CIS 5000: Software Foundations **Final Exam**

December 19, 2023

Solutions

1 [Standard Track Only] Miscellaneous (14 points)

1.1 The type `True` in Coq has a single constructor `I` with no arguments.

[✓ True] False

1.2 The axiom of *functional extensionality* states that

```
forall (A B : Type) (f g : A -> B), f = g -> forall x : A, f x = g x
```

True [✓ False]

1.3 It is possible to prove the following theorem without induction and without referring to facts from the Coq standard library.

```
Theorem mul_0_r : forall n: nat, n * 0 = 0
```

True [✓ False]

1.4 For Imp programs, if `c` is equivalent to `if b then c1 else c2` for all `b`, then it must be that `c1` and `c2` are equivalent.

[✓ True] False

1.5 If `c1` and `c2` are equivalent and `c1` terminates on all inputs, then `c2` terminates on all inputs.

[✓ True] False

1.6 If `c1` and `c2` both diverge from the same set of starting states, then they are equivalent.

True [✓ False]

1.7 If the Hoare triple `{{P}} c {{Q}}` is *valid*, then `c` is guaranteed to terminate when started in any state satisfying `P`.

True [✓ False]

1.8 If there exists a Hoare triple that two programs both satisfy, then the two programs must be equivalent.

True [✓ False]

$\boxed{1.9}$ What is the type of the Coq term `(fun b:bool => b = false)`?

- ☐ `Prop`
- ☐ `bool`
- ☐ `bool -> bool`
- ☒ `bool -> Prop`
- ☐ `Prop -> Prop`
- ☐ something else
- ☐ ill typed

$\boxed{1.10}$ What is the type of the Coq term `(fun b:bool => if b then false else true)`?

- ☐ `Prop`
- ☐ `bool`
- ☒ `bool -> bool`
- ☐ `bool -> Prop`
- ☐ `Prop -> Prop`
- ☐ something else
- ☐ ill typed

$\boxed{1.11}$ What is the type of the Coq term `(forall b:bool, b = false)`?

- ☒ `Prop`
- ☐ `bool`
- ☐ `bool -> bool`
- ☐ `bool -> Prop`
- ☐ `forall b:bool, b = false`
- ☐ something else
- ☐ ill typed

$\boxed{1.12}$ What is the type of the Coq term `((fun P:Prop => P) (true = false))`?

- ☒ `Prop`
- ☐ `bool`
- ☐ `bool -> Prop`
- ☐ `bool -> False`
- ☐ `Prop -> False`
- ☐ something else
- ☐ ill typed

$\boxed{2}$ **Semantic Styles** (8 points)

Briefly explain the difference between big-step and small-step styles of operational semantics. What are the advantages of the small-step style compared to the big-step 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. The major disadvantage of big-step definitions 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 also sometimes preferred because they are closer to implementations; in particular, they explicitly expose order of evaluation.

Grading scheme: 1.5 points for saying that big steps relate terms to final values. 1.5 points for an advantage (e.g. big step semantics are often more concise). 1.5 for saying that that small steps describe the intermediate states of computation. 1.5 for an advantage (e.g. that it doesn't conflate "nonterminating" with "stuck"). 2 "fudge points" for not saying anything bogus and for giving clear and precise explanations of the above.

$\boxed{3}$ **Representing Multisets** (16 points)

Recall that a *multiset* (also known as a *bag*) is similar to a set, i.e., the order of its elements does not matter, but it allows elements to appear many times.

In the homework for the `Lists.v` chapter, we represented a multiset as a list of natural numbers. We will return to the list representation later; for now, consider an alternative *functional representation*:

```
Definition func_multiset := nat -> nat.
```

That is, we represent multisets as functions from `nat` to `nat`. The input is the element, and the output is the number of times the element appears in the multiset. For example, the multiset containing the elements `1`, `2`, `2`, and `3` would be represented as a function that returns `1` when called with argument `1`; `2` for argument `2`; `1` for argument `3`; and `0` for any other argument.

