SOLUTIONS

1. **[Standard Track Only] Grab Bag** (13 points)
Mark the following statements as True or False. The exam appendices may be useful for the Hoare logic and STLC questions.

(a) In Coq, the type `False` is inhabited by the single value `nil`.
☐ True ☒ False

(b) In Coq, the term `(fun P ⇒ ∼P)` has type `Prop`.
☐ True ☒ False

(c) In Coq, the term `(fun (P: nat→ Prop) ⇒ ∀(m:nat), P m)` has type `(nat → Prop) → Prop`.
☒ True ☐ False

(d) In Coq, if we have a hypothesis `H : value t`, then the tactic `induction H.` will generate four new goals, one for each constructor. (The definition of `value` is in Appendix B.)
☒ True ☐ False

(e) For the following Imp program,

```
{{ X = 10 ∧ Y = 2 ∧ Z = 0 }}
while ∼(X = 0)
  X := X - Y
  Z := Z + 1
end
{{ Z = 5 }}
```

`10 = X + Y * Z ∧ Y = 2` is a valid loop invariant which can also be used to prove the given Hoare triple.
☒ True ☐ False

(f) For the same Imp program given in the previous problem, `X = Z * 2` is a valid loop invariant, but cannot prove the given Hoare triple.
☐ True ☒ False

(g) According to `cequiv`, an Imp command that doesn't terminate on any input is equivalent to every program `c`.
☐ True ☒ False

(h) For Imp programs, if `c1` is equivalent to `c` and `c2` is also equivalent to `c`, then for all `b`, `(if b then c1 else c2)` is equivalent to `c`.
☒ True ☐ False

(i) In STLC, the term `(\x:Bool, if true then false else true)` is a value.
☒ True ☐ False

(j) The usual purpose of type checking in a programming language is to prevent nontermination caused by divergence.
☐ True ☒ False

(k) The term `(\x:Bool, (\y:Bool, y) True) False` will single step to `([x:=True] (\y:Bool, y) True)`.
☒ True ☐ False

(l) If we extend the STLC with `fix`, it becomes possible to implement programs that diverge (i.e., go into an infinite loop).
☒ True ☐ False

(m) In the STLC extended with reference types, the *store typing* `ST` used in the typing judgment `Gamma ; ST ⊢ t ∈T` maps the heap (a.k.a. memory) locations to their static types.
☒ True ☐ False

2. **[Standard Only] Extremely Valid Hoare Triples** (12 points)

Recall that the definition of the semantics of Hoare triples is given by the following definition.

```
Definition hoare_triple
           (P : Assertion) (c : com) (Q : Assertion) : Prop :=
  ∀ st st',
     st =[ c ]⇒ st' →
     P st  →
     Q st'.
```

Appendix A contains a summary of the Imp semantics.

(a) Give a `P` such that for all `c` and all `Q`, `{{P}} c {{Q}}` is a valid Hoare triple.

```
P = False
```

(b) Give a `c` such that for all `P` and all `Q`, `{{P}} c {{Q}}` is a valid Hoare triple.

```
c = while true do skip end
```

(c) Give a `Q` such that for all `P` and all `c`, `{{P}} c {{Q}}` is a valid Hoare triple.

```
Q = True
```

3. **Reverse Hoare Triples** (12 points)

Consider this variant of a Hoare triple:

```
Definition reverse_triple
           (P : Assertion) (c : com) (Q : Assertion) : Prop :=
    ∀ st',
        Q st' →
        ∃ st, P st ∧ st =[ c ]⇒ st'.
```

Such a triple is sometimes used for something called *incorrectness logic*, which can help find bugs. The idea is that the post-condition `Q` specifies an *undesirable state* and the triple says that such a state is reachable (assuming termination) from an initial condition satisfying `P`.

We use `<< P >> c << Q >>` as notation for `reverse_triple P c Q`, and as usual, we say this triple is *valid* if the proposition holds and *invalid* otherwise. For example, we have the following instances:

```
<<True>> Z := 2 <<Z = 2>>         (* this triple is valid - any initial state is OK *)
<<True>> Z := 2 <<Z = 3>>         (* this triple is invalid - no initial state works *)
```

The triples below are all invalid. Find counterexamples `st'` that demonstrates this.

(a)

```
<< False >>
skip
<< Z = 2 >>
```

Counterexample: `st' = Z !→ 2`

(b)

```
<< True >>
Z := 2
<< True >>
```

Counterexample: `st' = Z !→ 3`

(c)

```
<< Z = 10 >>
if (X = 1)
    then Z := 42
    else skip
end
<< Z = 42 >>
```

Counterexample: `st' = Z !→ 42 ; X !→ 2`

4. **Simply-typed Lambda Calculus with Pairs** (22 points total)

Appendix B contains the syntax, small-step operational semantics, and typing relation for a variant of the simply-typed lambda calculus with `Bool` and pair types. Unlike the variant studied in class that used `fst` and `snd` "projection" operations, this version uses a pattern-matching operation called "split" to decompose a pair into its components. For instance, if `p` is a term that evaluates to `(v1, v2)` then the term `let (x,y) = p in x` will evaluate (in several steps) to `v1`. So the complete program below steps as shown:

`(\p : Bool * Bool, let (x, y) = p in x) (true, false)` ⟶* `true`.

