

Name or WPE-I number (please print clearly!):

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

If you are taking the exam as a WPE-I, you should follow the instructions for the Advanced track.

If you need extra space for any of your answers, you can use the back of the exam. However, if you do, make sure to put a clear note on the *front* of the exam indicating that we should look at the back!

1 (12 points) (Functional programming in Coq) For each type below, give either a term which inhabits that type, or write “uninhabited.”

1.1 forall (X Y : Type), Y -> X

1.2 forall (A B : Type), Option (A -> B) -> Option A -> B

1.3 forall (A B : Type), Option A -> Option B

1.4 forall (A B C : Type), (A * B -> C) -> A -> B -> C

1.5 forall (A B C : Type), ((A -> B) -> C) -> (A * B -> C)

2 (11 points) (Functional programming in Coq) We've seen the `fold` function over lists:

```
Fixpoint fold_list {A B : Type} (f : A -> B -> B) (b : B) (l : list A) : B :=
  match l with
  | [] => b
  | h :: t => f h (fold_list f b t)
  end.
```

The same idea can be instantiated for a wide variety of inductive data types. For instance, consider the type of binary trees (annotated with data items at their internal nodes).

```
Inductive bin (X : Type) :=
  | Node : bin X -> X -> bin X -> bin X
  | Leaf : bin X
```

2.1 (6 points) A “fold function” for this type would have the following type:

```
forall A B, (B -> A -> B -> B) -> B -> bin A -> B
```

Implement `fold_bin`. Here is the header:

```
Fixpoint fold_bin {A B : Type} (f : B -> A -> B -> B) (b : B) (t : bin A) : B :=
```

Here is type of `fold_bin` again:

```
forall (A B : Type), (B -> A -> B -> B) -> B -> bin A -> B
```

Now consider a hypothetical inductive type of *truffula trees*. A truffula tree has two kinds of internal nodes: **Q** nodes with one data element and two children, and **P** nodes with two data elements and three children. Its leaves are also of two varieties: they can each be a **R** with four data elements or a plain **S** leaf with no payload.

```
Inductive truffula (X : Type) :=
  | P : X -> X -> truffula X -> truffula X -> truffula X -> truffula X
  | Q : X -> truffula X -> truffula X -> truffula X
  | R : X -> X -> X -> X -> truffula X
  | S : truffula X
```

2.2 (5 points) What should be the type of a fold function over truffula trees?

```
fold_truffula :
```

3 (15 points) (Program equivalence in Imp) For each part, circle *True* or *False*. Some parts also ask for examples, counterexamples, or explanations.

3.1 (1 point) The following programs are equivalent.

```
Y ::= X * 2
```

and

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

True *False*

3.2 (1 point) The following programs are equivalent.

```
X ::= 0
```

and

```
IF X = 0 THEN  
  SKIP  
ELSE  
  WHILE True DO SKIP DONE  
FI
```

True *False*

3.3 (3 points) The following programs are equivalent for *all* choices of subcommand *c*. If you choose *False*, give a counterexample (a *c* that shows that they are not equivalent).

```
WHILE True DO c DONE      and      c;; WHILE True DO c DONE
```

True *False*

3.4 (3 points) The following programs are equivalent for *some* choice of subcommand *c*. If you choose *True*, give an example of such a *c*.

`c ;; c` and `c`
True *False*

3.5 (3 points) The following programs are equivalent for *some* choice of subcommand *c*. If you choose *True*, give an example of such a *c*.

`WHILE True DO c DONE` and `c`
True *False*

3.6 (4 points) For all boolean expressions *b* and for all subcommands *c*, there exists some subcommand *d* such that the following programs are equivalent. Briefly explain your answer.

`WHILE b DO`
`c;;` and `IF b THEN`
`d` `WHILE True DO SKIP DONE`
`DONE` `ELSE`
 `SKIP`
 `FI`

True *False*

4 [Standard Only] (4 points) (Hoare logic) Circle *True* or *False*.