$\boxed{3.1}$ Fill in the definition of a function `count`, which takes in an element `e` and a multiset `fm` and returns the number of times `e` appears in `fm`.

```
Definition count (e : nat) (fm : func_multiset) : nat :=

  fm e.
```

$\boxed{3.2}$ Fill in the definition of `singleton`, which takes in an element `e` and returns a multiset containing just the element `e`.

```
Definition singleton (e : nat) : func_multiset :=

  fun x => if x =? e then 1 else 0.
```

$\boxed{3.3}$ Fill in the definition of `sum`, which takes in two multisets `a`, `b` and returns a multiset that contains all of the elements of `a` and `b`. For example, if `a` contains elements `1` and `2` and `b` contains elements `2` and `3`, then `sum a b` should contain elements `1`, `2`, `2`, and `3`.

```
Definition sum (a b : func_multiset) : func_multiset:=

  fun x => a x + b x.
```

$\boxed{3.4}$ Fill in the definition of `is_empty`, which takes in a multiset `fm` and returns a *proposition* stating that `fm` contains no elements.

```
Definition is_empty (fm : func_multiset) : Prop :=

  forall e, fm e = 0.
```

Next, we return to the list representation of multisets from `Lists.v`:

```
Definition list_multiset := list nat.
```

That is, a multiset is represented as a list of `nat`s, where each number may appear in the list zero, one, or several times.

For example, the empty list represents the multiset containing no elements, while the list `[2, 1, 3, 2]` represents the multiset containing elements `1`, `2`, `2`, and `3`.

$\boxed{3.5}$ Fill in the definition of this conversion function from a list-represented multiset to a function-represented multiset. The input and output multisets should contain exactly the same elements. Use the functions defined above as appropriate.

```
Fixpoint list_to_func (lm : list_multiset) : func_multiset :=

  match lm with
  | [] => fun x => 0
  | (v :: vs) => sum (list_to_func vs) (singleton v)
  end.
```

$\boxed{3.6}$ Finally, let us consider whether we can write a conversion function in the other direction, from the function representation to a list representation. For which of the following types `A` can we write a `Fixpoint` that converts a `func_multiset` whose elements are of type `A` to a `list_multiset` whose elements are of type `A`? Choose all that apply.

☐ `nat`

☒ `bool`

☐ `nat * nat`

☒ `bool * bool`

☐ `list nat`

☐ `list bool`

☐ none of the above

$\boxed{4}$ **Imp with Subroutines** (15 points)

In this problem, we will explore operational semantics, behavioral equivalence, and Hoare-style reasoning for an Imp-like language extended with named, parameterless subroutines.

Syntax:

Here is the syntax of commands in this language:

```
c := skip
   | x := a
   | c ; c
   | if b then c else c end
   | call r
```

Some points to note:

- Subroutines do not take parameters or return results: they communicate with their callers through the variables in the global store.
- There is no explicit `return` command: a subroutine returns to its caller by "falling off the end."
- We've dropped `while` loops from the command syntax for brevity. If needed, loops can be simulated using subroutines.

We define a *program* to be an Imp command (the "top-level program") plus a collection of named subroutines (the "code context"), each of which is just a command. To invoke subroutines, we add a new form of command, written `call r`, to the syntax of commands.

For example, if we define the code context `R1` like this

```
Definition R1 : code_context :=
  (
   A |->
          <{ X := 1; call B; Z := 3 }>;
   B |->
          <{ Y := 2 }>
  ).
```

and the top-level program consists of the context `R1` plus the top-level command `call A`, written `(R1, call A)`, then the effect of running this program will be to set the variable `X` to `1`, `Y` to `2`, and `Z` to `3`.

Big-step semantics:

The operational semantics of arithmetic and boolean expressions remains unchanged from plain Imp.

The big-step reduction judgement for commands is written `R |-- st =[ c ]=> st'` and pronounced "In the code context `R`, the command `c` started in state `st` halts in state `st'`."

