, `T_Abs`, `T_TRue`, `T_False`, and `T_Unit` as final rules in the derivation, since in each of these cases `t` cannot take a step.
- The cases for `T_App`, `T_If`, and `T_Inr` are omitted.
- If the final rule in the derivation of `|- t \in T` is `T_Case`, then

  `t = case t0 of inl x => t1 | inr x => t2`,

  with `|- t0 : T1 + T2` and `x:T1 |- t1 : T` and `x:T2 |- t2 : T`. The induction hypothesis states that, if `t0 --> t0'`, then `|- t0 : T1 + T2`.

  Inspecting the step relation, we see that there are three rules that could have been used to step from `t` to `t'` — namely, `ST_Case`, `ST_CaseInl`, and `ST_CaseInr`.

  If the step rule was `ST_Case`, then `t0 --> t0'` and `t' = case t0' of inl x => t1 | inr x => t2`. By `T_Case`, we have `|- t' \in T`, as required.

  If the step rule was `ST_CaseInl`, then `t0 = inl T2 v1` and `t' = [x:=v1]t1`. By the substitution lemma, `|- t' \in T`, as required.

  The argument for the `ST_CaseInr` step rule is similar.
- [*Write answer for the Inr rule...*]

11 **Subtyping** (14 points)

The setting for this problem is the simply typed lambda-calculus with booleans, products, and subtyping (see page 9 in the handout).

11.1 Suppose `t = (\x:Bool, (x,x))`

Check *all* the types `T` such that `|- t \in T` (or "Not typeable"). *Update: (You should select "Some other type(s)," even though you have already selected some options above it, if the term has more types than what are listed.)*

- ☒ `Bool -> (Top*Top)`
- ☒ `Bool -> (Bool*Bool)`
- ☐ `Top -> (Bool*Bool)` *(Corrected from an earlier answer key)*
- ☐ `Top -> Top`
- ☒ `Top`
- ☒ Some other type(s)
- ☐ Not typeable

11.2 Which is the *minimal* type `T` such that `|- t \in T` (or check "Not typeable"): *(Update: The "minimal type" of a term is the smallest (in the sense of the subtype relation) type possessed by that term.*

- ☐ `Bool -> (Top*Top)`
- ☐ `Bool -> (Bool*Bool)`
- ☒ `Top -> (Bool*Bool)`
- ☐ `Top -> Top`
- ☐ `Top`
- ☐ Some other type(s)
- ☐ Not typeable

$\boxed{11.3}$ Suppose `t = (\x:Bool, \y:Top->Bool, y x) true`

Check *all* the types `T` such that `|- t \in T` (or "Not typeable"):

- ☒ `(Top->Bool) -> Top`
- ☐ `(Bool->Bool) -> Bool` *(Corrected from an earlier answer key)*
- ☐ `(Top->Top) -> Top`
- ☒ Some other type(s)
- ☐ Not typeable

$\boxed{11.4}$ Which is the *minimal* type `T` such that `|- t \in T` (or check "Not typeable"):

- ☐ `(Top->Bool) -> Top`
- ☐ `(Bool->Bool) -> Bool`
- ☐ `(Top->Top) -> Top`
- ☒ Some other type(s)
- ☐ Not typeable

$\boxed{11.5}$ Are there any types `T` and `U` such that `x:T |- (\x:T. x x) \in U`?

☒ Yes ☐ No

If so, give one.

```
T = Top->Top
U = Top
```

$\boxed{11.6}$ Does the subtype relation contain an infinite, strictly descending chain — that is, 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}?

☒ Yes ☐ No

If you chose "Yes," then show to construct such a chain by giving its first four elements.

T_1 = `Top`
T_2 = `Top -> Top`
T_3 = `Top -> (Top -> Top)`
T_4 = `Top -> (Top -> (Top -> Top))`

12 **References** (8 points)

The simply typed lambda-calculus with references is summarized on page 5 of the accompanying handout.