4.1 The Hoare rule $\{\{ True \} X ::= 2 \{ X = 2 \}$ can be proven directly using just the `hoare_asgn` rule.

True *False*

4.2 Everything that can be proven using the following rule can also be proven using the `hoare_if` rule.

$$\frac{\begin{array}{l} \{ P \} c1 \{ Q \} \\ \{ P \} c2 \{ Q \} \end{array}}{\{ P \} \text{IFB } b \text{ THEN } c1 \text{ ELSE } c2 \text{ FI } \{ Q \}} \quad ()$$

True *False*

5 [Standard Only] (14 points) (Hoare logic)

5.1 WHILE X > 0 DO
 X ::= X - 1;;
 IF X = Y THEN X ::= X + 1
END

Check the box next to each assertion that is an *invariant* of this loop.

- True
- X > 0
- X = Y + 1
- X > Y + 1
- X < Y + 1
- False

5.2 R ::= 0;;
X ::= 0;;
WHILE X < Y DO
 R ::= R + Y;;
 X ::= X + 1
END

Check all assertions that are valid *postconditions* of this program.

- True
- R = X + Y
- R = X * Y
- R = 2 * Y
- R = Y * Y
- False

5.3

```
R ::= 0;;  
X ::= 0;;  
WHILE X < Y DO  
  R ::= R + Z;;  
  X ::= X + 1  
END
```

Check assertions that are valid *postconditions* of this program.

- $R = Y + Z$
- $R = Y * Z$

Check assertions that are *invariants* of this loop and are *sufficient* to prove the postcondition(s) you checked.

- True
- $R = Y * (Z-X)$
- $X \leq Y \wedge R = Y * (Z-X)$
- $R = X * Z$
- $X \leq Y \wedge R = X * Z$
- False

6 [Advanced Only] (16 points) (Hoare logic) Recall the Hoare logic rule for WHILE loops:

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

Write a careful informal proof of the correctness of this rule.

7 (14 points) (Operational semantics) In this problem, we consider an alternate formulation of the small-step operational semantics for the simply-typed lambda calculus with booleans (without subtyping and no products until part (d)).

One annoying thing about the operational semantics is the number of “structural” rules (`ST_App1`, `ST_App2`, `ST_If`) that we have to deal with. An alternate formulation of the operational semantics is to give a syntax of “evaluation contexts” `E` (of type `ectx`) and “primitive steps” `s` (which are just particular terms—those that are “ready to take a top-level step”) like this:

```
(* Evaluation contexts E : ectx *)          (* prim_step : tm -> Prop *)
E ::= []          (* hole *)          s ::= (\x:T.t) v
     | E t        (* ST_App1 *)       | if true then t1 else t2
     | v E        (* ST_App2 *)       | if false then t1 else t2
     | if E then t1 else t2 (* ST_If *)
```

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

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

(Again, this is “Coq pseudocode”: the syntax of the patterns is informal.)

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

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

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

There are no questions on this page.

7.1 (3 pts.) If we use these evaluation contexts to prove soundness, we need a couple of different helper lemmas.

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

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

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

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

7.2 (4 pts.) We also need a kind of substitution lemma that relates to `fill`. A bad attempt at stating it might be something like this:

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

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

7.3 (3 pts.) A better way to state the substitution principal is:

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

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

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

7.4 (4 pts.) What would we need to add to the definition of **E** to support products? (The usual rules for products are given in the handout. You may need to add more than one clause.)

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

8 (15 points) (Simply typed lambda-calculus) In this problem we will develop a variant of the simply-typed lambda calculus with natural numbers and an induction operator. The starting point is the plain simply typed lambda with a base type of natural numbers and constructors for the constants 0 and successor `S`. You can find the syntax, typing rules, and small-step evaluation rules for this part of the language in the handout. Note that, for this problem, we do not consider subtyping, `fix`, or any other extensions to the STLC.