Most of the rules are basically identical to those in standard Imp (just adding the `R` parameter everywhere for the code context). The only new rule is the last one, which says that the command `call r` is executed by looking up `r` in `R` and executing the command that we find.

```
Inductive ceval : code_context -> com -> state -> state -> Prop :=
  (* All these rules are unchanged except for adding "R |--" *)
  | E_Skip : forall R st,
      R |-- st =[ skip ]=> st
  | E_Asgn  : forall R st a n x,
      aeval st a = n ->
      R |-- st =[ x := a ]=> (x !-> n ; st)
  | E_Seq : forall R c1 c2 st st' st'',
      R |-- st  =[ c1 ]=> st'  ->
      R |-- st' =[ c2 ]=> st'' ->
      R |-- st  =[ c1 ; c2 ]=> st''
  | E_IfTrue : forall R st st' b c1 c2,
      beval st b = true ->
      R |-- st =[ c1 ]=> st' ->
      R |-- st =[ if b then c1 else c2 end]=> st'
  | E_IfFalse : forall R st st' b c1 c2,
      beval st b = false ->
      R |-- st =[ c2 ]=> st' ->
      R |-- st =[ if b then c1 else c2 end]=> st'

  (* This is the only new one *)
  | E_Call : forall R st st' r c,
      R r = Some c ->
      R |-- st =[ c ]=> st' ->
      R |-- st =[ call r ]=> st'

  where "R |-- st =[ c ]=> st'" := (ceval R c st st').
```

$\boxed{4.1}$ According to the definition above, what happens if we execute the top-level command `call A` in the `empty` code context? (One or two sentences.)

Answer: Nothing! The program gets stuck and produces no result (just like an infinite loop). Formally, there are no `st` and `st'` such that `empty |-- st =[call A]=> st'`.

Program Equivalence

The definition of *program equivalence* extends smoothly from plain Imp to Imp with Subroutines.

Definition: A program `(R1,c1)` is *equivalent to* `(R2,c2)` if, for any pair of states `st` and `st'`, we have `R1 |-- st =[ c1 ]=> st'` iff `R2 |-- st =[ c2 ]=> st'`.

Now, consider the following code context:

```
Definition R : code_context :=
  (
   A |-> <{ X := 1; Y := 2 }>;
   B |-> <{ Y := 2; call C }>;
   C |-> <{ X := 1 }>;

   D |-> <{ X := X - 1; call D }>;
   E |-> <{ X := 0 }>;

   F |-> <{ if X > 0 then X := X - 1; call F else skip end }>;
   G |-> <{ X := 0 }>;

   H |-> <{ if X > 0 then X := X - 1; call H; X := X + 1 else skip end }>;
   I |-> <{ skip }>;

   J |-> <{ if X > 0 then X := X - 1; call J; X := X + 1 else skip end }>;
   K |-> <{ if X > 0 then X := X - 1; X := X + 1; call K else skip end }>
  ).
```

For each of the following pairs of programs, mark the appropriate box to indicate whether they are equivalent or inequivalent. If you choose "inequivalent," provide an example of a starting state on which they behave differently.

4.2 `(R, call A)` and `(R, call B)`

[✓ equivalent] ☐ inequivalent on `st =`

4.3 `(R, call D)` and `(R, call E)`

☐ equivalent [✓ inequivalent on `st = empty`]

4.4 `(R, call F)` and `(R, call G)`

[✓ equivalent] ☐ inequivalent on `st =`

4.5 `(R, call H)` and `(R, call I)`

[✓ equivalent] ☐ inequivalent on `st =`

4.6 `(R, call J)` and `(R, call K)`

☐ equivalent [✓ inequivalent on `st = X |-> 5`]

Hoare Logic with Subroutines

Finallt, we can extend the usual notion of Hoare triples to include a code context:

```
Definition valid_hoare_quad
           (R : code_context) (P : Assertion) (c : com) (Q : Assertion) :
  Prop := forall st st', R |-- st =[ c ]=> st' -> P st -> Q st'.

Notation "R |-- {{ P }} c {{ Q }}" :=
  (valid_hoare_quad R P c Q) (at level 40, c custom com at level 99, R
    constr, P at level 99, Q at level 99) : hoare_spec_scope.
```

All the existing Hoare Logic rules generalize to the new setting just by adding `R |--` to each Hoare triple.