Recall (from `References.v`) that the preservation theorem for this calculus is stated like this

```
Theorem preservation_theorem := forall ST t t' T st st',
  empty ; ST |- t \in T ->
  store_well_typed ST st ->
  t / st --> t' / st' ->
  exists ST',
     extends ST' ST /\
     empty ; ST' |- t' \in T /\
     store_well_typed ST' st'.
```

where:

- `st` and `st'` are *stores* (maps from locations to values);
- `ST` and `ST'` are *store typings* (maps from store locations to types);
- `empty ; ST |- t \in T` means that the closed term `t` has type `T` under the store typing `ST`;
- `t / st --> t' / st'` means that, starting with the store `st`, the term `t` steps to `t'` and changes the store to `st'`;
- `store_well_typed ST st` means that the contents of each location in the store `st` has the type associated with this location in `ST`; and
- `extends ST' ST` means that the domain of `ST` is a subset of that of `ST'` and that they agree on the types of common locations.

Briefly explain why the existential quantifier is needed in the statement of the preservation theorem. I.e., what would go wrong if we stated the theorem like this?

```
Theorem preservation_wrong2 : forall ST T t st t' st',
  empty ; ST |- t \in T ->
  t / st --> t' / st' ->
  store_well_typed ST st ->
  empty ; ST |- t' \in T.
```

Answer:

The `ST_RefValue` rule yields a new location `l`, which will appear in `t'` as the result of the `new` operation that has just been executed. But the original store `ST` will not contain a type for `l` (it will be one element too short), and the claimed typing derivation in the conclusion of `preservation_wrong2` will not exist (so the theorem will not be provable).

The existential quantifier in the good preservation theorem allows us to choose a one-element-larger store typing `ST'` in the case where `t` steps using `ST_RefValue`, where the new binding in `ST'` gives the new location `l` the type of the initial value in the new cell in the store.

For Reference

Simply Typed Lambda Calculus with Booleans and Unit

```
Syntax:

   T ::= T -> T                      arrow type
         Bool                        boolean type
         Unit                        unit type

   t ::= x                           variable
       | \x:T,t                      abstraction
       | t t                         application
       | true                        true
       | false                       false
       | if t then t else t          conditional
       | unit                        unit value

Values:

   v ::= \x:T,t
   | true
   | false
   | unit

Substitution:

   [x:=s]x               = s
   [x:=s]y               = y                        if x <> y
   [x:=s](\x:T, t)       = \x:T, t
   [x:=s](\y:T, t)       = \y:T, [x:=s]t            if x <> y
   [x:=s](t1 t2)         = ([x:=s]t1) ([x:=s]t2)
   [x:=s]true            = true
   [x:=s]false           = false
   [x:=s](if t1 then t2 else t3) = if [x:=s]t1 then [x:=s]t2 else [x:=s]t3
   [x:=s]unit            = unit


Small-step operational semantics:

                      value v2
             ---------------------------                  (ST_AppAbs)
             (\x:T2,t1) v2 --> [x:=v2]t1

                     t1 --> t1'
                   ----------------                       (ST_App1)
                   t1 t2 --> t1' t2
```

```
                    value v1
                    t2 --> t2'
                 ----------------                        (ST_App2)
                 v1 t2 --> v1 t2'

          --------------------------------               (ST_IfTrue)
          (if true then t1 else t2) --> t1

          ---------------------------------              (ST_IfFalse)
          (if false then t1 else t2) --> t2

                   t1 --> t1'
 --------------------------------------------------------    (ST_If)
 (if t1 then t2 else t3) --> (if t1' then t2 else t3)
```