The syntax and rules presented in the appendix are completely identical to those from the course notes except for those marked ★, which have to do with the new split operation. Note that the new syntax `let (x,y) = t1 in t2` binds the variables `x` and `y` for use in `t2`.

a. (4 points) As usual, the operational semantics depends on a notion of *substitution* `[x:=s]t`, which is defined inductively on the structure of `t`. All of the cases are given in the Appendix except for those dealing with split. Which of the following clauses should we add to the definition to properly define substitution? (Choose one or more.)

☐	`[x:=s](let (y,z) = t1 in t2)`	=	`let (y,z) = [x:=s]t1 in [x:=s]t2`	
☒	`[x:=s](let (y,z) = t1 in t2)`	=	`let (y,z) = [x:=s]t1 in t2`	*when* `x=y` or `x=z`
☐	`[x:=s](let (y,z) = t1 in t2)`	=	`let (y,z) = t1 in [x:=s]t2`	*when* `x=y` or `x=z`
☒	`[x:=s](let (y,z) = t1 in t2)`	=	`let (y,z) = [x:=s]t1 in [x:=s]t2`	*when* `x<>y` and `x<>z`
☐	`[x:=s](let (y,z) = t1 in t2)`	=	`let (y,z) = t1 in t2`	*when* `x<>y` and `x<>z`
☐	`[x:=s](let (x,x) = t1 in t2)`	=	`let (s,s) = [x:=s]t1 in [x:=s]t2`	

b. Using the rules in the appendix, there is exactly one possible typing derivation for the following claim:

`empty ⊢ \p : Bool * Bool, let (x,y) = p in x ∈ (Bool * Bool) → Bool`

(i) (3 points) What form will the typing context `Gamma` have at the point in the derivation where the rule `T_Var` is used to check the variable `p`? (Choose one.)

- ☐ `empty`
- ☐ `y ↦Bool ; x ↦Bool ; empty`
- ☒ `p ↦Bool * Bool ; empty`
- ☐ `y ↦Bool ; x ↦Bool ; p ↦Bool * Bool ; empty`
- ☐ `p ↦Bool * Bool ; y ↦Bool ; x ↦Bool ; empty`

(ii) (3 points) What form will the typing context `Gamma` have at the point in the derivation where the rule `T_Var` is used to check the variable `x`? (Choose one.)

- ☐ `empty`
- ☐ `y ↦Bool ; x ↦Bool ; empty`
- ☐ `p ↦Bool * Bool ; empty`
- ☒ `y ↦Bool ; x ↦Bool ; p ↦Bool * Bool ; empty`
- ☐ `p ↦Bool * Bool ; y ↦Bool ; x ↦Bool ; empty`

c. (3 points) Recall that a term is *closed* if it contains no free variables, which means, for well-typed terms, that it typechecks in an empty context. Now consider the following lemma, which would be a useful result about the values in this language:

```
Lemma value_closed:
  ∀ t, value t → ∃ T, empty ⊢ t ∈ T.
```

Unfortunately, this lemma is not provable. In the space below, provide a counterexample `t` that refutes the claim.

t = \x:Bool,y .

d. (3 points) Suppose we add the following rule to the step semantics of the language:

```
--------------- (ST_Pair3)
(s,t) --> (t,s)
```

Which of the following properties will *fail* for this version of the language?

☐ progress

☒ preservation

☒ determinacy of evaluation

☐ (they all remain valid)

e. (3 points) Suppose instead that we add the following typechecking rule:

```
Gamma x = T1*T2
----------------- (T_Var2)
Gamma |- x \in T1
```

Which of the following properties will *fail* for this version of the language?

☐ progress

☒ preservation

☐ determinacy of evaluation

☐ (they all remain valid)

f. (3 points) Suppose instead we add the following rule to the step semantics of the language:

```
value s    value t
------------------ (ST_Pair3)
(s,t) --> true
```

Which of the following properties will *fail* for this version of the language?

☐ progress

☒ preservation

☒ determinacy of evaluation

☐ (they all remain valid)

5. **STLC + Boxes** (36 points total)

In this problem, we consider a variant of the simply-typed lambda calculus with *boxed* values and a new type “box”, written □. The type □ is used to describe a “boxed” value, which is a value tagged with a run-time representation of its type. For instance, we write `[true : Bool]` for a boxed value `true` tagged with its type `Bool`. Similarly, we write `[\x:Bool,x : Bool → Bool]` for a boxed function value `\x:Bool, x` tagged with its type `Bool → Bool`. Importantly, all such boxes, regardless of their contents, have the *same* static type, □. So, when we “box” a value, we forget it static type.