Rather than adding `if0` and the general recursion operator `fix` for writing programs over natural numbers, this extension follows Coq and adds a built-in form of natural-number induction:

```
t ::= ...
| nat_ind t t t
```

The term `nat_ind tz ts tn` acts like a `fold` over the natural number datatype. The term `tz` specifies what to do for the base (zero) case of the induction, and the term `ts` (successor) shows how to compute the answer for `S n` given `n` itself and the inductive result for `n`. The argument `tn` is the natural number over which induction is being done. Once we have added `nat_ind` to the STLC, we can write many familiar programs using natural numbers. For example, here is a function that adds two natural numbers, defined by induction on `n`. The base case is just `m` and the inductive step computes the successor of the recursive result:

```
Nat_plus = \n:Nat. \m:Nat. nat_ind m (\x:Nat.\y:Nat. S y) n
```

The steps it takes when computing `Nat_plus 2 1` look like this, where we have marked the novel behavior of `nat_ind` with `!!` and where we write 2 as a shorthand for `S (S 0)`, etc.:

```
(\n:Nat. \m:Nat. nat_ind m (\x:Nat.\y:Nat. S y) n) 2 1
==>
(\m:Nat. nat_ind m (\x:Nat.\y:Nat. S y) 2) 1
==>
nat_ind 1 (\x:Nat.\y:Nat. S y) 2
==> !! (nonzero case)
(\x:Nat.\y:Nat. S y) 1 (nat_ind 1 (\x:Nat.\y:Nat. S y) 1)
==> !! (nonzero case)
(\x:Nat.\y:Nat. S y) 1 ((\x:Nat.\y:Nat. S y) 0 (nat_ind 1 (\x:Nat.\y:Nat. S y) 0))
==> !! (zero case)
(\x:Nat.\y:Nat. S y) 1 ((\x:Nat.\y:Nat. S y) 0 1)
==>
(\x:Nat.\y:Nat. S y) 1 ((\y:Nat. S y) 1)
==>
(\x:Nat.\y:Nat. S y) 1 2
==>
(\y:Nat. S y) 2
==>
3
```

Intuitively, the small-step operational semantics of `nat_ind` should work like this:

```
nat_ind vz vs 3 ==>* vs 2 (vs 1 (vs 0 vz))
```

- 8.1** (5 points) First, let's complete the small-step operational semantics for `nat_ind`. There are three congruence rules that evaluate the arguments to `nat_ind` in order from left-to-right. The first is:

$$\frac{tz ==> tz'}{\text{nat_ind } tz \text{ } ts \text{ } tn ==> \text{nat_ind } tz' \text{ } ts \text{ } tn}$$

Write the other two structural rules below. Use the `value` predicate as appropriate.

After reducing all three arguments to values, the “interesting” rules of the small step semantics do case analysis on the third argument, yielding the base case, or performing a recursive call as appropriate. Complete these two rules for the small-step operational semantics of `nat_ind`.

8.2 (4 points) It remains to give a typing rule for `nat_ind`. We know that the third argument to `nat_ind` is supposed to be a `Nat`, so that part is easy. The result type of a `nat_ind` expression can be any type `T`, since we could conceivably construct any value by induction on a natural number. We have filled in those parts of the typing rule below. Your job is to complete the typing rule. Consider that this rule should be sound (i.e. satisfy preservation and progress) with respect to the operational semantics outlined above. For example, the term `Nat_plus` defined above should be well-typed according to your rule.

```
Gamma |- tz : T
-----
Gamma |- nat_ind tz ts tn : T
```

8.3 (4 points) Above, we used `nat_ind` to define the `Nat_plus` function. Now, use `Nat_plus` and `nat_ind` to define *multiplication* of two numbers.

```
Nat_mult : Nat -> Nat -> Nat
Nat_mult =
```


9 (25 points) (Simply typed lambda-calculus with subtyping) The syntax, operational semantics, and typing rules for the simply-typed lambda calculus with subtyping and booleans are given in the handout.

For each variant of this system described below, indicate which of the properties remain true in the presence of this rule. For each one, circle either “*Remains true*” or else “*Becomes false.*” If a property becomes false, give a counterexample.