Typing:

```
                  Gamma x = T1
                ------------------                       (T_Var)
                Gamma |- x \in T1

           x |-> T2 ; Gamma |- t1 \in T1
           -----------------------------                 (T_Abs)
            Gamma |- \x:T2,t1 \in T2->T1

           Gamma |- t1 \in T2->T1
             Gamma |- t2 \in T2
            ----------------------                       (T_App)
            Gamma |- t1 t2 \in T1

            ---------------------                        (T_True)
            Gamma |- true \in Bool

            ---------------------                        (T_False)
            Gamma |- false \in Bool

Gamma |- t1 \in Bool    Gamma |- t2 \in T1    Gamma |- t3 \in T1
---------------------------------------------------------------- (T_If)
           Gamma |- if t1 then t2 else t3 \in T1

            ---------------------                        (T_Unit)
            Gamma |- unit \in Unit
```

```
Lemma substitution_preserves_typing : forall Gamma x U t v T,
  x |-> U ; Gamma |- t \in T ->
  |- v \in U   ->
  Gamma |- [x:=v]t \in T.
```

Sum Types

(Based on the STLC with booleans and Unit.)

```
Syntax:

   T ::= ...
       | T + T                         sum type

   t ::= ...
       | inl T v                       tagged value (left)
       | inr T v                       tagged value (right)
       | case t of                     case
           inl x => t
         | inr x => t

Values:

   v ::= ...
       | inl v
       | inr v

Substitution:

   ...
   [x:=s](inl T t)       = inl T ([x:=s]t)
   [x:=s](inr T t)       = inr T ([x:=s]t)
   [x:=s](case t1 of inl y => t2 | inr y => t3)
     = case [x:=s]t1 of
           inl x => (if x=y then t2 else [x:=s]t2)
         | inr x => (if x=y then t3 else [x:=s]t3)


Small-step operational semantics:

                       t1 --> t1'
                ------------------------                   (ST_Inl)
                inl T2 t1 --> inl T2 t1'

                       t2 --> t2'
                ------------------------                   (ST_Inr)
                inr T1 t2 --> inr T1 t2'

                       t0 --> t0'
         -------------------------------------------       (ST_Case)
          case t0 of inl x1 => t1 | inr x2 => t2 -->
         case t0' of inl x1 => t1 | inr x2 => t2

      -----------------------------------------------      (ST_CaseInl)
      case (inl T2 v1) of inl x1 => t1 | inr x2 => t2
                     -->  [x1:=v1]t1

      -----------------------------------------------      (ST_CaseInr)
      case (inr T1 v2) of inl x1 => t1 | inr x2 => t2
                     -->  [x2:=v2]t2
```

Typing:

```
                 Gamma |- t1 \in T1
          ------------------------------                (T_Inl)
          Gamma |- inl T2 t1 \in T1 + T2

                 Gamma |- t2 \in T2
          -------------------------------               (T_Inr)
           Gamma |- inr T1 t2 \in T1 + T2

               Gamma |- t0 \in T1+T2
            x1|->T1; Gamma |- t1 \in T3
            x2|->T2; Gamma |- t2 \in T3
----------------------------------------------------    (T_Case)
Gamma |- case t0 of inl x1 => t1 | inr x2 => t2 \in T3
```