To open a box and access its contents, we need to perform a *dynamic* type check that determines whether the contents of the box have a given type, `T`. We introduce new syntax `unbox - for - else`, and write `unbox t for (\x:T, t1) else t2` for this “dynamic type test” operation. Here, `t` must be a term that evaluates to a boxed value, `[v : U]` and the term after `for` must be (or evaluate to) a function value. Recall that the function argument is annotated with a type `T`. To evaluate `unbox`, we check whether `T = U`, and, if so, the program calls the function on `v`, otherwise, when the types are different, i.e. `T<>U`, the else branch is taken.

For example, according to the intended operational semantics we have the following three reduction sequences:

(1) `unbox [true:Bool]for (\x:Bool, x) else false ⟶(\x:Bool, x) true ⟶true`
(1) *applies the function* `(\x:Bool,x)` *to* `true` *because the type in the box,* `Bool`*, equals the type of* `x`

(2) `unbox [true:Bool]for (\f:Bool → Bool, f true) else false ⟶false`
(2) *takes the else branch because* `Bool <> (Bool → Bool)`

(3) `unbox [\x:Bool,x:Bool→ Bool]for (\f:Bool → Bool, f true) else false ⟶`
`(\f:Bool → Bool, f true) (\x:Bool, x) ⟶(\x:Bool, x) true ⟶true`
(3) *applies the function because the types are equal*

Boxed values are thus a simple model of languages like Python that support “dynamic types” and run-time type dispatch. In this problem we develop a type system and prove type safety for this feature. Appendix C shows the changes to the grammars for types and new terms for this language; the omitted terms are the usual ones for STLC with `Bool` as in Appendix B. (Note: for simplicity we *do not* include pairs in this problem.) The type system presented in Appendices B & C satisfies all of the key lemmas for STLC.

a. (3 points) Which of the new small-step semantics rules are considered to be *congruence rules*? (Mark all that apply.)

☒ `ST_Box` ☒ `ST_Unbox1` ☒ `ST_Unbox2` ☐ `ST_UnboxEQ` ☐ `ST_UnboxNEQ`

b. (3 points) Which of the following is the correct statement of the *canonical forms lemma* for the □ type?

☐ `∀ t T Gamma, value t → Gamma ⊢ t ∈ □ → ∃ v, t = [v:T]`

☐ `∀ t Gamma, value t → Gamma ⊢ t ∈ □ → ∀ v, ∃T, t = [v:T]`

☒ `∀ t Gamma, value t → Gamma ⊢ t ∈ □ → ∃ v, ∃T, t = [v:T]`

☐ `∀ t T Gamma v, value t → Gamma ⊢ t ∈ □ → t = [v:T]`

c. (9 points) Recall that the typing rules for STLC+□ in Appendices B & C are *syntax directed*, which means that which rule applies at some step of the derivation is uniquely determined by the syntax of the term. We say that such a rule *fails* if the syntax of the term matches the rule, but one or more of the hypotheses of the rule is not satisfied.

For each of the following terms in STLC+□, indicate whether the given typing judgment is derivable using the rules from Appendices B & C. If it is not derivable, write the name of a rule that fails for the derivation (there might be more than one) in the space provided. We have done the first two for you.

`x ↦ Bool ⊢ x ∈ Bool`

☒ is derivable ☐ is *not* derivable because ____________________________ fails

`empty ⊢ x ∈ Bool`

☐ is derivable ☒ is *not* derivable because ______T_Var______ fails

`empty ⊢ [true : Bool] ∈ □`

☒ is derivable ☐ is *not* derivable because fails

`empty ⊢ \x:□, unbox x for (\b:Bool, x) else false ∈ □ → Bool`

☐ is derivable ☒ is *not* derivable because (`T_Unbox` or `T_Var`) *are both correct* fails

`empty ⊢ \x:□, unbox x for (\b:Bool, b) else (\c:Bool,false) ∈ □ → Bool`

☐ is derivable ☒ is *not* derivable because `T_Unbox` fails

d. (3 points) Suppose we change the typing rule `T_Box` to the one shown below, rather than the one in the appendix—note that `U` appears as the annotation in the box.

```
    Gamma  ⊢  t ∈ T
--------------------- (T_Box')
Gamma  ⊢  [t:U] ∈ □
```

Which of the following properties will *fail* for this version of the language?

☐ progress

☒ preservation

☐ determinacy of evaluation

☐ (they all remain valid)

e. (3 points) Suppose we instead change the typing rule `T_Box` to the one shown below, rather than the one in the appendix—note that `t` is given type □ in the premise.

```
    Gamma ⊢ t ∈ □
--------------------- (T_Box')
Gamma ⊢ [t:T] ∈ □
```

Which of the following properties will *fail* for this version of the language?