Now, what rule should we introduce for the new `call r` command? One natural possibility is to simply to look up the command associated with `r` and say that any pre- and post-conditions we can establish for that command will also hold for `call r`:

```
R r = Some c      R |-- {{ P }} c {{ Q }}
-----------------------------------------
     R |-- {{ P }} call r {{ Q }}
```

$\boxed{4.7}$ Is this rule *unsound*? I.e., are there any invalid triples that we can prove with this rule? Briefly (one sentence) explain why or why not.

Answer: The rule is sound (anything we can prove using the rule will be valid according to the definition of Hoare quads): the behavior of `call r` *is exactly the behavior of the definition of* `r` *in the code context, so they satisfy the same Hoare triples.*

$\boxed{4.8}$ Is this rule *incomplete*? I.e., are there any valid triples that we cannot prove with this rule? Briefly (one sentence) explain why or why not.

Answer: The rule is incomplete: we cannot use it to prove anything about programs that call themselves recursively (either directly or indirectly), nor can we prove anything about programs that get stuck as a result of calling a subroutine that does not exist in the code context..

5 Hoare Logic (10 points)

For this problem, we return to the standard Imp and Hoare triple definitions, with no subroutines.

Suppose we are given a command `c` and a desired postcondition `Q`. In general, there may be many preconditions `P` that make the Hoare triple `{{P}} c {{Q}}` valid. But it is a property of Hoare logic that, among all these, there will be one such `P` that is weaker than all the others—i.e., such that `P' ->> P` whenever `{{P'}} c {{Q}}` is valid.

For example, these are all valid triples

```
{{ False }} X := Y {{ X = 1 }}
{{ X = 1 /\ Y = 1 }} X := Y {{ X = 1 }}
{{ Y = 1 }} X := Y {{ X = 1 }}
```

but `Y = 1` is the weakest precondition for this command and postcondition.

Select the weakest precondition `P` for each of the following triples. If the weakest precondition is not listed, then select "Some other precondition."

5.1 `{{ P }} Z := X; X := Y; Y := Z {{ Z = X }}`

- ☐ `True`
- ☐ `False`
- ☐ `Z = Y`
- ☐ `Z = X`
- [✓ `X = Y`]
- ☐ Some other precondition

5.2 `{{ P }} while X = 1 do Y := Y + 1 end {{ False }}`

- ☐ `True`
- ☐ `False`
- [✓ `X = 1`]
- ☐ `X <> 1`
- ☐ Some other precondition

5.3 `{{ P }} if X <> Y then X := Y else skip {{ X <> Y }}`

- ☐ `True`
- [✓ `False`]
- ☐ `X = Y`
- ☐ `X <> Y`
- ☐ Some other precondition

$\boxed{5.4}$ `{{ P }} while X < Y do X := X + 1 end {{ X <> Y }}`

- ☐ `True`
- ☐ `False`
- ☐ `X <> Y`
- ☐ `X >= Y`
- ☒ Some other precondition

  Precondition: `X > Y`

$\boxed{5.5}$

```
{{ P }}
  X := 0;
  while X < Y do
    X := X + 1;
    if X = m then X := 0 else skip
  end
{{ X = Y }}
```

- ☒ `True`
- ☒ `False`
- ☐ `Y > m`
- ☐ `Y = 0`
- ☐ `Y = m`
- ☐ Some other precondition

$\boxed{6}$ **STLC with Error** (16 points)

In this problem, we will take the simply typed lambda-calculus (with `unit` and no other extensions, see page 2 of the appendix) and add a simple form of exceptions. In particular, we add to the definition of terms a new constructor

```
  | error
```

and we propagate `error`s throughout a term by adding two new rules for applications to the small-step operational semantics:

```
                    ------------------                    (ST_Error1)
                    error t2 --> error

                         value v1
                    ------------------                    (ST_Error2)
                    v1 error --> error
```

Note that, even though `error` is a normal form, we do not define it to be a value (we will explore why shortly). Instead, we modify `progress` to allow normal forms to either be a value or `error`.