References

(Based on the STLC with booleans and Unit.)

```
Syntax:

   T ::= ...
       | Ref T             Ref type

   t ::= ...
      | ref t              allocation
      | !t                 dereference
      | t := t             assignment
      | l                  location

   v ::= ...
      | l                  location

Substitution:

   [x:=s](ref t)         = ref ([x:=s]t)
   [x:=s](!t)            = ! ([x:=s]t)
   [x:=s](t1 := t2)      = ([x:=s]t1) := ([x:=s]t2)
   [x:=s]l               = l

Small-step operational semantics:

                          value v2
            -------------------------------------   (ST_AppAbs)
            (\x:T2.t1) v2 / st --> [x:=v2]t1 / st

                   t1 / st --> t1' / st'
                 ---------------------------        (ST_App1)
                 t1 t2 / st --> t1' t2 / st'

              value v1     t2 / st --> t2' / st'
              ---------------------------------     (ST_App2)
                 v1 t2 / st --> v1 t2' / st'

                   t1 / st --> t1' / st'
                   ---------------------            (ST_Deref)
                   !t1 / st --> !t1' / st'

                           l < |st|
                 ---------------------------------- (ST_DerefLoc)
                 !(loc l) / st --> lookup l st / st

                   t1 / st --> t1' / st'
             -----------------------------------    (ST_Assign1)
             t1 := t2 / st --> t1' := t2 / st'


                   t2 / st --> t2' / st'
              ---------------------------------     (ST_Assign2)
              v1 := t2 / st --> v1 := t2' / st'

                           l < |st|
```

```
      -----------------------------------    (ST_Assign)
      loc l := v / st --> unit / [l:=v]st

              t1 / st --> t1' / st'
          ---------------------------        (ST_Ref)
          ref t1 / st --> ref t1' / st'

         --------------------------------    (ST_RefValue)
         ref v / st --> loc |st| / st,v
```

Typing:

```
                     l < |ST|
        --------------------------------------     (T_Loc)
        Gamma; ST |- loc l : Ref (lookup l ST)

               Gamma; ST |- t1 : T1
           ----------------------------            (T_Ref)
           Gamma; ST |- ref t1 : Ref T1

             Gamma; ST |- t1 : Ref T1
             -------------------------             (T_Deref)
               Gamma; ST |- !t1 : T1

             Gamma; ST |- t1 : Ref T2
               Gamma; ST |- t2 : T2
          ----------------------------             (T_Assign)
          Gamma; ST |- t1 := t2 : Unit
```

Products

(Based on the STLC with Booleans and Unit.)

```
Syntax:

   t ::=                Terms
       | ...
       | (t,t)             pair
       | t.fst             first projection
       | t.snd             second projection

   v ::=                Values
       | ...
       | (v,v)             pair value

   T ::=                Types
       | ...
       | T * T             product type


Small-step operational semantics:

                            t1 --> t1'
                       --------------------                      (ST_Pair1)
                       (t1,t2) --> (t1',t2)

                            t2 --> t2'
                       --------------------                      (ST_Pair2)
                       (v1,t2) --> (v1,t2')

                             t1 --> t1'
                         ------------------                      (ST_Fst1)
                         t1.fst --> t1'.fst

                        ------------------                       (ST_FstPair)
                        (v1,v2).fst --> v1

                             t1 --> t1'
                         ------------------                      (ST_Snd1)
                         t1.snd --> t1'.snd

                        ------------------                       (ST_SndPair)
                        (v1,v2).snd --> v2
```

Typing:

```
          Gamma |- t1 \in T1     Gamma |- t2 \in T2
          -----------------------------------------               (T_Pair)
                  Gamma |- (t1,t2) \in T1*T2

                   Gamma |- t0 \in T1*T2
                   ----------------------                          (T_Fst)
                   Gamma |- t0.fst \in T1

                   Gamma |- t0 \in T1*T2
                   ----------------------                          (T_Snd)
                   Gamma |- t0.snd \in T2
```

Subtyping

(Based on the STLC with Booleans, Unit, and Products.)

```
Syntax:

   T ::=                    Types
       | ...
       | Top                   top type

Subtyping:

                         S <: U    U <: T
                         ----------------               (S_Trans)
                              S <: T

                              ------                    (S_Refl)
                              T <: T

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

                       S1 <: T1    S2 <: T2
                       --------------------             (S_Prod)
                        S1 * S2 <: T1 * T2

                       T1 <: S1    S2 <: T2
                       --------------------             (S_Arrow)
                       S1 -> S2 <: T1 -> T2

                       S1 <: T1    S2 <: T2
                       --------------------             (S_Prod)
                          S1*S2 <: T1*T2

Typing:

                    Gamma |- t1 \in T1     T1 <: T2
                    -------------------------------     (T_Sub)
                          Gamma |- t1 \in T2
```