☐ progress

☐ preservation

☐ determinacy of evaluation

☒ (they all remain valid)

f. (3 points) Suppose we instead change the typing rule `T_Unbox` to the one shown below, rather than the one in the appendix—note that the type of `t2` is just `U` in the premise.

```
Gamma ⊢ t1 ∈ □     Gamma ⊢ t2 ∈ U    Gamma ⊢ t3 ∈ U
-----------------------------------------------------------------      (T_Unbox)
            Gamma ⊢ unbox t1 for t2 else t3 ∈ U
```

Which of the following properties will *fail* for this version of the language?

☒ progress

☒ preservation

☐ determinacy of evaluation

☐ (they all remain valid)

g. (6 points) It turns out that, because STLC+□ is so simple, there is a *closed* value for any type `T`. Fill in the blanks below to complete the function to produce such a value.

```
Fixpoint cv (T:ty) : tm :=
  match T with
  | Ty_Bool ⇒ tm_true
  | Ty_Arrow U V ⇒ tm_abs x U (cv V)
  | Ty_Box ⇒ tm_box <{true}> Ty_Bool
  end.
```

Your code above should be such that the following lemma holds:

Lemma 1 (Values of any type)**.** `∀ T, value (cv T) ∧ empty ⊢ (cv T) ∈ T`

The addition of this "box" construct does not seem that powerful at first glance, because it simply lets us write programs that test for type information at runtime. However, it is now possible to write a program that, when run, goes into an infinite loop—something that is impossible in just STLC! (And, with a bit more work, it is possible to implement the `fix` operator that implements general recursion.)

h. (3 points) To see how, first note that there is a type `T` such that the following judgment is derivable according to the rules of STLC+□. (There is only one possible type for this program.)

`empty ⊢ \x:□, unbox x for (\f:T, f x) else x ∈ T`

What type can be filled in for `T`?

☐ `T = □`

☒ `T = □→ □`

☐ `T = (□→ □) → □`

☐ `T = □→ □→ □`

i. (3 points) Next, let us abbreviate the program above as `m`:

`m = \x:□, unbox x for (\f:T, f x) else x`

Then, for some type `U`, we also have this well-typed program: `empty ⊢ m [m:T] ∈ U`. Looking at the operational semantics, we have the following sequence of steps, which demonstrates the infinite loop:

`m [m:T] ⟶ ? ⟶ ...⟶ m [m:T]`

What term should be filled in for `?` above as the result of the first step of evaluation?

☐ `unbox m for (\f:T, f m) else m`

☒ `unbox [m:T] for (\f:T, f [m:T]) else [m:T]`

How many more steps (after this first one) does it take to reach `m [m:T]` for the first time?

☐ 1 ☒ 2 ☐ 3 ☐ 4 ☐ 5

(**Note:** Advanced Track students may want to continue to problem 7 before doing problem 6.)

6. **Subtyping** (15 points total)

Appendix E contains the additions needed to add subtyping to the simply-typed lambda calculus defined in Appendix B. In particular we extend the types `T` to include `Top`, add the *subsumption* rule `T_Sub` to the type system, and define type subtyping relation `S <: T` as shown in the appendix. The definition of values, substitution, and the small-step semantics remain unchanged.

a. (5 points) For each of claims below, mark the box if it is a *valid* subtyping relation according to the rules in Appendix E.

☐ `Top <: Bool`

☒ `(Bool, Bool) <: (Top, Bool)`

☒ `(Bool → Top) → Bool <: (Top → Bool) → Bool.`

☒ `(Bool → Top) → (Bool → Bool) <: (Top → Bool) → Top.`

☐ `(Bool → Top) → (Bool → Bool) <: Top → Top`

b. (5 points) Give a *short* example term of that is well-typed at type `Bool` in the empty context and such that its typing derivation *must* use the rule `T_Sub`. Then fill in the blanks below to indicate which types are needed for the use of `T_Sub`

`empty ⊢ (x:Top.true) false ∈Bool`

The derivation relies on an instance of `T_Sub` in which:

`T1 = Bool`

`T2 = Top`

c. (2 points) How many types `U` exist such that `Bool <: U` (according to the rules of in the appendix?) (choose one)

☐ 0 ☐ 1 ☒ 2 ☐ infinitely many

d. (3 points) We saw in class that there are various *inversion* lemmas related to subtyping. For instance, we proved that `∀ U, U <: Bool → U = Bool`. Here we consider the inversion of *supertypes*. What is the appropriate property that can be filled in for `???` such that the following lemma is provable? (choose one)

```
Lemma sub_inversion_pair2 : ∀ S1 S2, S1 * S2 <: U → ???
```

☐ `U = Top`

☐ `∃ T1 T2, U <: T1 * T2`

☐ `U = Top ∨ ∃T1 T2, U = T1 * T2 ∧ T1 <: S1 ∧ T2 <: S2`

☒ `U = Top ∨ ∃T1 T2, U = T1 * T2 ∧ S1 <: T1 ∧ S2 <: T2`

☐ `False`

7. **[Advanced Only] Formal Proof** (25 points total)

A program translation is called *type directed* when it is defined by using the structure of the typing derivation for a (well-typed) term. In this problem, we prove that a simple translation preserves typing. Appendix D defines a type-directed translation on the STCL+□ language used in Problem 5. This translation "boxes" the type `Bool` in a source program by replacing `Bool` with □ and putting boolean constants into boxes. The hard part is unboxing them for use in conditionals.