Instructions for the first three subproblems: For each starting term `t` below, give the term `t'` such that `t -->* t'` and `t'` is a normal form. Select which (single-)step rules are used to step to `t'`, if any. For example, `((\x:Unit, x) unit) unit` multi-steps to `unit unit` via rules `ST_App1` and `ST_AppAbs`.

$\boxed{6.1}$ The term `t =`

```
    (error error) error
```

multi-steps to `t' =`

`error`

...via rules

- [✓] `ST_App1`
- ☐ `ST_App2`
- ☐ `ST_AppAbs`
- [✓] `ST_Error1`
- ☐ `ST_Error2`
- ☐ none (`t` itself is a normal form)

$\boxed{6.2}$ The term `t =`

```
    (\x:Unit, x x)  error
```

multi-steps to `t' =`

`error`

...via rules

- ☐ `ST_App1`
- ☐ `ST_App2`
- ☐ `ST_AppAbs`
- ☐ `ST_Error1`
- ☒ `ST_Error2`
- ☐ none (`t` itself is a normal form)

$\boxed{6.3}$ The term `t =`

```
    (\x:Unit, (unit unit) (x error))  unit
```

multi-steps to `t' =`

`(unit unit) (unit error)`

...via rules

- ☐ `ST_App1`
- ☐ `ST_App2`
- ☒ `ST_AppAbs`
- ☐ `ST_Error1`
- ☐ `ST_Error2`
- ☐ none (`t` itself is a normal form)

$\boxed{6.4}$ If we incorrectly defined `error` to be a value, we would break determinism.

To demonstrate this, provide a term `t` and two distinct terms `t1` and `t2` such that if `value error` held, then we would have `t --> t1` and `t --> t2`.

`t = (\x:Unit, unit) error`

`t1 = unit`

`t2 = error`

Next, we add to the typing relation the rule

```
                    --------------------                        (T_Error)
                    Gamma |- error \in T
```

which says that `error` can have any type.

Instructions for the next three subproblems: For each term `t`, select all types `T` such that `t` can have type `T` under the empty context, or select "none of the above" if appropriate.

$\boxed{6.5}$ The term `t =`

```
error error
```

can have type(s)

- ☒ `Unit`
- ☒ `Unit -> Unit`
- ☒ `Unit -> (Unit -> Unit)`
- ☒ `(Unit -> Unit) -> Unit`
- ☐ none of the above

$\boxed{6.6}$ The term `t =`

```
\x:Unit, error
```

can have type(s)

- ☐ `Unit`
- ☒ `Unit -> Unit`
- ☒ `Unit -> (Unit -> Unit)`
- ☐ `(Unit -> Unit) -> Unit`
- ☐ none of the above

$\boxed{6.7}$ The term `t =`

```
error (\x:Unit, x x)
```

can have type(s)

- ☐ `Unit`
- ☐ `Unit -> Unit`
- ☐ `Unit -> (Unit -> Unit)`
- ☐ `(Unit -> Unit) -> Unit`
- ☒ none of the above

$\boxed{6.8}$ If we had incorrectly written the typing rule for `error` so that `error` only has type `Unit`, i.e.

```
                        -----------------------                (T_Error)
                        Gamma |- error \in Unit
```

we would break preservation.

To construct a counterexample, provide a term `f` such that

```
  (\x:Unit->Unit, x) (f error)
```

has type `Unit -> Unit` but the entire term steps to something ill-typed.

```
f = (\x:Unit, (\y:Unit, y))
```

$\boxed{7}$ **Subtyping** (14 points)

The setting for this problem is the simply typed lambda-calculus with booleans, products, and subtyping (see pages 1 to 5 of the appendix).