The translation is given in two parts. First, we define a type translation `T†` that converts each occurrence of `Bool` in `T` into □. Its definition is shown at the top of Appendix D. Then we define a type-directed translation judgment `Gamma ⊢ t ⇝ t' ∈T`. This judgment indicates that the source term `t` translates to the target term `t'`. The rules that define the translation (also given in Appendix D) are inductive, and they mirror the typing rules—in particular, the context `Gamma` and the type `T` correspond to the *source* program (they don't have translated types). As such, we can think of this translation as "decorating" a typing derivation of the program `t`, and it is easy to prove by straight-forward induction the following lemma:

Lemma 2 (Well-typed Source)**.** *If* `Gamma ⊢ t ⇝ t' ∈ T` *then* `Gamma ⊢ t ∈ T`.

In this problem you will prove the more interesting result, namely:

Lemma 3 (Well-typed Target)**.** *If* `Gamma ⊢ t ⇝ t' ∈T` *then* `Gamma† ⊢ t' ∈T†`.

Here, `Gamma†` is the "translated" target context, i.e., the one such that `Gamma x = Some T ↔ Gamma† x = Some (T†)`.

Looking carefully at the translation, you will see that it mostly the identity—most rules just apply the translation recursively to the subterms. However, note that the type annotations in lambda abstraction and boxes are translated. The interesting rules are `TR_True` and `TR_False`, which replace Boolean constants by their boxed versions, and, most interestingly, the `TR_If` rule.

`TR_If` unboxes the guard expression `t1'` as a `Bool` to perform the conditional. There are two wrinkles. First, because the `unbox` operation needs a function, this translation introduces new lambda-bound variables, which must not otherwise be used in `Gamma`—we write "`x` is fresh for `Gamma`" to indicate that. Second, the `unbox` operation needs an "else" case, which should be the same type as the (translated) branches, but those are of type `T1†`. Our translation will ensure (though we will not prove it) that the unboxing never fails, so it doesn't matter what term we put as the "else" case; but nevertheless, we need to provide some term, which we write as `error T1†`.

a. (3 points) Let us address the second problem first. In the Problem 5, we implemented a function `cv` that can produce well-typed values of any type. We can therefore take `error T = cv T` and prove:

Lemma 4 (Error well typed)**.**
`Gamma ⊢ error T1†: T1†`

Proof. The proof follows immediately from Lemma 1 plus a use of: (choose 1)

- ☐ Substitution preserves typing
- ☒ Weakening
- ☐ Preservation
- ☐ Canonical forms

☐

[Advanced Only]

b. To handle the problem that the translation introduces new variables, we need to more carefully account for how the source and target contexts are related. We therefore define the following inductive relation ⊆ †, which explains how a context `Gamma` relates to a translated context `Gamma'`.

```
∀ x T, Gamma x = Some T → Gamma' x = Some T†
-------------------------------------------------    (G_Related)
              Gamma ⊆† Gamma'

  Gamma ⊆† Gamma'   x is fresh for Gamma
-------------------------------------------------    (G_Fresh)
         Gamma ⊆† (x ↦ Bool ; Gamma')


              Gamma ⊆† Gamma'
-------------------------------------------------    (G_Extend)
     (x ↦ T ; Gamma) ⊆† (x ↦ T† ; Gamma')
```

We have the following lemma:

Lemma 5 (Translated Contexts)**.**
`∀ Gamma Gamma' x T, Gamma ⊆ †Gamma' → Gamma x = Some T → Gamma' x = Some T†.`

We can finally state a strong enough lemma to prove that the translation produces well-typed terms. Fill in the three indicated cases (we omit the other cases) to complete the proof. You may use Lemmas 4 and 5 where needed. In the inductive cases, be explicit about the form of any induction hypotheses and how you use them, and indicate the inference rules using the names provided.

Lemma 6 (Translation well-typed)**.**
`∀ Gamma t t' T, Gamma ⊢ t ⇝ t' ∈T → ∀Gamma', Gamma ⊆ †Gamma' → Gamma' ⊢ t' ∈T†.`

Proof. The proof proceeds by induction on the derivation that `Gamma ⊢ t ⇝ t' ∈ T`. We consider that cases based on the last rule applied in the derivation. Most of them follow by straightforward induction; we consider the most interesting cases below:

Case `TR_Var` (4 points) Then `t = x` and `t' = x` and we have `Gamma x = Some T` and, by assumption `Gamma ⊆ †Gamma'`. We need to show...