$\boxed{7.1}$ In this language, is there a type with infinitely many subtypes (i.e., is there some type `T` such that the set of all `S` with `S <: T` is infinite?)

[✓ Yes] ☐ No

If yes, give an example:

Example: `Top`

$\boxed{7.2}$ In this language, is there a type with infinitely many supertypes (i.e., is there some type `S` such that the set of all `T` with `S <: T` is infinite?)

[✓ Yes] ☐ No

If yes, give an example:

Example: `Top -> Top`

$\boxed{7.3}$ Suppose `t = (\x:Top * Bool, x.snd)`. Check *all* the types `T` such that `|-- t \in T`.

Select "Some other type(s)," even if you have already selected some options above it, if the term has more types than what are listed. Select "Not typeable" if none of the choices apply.

- [✓ `(Top * Bool) -> Top`]
- ☐ `Top -> Bool`
- [✓ `((Bool * Bool) * Bool) -> Top`]
- ☐ `(Top * Top) -> Top`
- [✓ Some other type(s)]
- ☐ Not typeable

$\boxed{7.4}$ Suppose `t = (\x:Bool->Top, true)`. Check *all* the types `T` such that `|-- t \in T`.

Select "Some other type(s)," even if you have already selected some options above it, if the term has more types than what are listed. Select "Not typeable" if none of the choices apply.

- [✓ `(Bool -> Top) -> Bool`]
- ☐ `Top -> Top`
- [✓ `(Bool -> (Top -> Bool)) -> Top`]
- [✓ `(Top -> (Bool -> Top)) -> Top`]
- [✓ Some other type(s)]
- ☐ Not typeable

$\boxed{7.5}$ Let $\mathcal{S}$ stand for the set of types `T` such that `empty |-- \x:Bool->Bool, x \in T`. What is the smallest element of $\mathcal{S}$ (i.e., which element of S is a subtype of all the others)?

- ☐ `Top`
- ☐ `Top -> Top`
- ☐ `(Top -> Bool) -> Bool -> Bool`
- ☒ `(Bool -> Bool) -> Bool -> Bool`
- ☐ `(Bool -> Top) -> Bool -> Bool`
- ☐ `Top -> Bool -> Bool`
- ☐ `Top -> Bool -> Bool`
- ☐ Some other type is the smallest one in S
- ☐ S has no smallest element

$\boxed{7.6}$ Now let $\mathcal{S}$ stand for the set of types `T` such that `empty |-- \x:T, x \in T->T`. What is the smallest element of $\mathcal{S}$?

- ☐ `Top`
- ☐ `Bool`
- ☐ `Top -> Top`
- ☐ `Top -> Bool`
- ☐ `Bool -> Top`
- ☐ Some other type is the smallest one in S
- ☒ S has no smallest element

$\boxed{7.7}$ What is the *largest* element of $\mathcal{S}$?

- ☒ `Top`
- ☐ `Bool`
- ☐ `Top -> Top`
- ☐ `Top -> Bool`
- ☐ `Bool -> Top`
- ☐ Some other type is the largest one in $\mathcal{S}$
- ☐ $\mathcal{S}$ has no largest element.

$\boxed{8}$ **Progress, Preservation, and Determinism for STLC with Subtyping** (15 points)

(The syntax, operational semantics, and typing rules for the simply-typed lambda calculus with booleans, products, and subtyping can be found on pages 1 to 5 in the appendix.)

For each variant below, indicate which properties of the original system remain true or become false in the presence of this rule. (The definitions of the properties are on page 8 in the appendix.)

$\boxed{8.1}$ Suppose that we add the following reduction rule:

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

- Progress ☒ Remains true ☐ Becomes false
- Preservation ☒ Remains true ☐ Becomes false
- Determinism ☐ Remains true ☒ Becomes false

  Counterexample: `((\x:Bool, \y:Bool, (x, y)) true) ((\x:Bool, x) true)` *can now reduce in one step to either*

  `(\y:Bool, (true, y)) ((\x:Bool, x) true)` *or*

  `((\x:Bool, \y:Bool, (x, y)) true) true`.