Fill in here:

We need to show that `Gamma' x = T†` and conclude using `T_Var`, but `Gamma' x = T†` follows immediately from Lemma 5, the assumption that `Gamma ⊆ †Gamma'`, and the fact that `Gamma x = Some T`.

[Advanced Only]

Case `TR_Abs` (8 points) Then `T = U → V` for some `U` and `V` and we have `t = \x:U, t1` for some `x` and `t1`. We also have `t' = \x:U†, t1'`, where we know `(x ↦U; Gamma) ⊢ t1 ⇝ t1' ∈V`. Furthermore, `Gamma ⊆ †Gamma'`. We need to show...

We need to show `Gamma' ⊢ \x : U†, t1' ∈(U → V)†`, and since `(U → V)† = U† → V†` we can conclude using the rule `T_Abs` if we can show `(x ↦U†;Gamma') ⊢ t' ∈V'`. Our induction hypothesis says that

`∀ Gamma'', (x ↦U; Gamma) ⊆ †Gamma'' → Gamma'' ⊢ t1' ∈V†`

So the desired result follows from the induction hypothesis by instantiating with `Gamma'' = (x ↦U†; Gamma')` so long as `(x ↦U; Gamma) ⊆ †(x ↦U†; Gamma')`, but that follows from `G_Extend` and the assumption.

[Advanced Only]

Case `TR_If` (10 points) Then `t = if t1 then t2 else t3` for some `t1`, `t2`, and `t3`, where we also know that `Gamma` ⊢ `t1` ⇝ `t1'` ∈`Bool` and `Gamma` ⊢ `t2` ⇝ `t2'` ∈`U` and `Gamma` ⊢ `t3` ⇝ `t3'` ∈`U` for some `U`. Furthermore, `Gamma` ⊆ †`Gamma'`.The translated term `t'` is `unbox t1' for (\x:Bool, if x then t2' else t3') else (error U†)`, where `x` is fresh for `Gamma`. We must show that...

`Gamma'` ⊢ `t'` ∈`U†`. We have three induction hypotheses:

(1) ∀ `Gamma''`, `Gamma` ⊆ †`Gamma''` → `Gamma''` ⊢ `t1'` ∈ `Bool†`

(2) ∀ `Gamma''`, `Gamma` ⊆ †`Gamma''` → `Gamma''` ⊢ `t2'` ∈`U†`

(3) ∀ `Gamma''`, `Gamma` ⊆ †`Gamma''` → `Gamma''` ⊢ `t3'` ∈`U†`

The desired conclusion follows by applying rule `T_Unbox`, assuming we can show:

(a) `Gamma'` ⊢ `t1'` ∈□, but this follows from (1), choosing `Gamma'' = Gamma'` and observing that `Bool† =` □.

(b) `Gamma'` ⊢ `(\x:Bool, if x then t2' else t3')` ∈`U†`. This follows using `T_Abs` and `T_If` because we have `(x` ↦`Bool; Gamma')` ⊢ `x : Bool` by `T_Var`, `(x` ↦`Bool; Gamma')` ⊢ `t2'` ∈`U†` by (2), choosing `Gamma'' = (x` ↦`Bool; Gamma')`, where we show `Gamma` ⊆ †`(x` ↦`Bool; Gamma')` by rule `G_Fresh`. The case for `t3'` follows similarly.

(c) Finally, we must also show that `Gamma'` ⊢ `error (U†) : U†`, but this follows immediately from Lemma 4.

CIS 5000 2022 Final Exam Appendices

(Do not write answers in the appendices. They will not be graded)

Appendix A: Imp Semantics and Hoare Logic Rules

Imp Large Step Semantics

```
                  -----------------                                    (E_Skip)
                  st =[ skip ]⇒ st

                  aeval st a = n
          ------------------------------                               (E_Asgn)
          st =[ x := a ]⇒ (x !→ n ; st)

                  st  =[ c1 ]⇒ st'
                  st' =[ c2 ]⇒ st''
                ---------------------                                  (E_Seq)
                st =[ c1;c2 ]⇒ st''

                 beval st b = true
                  st =[ c1 ]⇒ st'
       --------------------------------------                          (E_IfTrue)
       st =[ if b then c1 else c2 end ]⇒ st'

                beval st b = false
                  st =[ c2 ]⇒ st'
       --------------------------------------                          (E_IfFalse)
       st =[ if b then c1 else c2 end ]⇒ st'

                beval st b = false
           -----------------------------                               (E_WhileFalse)
           st =[ while b do c end ]⇒ st

                 beval st b = true
                  st =[ c ]⇒ st'
         st' =[ while b do c end ]⇒ st''
         -------------------------------                               (E_WhileTrue)
         st  =[ while b do c end ]⇒ st''

Definition cequiv (c1 c2 : com) : Prop :=
  ∀ (st st' : state),
    (st =[ c1 ]⇒ st') ↔ (st =[ c2 ]⇒ st').
```