$\boxed{8.2}$ Suppose instead that we add the following reduction rule:

```
-------------------------------------  (Funny_If)
(if false then t1 else t2) --> false
```

- Progress ☒ Remains true ☐ Becomes false
- Preservation ☐ Remains true ☒ Becomes false

  Counterexample: `if false then (false, false) else (true, true)` *has type* `Bool*Bool` *but can step to* `false` *which has type* `Bool`.

- Determinism ☐ Remains true ☒ Becomes false

  Counterexample: `if false then false else true` *can now reduce in one step to either* `true` *or* `false`.

$\boxed{8.3}$ Suppose instead that we add the following typing rule:

```
Gamma |-- t \in T1 -> T2
------------------------    (Funny_Lambda_Type)
Gamma |-- t \in Top -> T2
```

- Progress [✓ Remains true] Becomes false
- Preservation Remains true [✓ Becomes false]

  Counterexample: `(\x:(Bool*Bool), x) true` *has type* `Bool*Bool` but steps to `true` *which has type* `bool`.
- Determinism [✓ Remains true] Becomes false

$\boxed{8.4}$ Suppose instead that we add the following typing rule:

```
-----------------------  (Funny_Prod_Arrow)
Top * Top <: Top -> Top
```

- Progress Remains true [✓ Becomes false]

  Counterexample: `(true,true) true` *has type* `Top` *but is not a value and can't take a step.*
- Preservation [✓ Remains true] Becomes false
- Determinism [✓ Remains true] Becomes false

$\boxed{8.5}$ Suppose instead that we add the following subtyping rule:

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

- Progress Remains true [✓ Becomes false]

  Counterexample: By transitivity, every type is now a subtype of every other type. For example, `Bool*Bool <: Bool`. *This means that* `if (true, true) then true else false` *has type* `Bool` *but is not a value and can't take a step.*
- Preservation Remains true [✓ Becomes false]

  Counterexample: `(\x:(Bool*Bool), x) true` *has type* `Bool*Bool` but steps to `true` *which has type* `Bool`.
- Determinism [✓ Remains true] Becomes false

$\boxed{9}$ **[Advanced Track Only] Informal Proof** (14 points)

The simply typed lambda-calculus with booleans, products, and subtyping is summarized on pages 1 to 5 of the appendix.

$\boxed{9.1}$ The subtype relation in this language has the following structural property:

Lemma (TTop): If `Top <: U`, then `U = Top`.

Give a careful informal proof of this Lemma. If your proof uses induction, make sure to state the induction hypothesis explicitly.

To prove this lemma carefully, we need to reword it a bit: `V <: U -> V = Top -> U = Top`*.*

We then prove `V = Top -> U = Top` *by induction on a derivation of* `V <: U`*.*

Inspecting the subtyping rules, we see that `Top` *can only appear on the left-hand of rules* `S_Refl`*,* `S_Top`*, and* `S_Trans`*.*

If the final rule is `S_Refl`*, then* `U = V` *and* `U = Top` *is immediate.*

If the final rule is `S_Top`*, then* `U = Top` *by the form of the rule.*

Lastly, suppose the final rule is `S_Trans`*—that is, suppose we have* `V <: T` *and* `T <: U`*. The IH for the first subderivation says that if* `V = Top` *then* `T = Top`*. The IH for the second subderivation says that if* `T = Top` *then* `U = Top`*. Putting these together with the assumption that* `V = Top`*, we obtain* `U = Top`*, as required.*

(Nobody actually wrote their solution this way, but we graded according to how closely the solutions approximated the reasoning underlying this one.)

$\boxed{9.2}$ Similarly, we have:

Lemma (TArrow):
If `S1 -> S2 <: U` then `U = Top` or `U` has the form `U1 -> U2` for some `U1` and `U2`.

and:

Lemma (TPair):
If `S1 * S2 <: U` then `U = Top` or `U` has the form `U1 * U2` for some `U1` and `U2`.

(You do not need to prove these lemmas.)

It follows that, if `U` is a supertype of *both* an arrow type and a pair type, then `U = Top`.

Theorem: If `S1 -> S2 <: U` and `T1 * T2 <: U`, then `U = Top`.

Give a careful informal proof of this theorem. Your proof may, if you like, use the `TTop`, `TArrow`, and/or `TPair` lemmas. If it uses induction, make sure to state the induction hypothesis explicitly.

By `TArrow`*,* `U` *must be either* `Top` *or an arrow type. By* `TPair` *it must be either* `Top` *or a pair type. Hence it can only be* `Top`*.*

$\boxed{10}$ **References** (12 points) The simply typed lambda-calculus with references is summarized on pages 6 to 7 of the appendix.

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

```
Theorem preservation_theorem_with_references := 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.

By contrast, the preservation theorem for the plain STLC without references looks quite a bit simpler:

```
Theorem preservation : forall t t' T,
  empty |-- t \in T  ->
  t --> t'  ->
  empty |-- t' \in T.
```

Briefly identify the differences between the two versions of the theorem, and explain why they are needed.

Answer: The main differences are:

- Store typings are added to both the hypothesis (`ST`) and the conclusion (`ST'`), reflecting the fact that a term containing locations must be typed in a context that records our assumptions about what can be in those locations.
- The starting store `st` is assumed to satisfy the assumptions in `ST`.
- After a step of execution, the final store `st'` will satisfy the store typing `ST'`, where `ST'` differs from `ST` only in potentially adding a type for one new location.

  Concretely, 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). The existential quantifier in the extended 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.

Grading scheme: In retrospect, the way this question was phrased made it a bit hard to understand what would count as a "complete enough" answer. We therefore graded fairly generously. To receive most of the points, an answer had to at least mention the fact that the existential quantifier is needed to account for new store locations allocated by the ST_RefValue rule (otherwise the store and its store typing would get desynchronized). Beyond this, additional points were given for discussing other differences between the rules.

For Reference

Simply Typed Lambda Calculus

Syntax and rules for STLC with no extensions. (Base types will be added later.)

```
Syntax:

   T ::= T -> T                        arrow type

   t ::= x                             variable
       | \x:T,t                        abstraction
       | t t                           application

Values:

   v ::= \x:T,t

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)

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'

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
```

STLC + Unit

```
Syntax:

   T ::= ...
       | Unit                     unit type

   t ::= ...
       | unit                     unit value

Values:

   v ::= ...
       | unit

Substitution:

   ...
   [x:=s]unit           = unit

Small-step operational semantics:

(no new rules)

Typing:

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

STLC + Booleans + Products + Subtyping

Booleans

```
Syntax:

   T ::= ...
       | Bool                      boolean type

   t ::= ...
       | true                      true
       | false                     false
       | if t then t else t        conditional

Values:

   v ::= ...
       | true
       | false

Substitution:

   ...
   [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

Small-step operational semantics:

                --------------------------------              (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:

                     ---------------------                      (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
```

Products

```
Syntax:

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

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

Values:

   v ::= ...
       | (v,v)

Substitution:

   ...
   [x:=s](t1, t2)        = ([x:=s] t1, [x:=s] t2)
   [x:=s]t.fst           = ([x:=s] t).fst
   [x:=s]t.snd           = ([x:=s] t).snd

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

Syntax:

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

STLC + References

(Based on the STLC with 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
```

Properties

```
Definition deterministic {X : Type} (R : relation X) :=
  forall x y1 y2 : X, R x y1 -> R x y2 -> y1 = y2.

Theorem step_deterministic:
  deterministic step.

Theorem progress : forall t T,
  empty |-- t \in T ->
  value t \/ exists t', t --> t'.

Theorem preservation : forall t t' T,
  empty |-- t \in T  ->
  t --> t'  ->
  empty |-- t' \in T.
```