Imp Hoare Logic Rules

```
               --------------------------- (hoare_asgn)
               {{Q [X ↦ a]}} X:=a {{Q}}

               --------------------  (hoare_skip)
               {{ P }} skip {{ P }}

                 {{ P }} c1 {{ Q }}
                 {{ Q }} c2 {{ R }}
                ----------------------  (hoare_seq)
                {{ P }} c1;c2 {{ R }}

                {{P ∧   b}} c1 {{Q}}
                {{P ∧ ∼ b}} c2 {{Q}}
        -----------------------------------  (hoare_if)
        {{P}} if b then c1 else c2 end {{Q}}

                 {{P ∧ b}} c {{P}}
          ----------------------------------  (hoare_while)
          {{P}} while b do c end {{P ∧ ∼ b}}

                  {{P'}} c {{Q'}}
                     P ↠ P'
                     Q' ↠ Q
           -----------------------------   (hoare_consequence)
                  {{P}} c {{Q}}
```

Appendix B: Simply-typed Lambda Calculus

This appendix contains the syntax, small-step operational semantics, and typing relation for a variant of the simply-typed lambda calculus with `Bool` and pair types. Unlike the variant studied in class (which used `fst` and `snd` "projection" operations), this version uses a pattern-matching operation called "split" to decompose a product into its components. The syntax and rules presented here are completely identical to those from the course notes except for those marked ★, which have to do with the new `split` operation.

Syntax and Types

```
t ::=                     Terms
    | x                       variable
    | \x:T,t                  abstraction
    | t t                     application
    | true                    constant true
    | false                   constant false
    | if t then t else t      conditional
    | (t, t)                  pair
    | let (x,y) = t in t      split ★

T ::=                     Types
    | T → T                   arrow type
    | Bool                    Boolean type
    | T * T                   product type
```

```
Inductive value : tm → Prop :=
  | v_abs : ∀ x T2 t1, value <{\x:T2, t1}>
  | v_true : value <{ true }>
  | v_false : value <{ false }>
  | v_pair : ∀ v1 v2,  value v1 → value v2 → value <{(v1, v2)}>.
```

Substitution

Note: this definition is *incomplete*.

```
[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](t1, t2)         = ([x:=s]t1, [x:=s]t2)
[x:=s](let (y,z) = t1 in t2) =                ★
    // omitted
```

STLC: 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)


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

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


                 t1 ⟶ t1'                           (ST_Split1)★
-----------------------------------------------
 let (x,y) = t1 in t2 ⟶ let (x,y) = t1' in t2


           value v1       value v2
------------------------------------------------    (ST_Split2)★
 let (x,y) = (v1,v2) in t2 ⟶ [x:=v1][y:=v2]t2
```

STLC: typing relation

```
                    Gamma x = T1
                  -----------------                        (T_Var)
                  Gamma ⊢ x ∈ T1

            x ↦ T2 ; Gamma ⊢ t1 ∈ T1
            -----------------------------                  (T_Abs)
            Gamma ⊢ \x:T2,t1 ∈ T2→T1

              Gamma ⊢ t1 ∈ T2→T1
                Gamma ⊢ t2 ∈ T2
               ----------------------                      (T_App)
               Gamma ⊢ t1 t2 ∈ T1

               ---------------------                       (T_True)
               Gamma ⊢ true ∈ Bool

               ---------------------                       (T_False)
               Gamma ⊢ false ∈ Bool

Gamma ⊢ t1 ∈ Bool    Gamma ⊢ t2 ∈ T1    Gamma ⊢ t3 ∈ T1
-------------------------------------------------------------   (T_If)
           Gamma ⊢ if t1 then t2 else t3 ∈ T1

      Gamma ⊢ t1 ∈ T1     Gamma ⊢ t2 ∈ T2
      -----------------------------------------              (T_Pair)
            Gamma ⊢ (t1,t2) ∈ T1*T2

  Gamma ⊢ t1 ∈ T1 * T2   y ↦ T2; x ↦ T1; Gamma ⊢ t2 ∈ T2
  -------------------------------------------------------------   (T_Split)★
          Gamma ⊢ let (x,y) = t1 in t2 ∈ T2
```

Key Lemmas for STLC

```
Lemma canonical_forms_bool : ∀ t,
empty  ⊢ t ∈ Bool → value t → t = <{true}> ∨ t = <{false}>.

Lemma canonical_forms_arrow : ∀ t,
 empty ⊢ t ∈ (T1 → T2) → value t → ∃ x t1, t = <{\x : T1, t1}>.

Lemma substitution_preserves_typing : ∀ Gamma x U t v T,
  (x ↦ U ; Gamma) ⊢ t ∈ T →
  empty ⊢ v ∈ U   →
  Gamma ⊢ [x:=v]t ∈ T.

Theorem progress : ∀ t T,
empty ⊢ t ∈ T →
  value t ∨ ∃ t', t ⟶ t'.

Theorem preservation : ∀ t t' T,
empty ⊢ t ∈ T →
  t ⟶ t' →
empty ⊢ t' ∈ T.

Definition deterministic {X : Type} (R : relation X) :=
  ∀ x y1 y2 : X, R x y1 → R x y2 → y1 = y2.

Theorem step_deterministic:
  deterministic step.
```

Appendix C: Box Types

Changes to base STLC

```
T ::=    Types                          t ::=   Terms
    |  T → T    arrow types                 |   ...
    |  Bool     Boolean type                |   [t : T]                  box
    |  □        box type                    |   unbox t for t else t     unbox
```

Substitution and Values

```
... (* usual rules, plus: *)
[x:=s] [t : T]   =   [[x:=s] t : T]
[x:=s](unbox t1 for t2 else t3) = unbox [x:=s]t1 for [x:=s]t2 else [x:=s]t3
```

```
Inductive value : tm → Prop :=
| v_abs : ∀ x T2 t1, value <{\x:T2, t1}>
| v_true : value <{ true }>
| v_false : value <{ false }>
| v_box : ∀ v T, value v → value <{ [v : T] }>.
```

Small-step semantics

```
              t1 ⟶ t1'
        -----------------------                    (ST_Box)
      [t1 : T] ⟶ [t1' : T]

              t1 ⟶ t1'
----------------------------------------------------     (ST_Unbox1)
unbox t1 for t2 else t3 ⟶ unbox t1' for t2 else t3

              value v1
              t2 ⟶ t2'
----------------------------------------------------     (ST_Unbox2)
unbox v1 for t2 else t3 ⟶ unbox v1 for t2' else t3

              value v
------------------------------------------------------   (ST_UnboxEQ)
unbox [v : T] for (\x:T, t1) else t2 ⟶ (\x:T, t1) v

              value v
               T <> U
-----------------------------------------------   (ST_UnboxNEQ)
   unbox [v : U] for (\x:T, t1) else t2 ⟶ t2
```

Typing Rules

```
          Gamma ⊢ t ∈ T
     --------------------------          (T_Box)
     Gamma ⊢ [t : T] ∈ □


Gamma ⊢ t1 ∈ □    Gamma ⊢ t2 ∈ (T → U)   Gamma ⊢ t3 ∈ U
------------------------------------------------------------------      (T_Unbox)
            Gamma ⊢ unbox t1 for t2 else t3 ∈ U
```

Appendix D: [ADVANCED ONLY] - Box Translation

```
                         TYPE TRANSLATION :
                         (Bool)† = □
                     (T1 → T2)† = (T1)† → (T2)†
                             □† = □

                   TYPE-DIRECTED TERM TRANSLATION:

                          Gamma x = T1
                       ----------------------                 (TR_Var)
                        Gamma ⊢ x ⇝ x ∈ T1

                x ↦ T2 ; Gamma ⊢ t1 ⇝ t1' ∈ T1
            ------------------------------------------        (TR_Abs)
           Gamma ⊢ \x:T2,t1 ⇝ \x:T2†, t1' ∈ T2→T1

                 Gamma ⊢ t1 ⇝ t1' ∈ T2→T1
                   Gamma ⊢ t2 ⇝ t2' ∈ T2
                  -------------------------------             (TR_App)
                  Gamma ⊢ t1 t2 ⇝ t1' t2' ∈ T1

                ------------------------------------          (TR_True)
                Gamma ⊢ true ⇝ [true:Bool] ∈ Bool

              --------------------------------------          (TR_False)
              Gamma ⊢ false ⇝ [false:Bool] ∈ Bool


                  Gamma ⊢ t1 ⇝ t1' ∈ Bool
                  Gamma ⊢ t2 ⇝ t2' ∈ T1
                  Gamma ⊢ t3 ⇝ t3' ∈ T1      (x fresh for Gamma)
-------------------------------------------------------       (TR_If)
 Gamma ⊢ if t1 then t2 else t3  ⇝
      unbox t1' for (\x:Bool, if x then t2' else t3')
                else (error T1†)

                      Gamma ⊢ t ⇝ t' ∈ T
                     --------------------------               (TR_Box)
                    Gamma ⊢ [t' : T†] ∈ □


                      Gamma ⊢ t1 ⇝ t1' ∈ □
                      Gamma ⊢ t2 ⇝ t2' ∈ (T → U)
                      Gamma ⊢ t3 ⇝ t3' ∈ U
---------------------------------------------                 (TR_Unbox)
     Gamma ⊢ unbox t1 for t2 else t3 ⇝
                      unbox t1' for t2' else t3' ∈ U
```

Appendix E: Subtyping

Additions to the type system:

```
T :=            Types
    | T → T
    | Bool
    | T * T
    | Top        ( added )


Gamma ⊢ t1 ∈ T1      T1 <: T2
-------------------------------              (T_Sub)
      Gamma ⊢ t1 ∈ T2
```

Subtyping relation:

```
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
```