9.1 Consider a variant in which we add the following new subtyping rule:

$$\frac{}{\text{Top} * \text{Top} <: \text{Top} \rightarrow \text{Top}} \text{S_ProdArrow}$$

- Determinism of \Rightarrow

Remains true *Becomes false* (*counterexample...*)

- Progress

Remains true *Becomes false* (*counterexample...*)

- Preservation

Remains true *Becomes false* (*counterexample...*)

9.2 Consider a variant in which we add a new term `loop` with the following reduction and typing rules:

$$\begin{array}{ll}
 \text{----- (ST_Loop)} & \text{----- (T_Loop)} \\
 \text{loop} \Rightarrow \text{loop} & \Gamma \vdash \text{loop} \in \mathbb{T}
 \end{array}$$

- Determinism of \Rightarrow

Remains true *Becomes false (counterexample...)*

- Progress

Remains true *Becomes false (counterexample...)*

- Preservation

Remains true *Becomes false (counterexample...)*

9.3 Suppose instead that we add the following typing rule:

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

- Determinism of \Rightarrow

Remains true *Becomes false* (*counterexample...*)

- Progress

Remains true *Becomes false* (*counterexample...*)

- Preservation

Remains true *Becomes false* (*counterexample...*)

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

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

and the following typing rule:

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

- Determinism of \Rightarrow

Remains true *Becomes false* (*counterexample...*)

- Progress

Remains true *Becomes false* (*counterexample...*)

- Preservation

Remains true *Becomes false* (*counterexample...*)

9.5 Instead, suppose that we add a new reduction rule:

$$\frac{t \Rightarrow t'}{\lambda x:T. t \Rightarrow \lambda x:T. t'} \quad (\text{ST_LCong})$$

- Determinism of \Rightarrow

Remains true *Becomes false* (*counterexample...*)

- Progress

Remains true *Becomes false* (*counterexample...*)

- Preservation

Remains true *Becomes false* (*counterexample...*)

10 [Standard Only] (10 points) (Subtyping) For each of the following pairs of types, S and T , circle the appropriate description of how they are ordered by the subtype relation.

10.1 $S = \text{Top} \rightarrow \text{Top}$
 $T = (\text{Top} \rightarrow \text{Top}) \rightarrow \text{Top}$

- $S <: T$
- $T <: S$
- equivalent (both $S <: T$ and $T <: S$)
- unrelated (neither $S <: T$ nor $T <: S$)

10.2 $S = \{x: \text{Bool}, y: \text{Bool} \rightarrow \text{Top}\}$
 $T = \{x: \text{Top}, y: \text{Top} \rightarrow \text{Bool}\}$

- $S <: T$
- $T <: S$
- equivalent
- unrelated

10.3 $S = \{x: \text{Bool} \rightarrow \text{Top}\}$
 $T = \{y: \text{Nat}, x: \text{Top} \rightarrow \text{Top}\}$

- $S <: T$
- $T <: S$
- equivalent
- unrelated

10.4 $S = (\text{Bool} \rightarrow \text{Top}) \rightarrow (\text{Nat} \rightarrow \text{Nat})$
 $T = (\text{Top} \rightarrow \text{Bool}) \rightarrow \text{Top}$

- $S <: T$
- $T <: S$
- equivalent
- unrelated

10.5 $S = \{x: \text{Nat}, y: \text{Bool}\} \rightarrow \text{Nat}$
 $T = \{y: \text{Bool}, x: \text{Nat}\} \rightarrow \text{Nat}$

- $S <: T$
- $T <: S$
- equivalent
- unrelated

11 [Advanced Only] (12 points) (Subtyping) Give a (careful and detailed) *informal* proof of the following lemma about the simply typed lambda-calculus with subtyping (a slight variation on one we saw in the Subtyping chapter of *Software Foundations*):

Lemma: If $\Gamma \vdash s \in T_1 \rightarrow T_2$ and s is a value, then there exist x , S_1 , and s_2 such that:

- $s = \lambda x:S_1. s_2$
- $\Gamma, x:S_1 \vdash s_2 \in S_2$
- $T_1 <: S_1$
- $S_2 <: T_2$

For Reference

Formal definitions for Imp

Syntax

```
Inductive aexp : Type := | ANum : nat -> aexp | AId : id -> aexp |
APlus : aexp -> aexp -> aexp | AMinus : aexp -> aexp -> aexp | AMult :
aexp -> aexp -> aexp.
```

```
Inductive bexp : Type :=
| BTrue : bexp
| BFalse : bexp
| BEq : aexp -> aexp -> bexp
| BLe : aexp -> aexp -> bexp
| BNot : bexp -> bexp
| BAnd : bexp -> bexp -> bexp.
```

```
Inductive com : Type :=
| CSkip : com
| CAss : id -> aexp -> com
| CSeq : com -> com -> com
| CIf : bexp -> com -> com -> com
| CWhile : bexp -> com -> com.
```

```
Notation "'SKIP'" :=
  CSkip.
```

```
Notation "l '::=' a" :=
  (CAss l a) (at level 60).
```

```
Notation "c1 ;; c2" :=
  (CSeq c1 c2) (at level 80, right associativity).
```

```
Notation "'WHILE' b 'DO' c 'END'" :=
  (CWhile b c) (at level 80, right associativity).
```

```
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
  (CIf e1 e2 e3) (at level 80, right associativity).
```


Evaluation relation

```
Inductive ceval : com -> state -> state -> Prop :=
| E_Skip : forall st,
  SKIP / st \\ st
| E_Ass  : forall st a1 n X,
  aeval st a1 = n ->
  (X ::= a1) / st \\ (update st X n)
| E_Seq  : forall c1 c2 st st' st'',
  c1 / st \\ st' ->
  c2 / st' \\ st'' ->
  (c1 ;; c2) / st \\ st''
| E_IfTrue : forall st st' b1 c1 c2,
  beval st b1 = true ->
  c1 / st \\ st' ->
  (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
| E_IfFalse : forall st st' b1 c1 c2,
  beval st b1 = false ->
  c2 / st \\ st' ->
  (IFB b1 THEN c1 ELSE c2 FI) / st \\ st'
| E_WhileEnd : forall b1 st c1,
  beval st b1 = false ->
  (WHILE b1 DO c1 END) / st \\ st
| E_WhileLoop : forall st st' st'' b1 c1,
  beval st b1 = true ->
  c1 / st \\ st' ->
  (WHILE b1 DO c1 END) / st' \\ st'' ->
  (WHILE b1 DO c1 END) / st \\ st''
```

where "c1 '/' st '\\ st'" := (ceval c1 st st').

Program equivalence

```
Definition bequiv (b1 b2 : bexp) : Prop :=
  forall (st:state), beval st b1 = beval st b2.
```

```
Definition cequiv (c1 c2 : com) : Prop :=
  forall (st st' : state),
    (c1 / st \\ st') <-> (c2 / st \\ st').
```

Hoare triples

```
Definition hoare_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=
  forall st st', c / st \\ st' -> P st -> Q st'.
```

Notation "{ P } c { Q }" := (hoare_triple P c Q).

Implication on assertions

Definition `assert_implies (P Q : Assertion) : Prop := forall st, P st -> Q st.`

Notation "`P ->> Q`" := (`assert_implies P Q`) (at level 80).

(ASCII `->>` is typeset as a hollow arrow in the rules below.)

Hoare logic rules

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

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

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

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

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

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

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

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

Decorated programs

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

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

2. The sequential composition of c_1 and c_2 is locally consistent (with respect to assertions P and R) if c_1 is locally consistent (with respect to P and Q) and c_2 is locally consistent (with respect to Q and R):

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

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

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

4. A conditional is locally consistent (with respect to assertions P and Q) if the assertions at the top of its "then" and "else" branches are exactly $P \wedge b$ and $P \wedge \sim b$ and if its "then" branch is locally consistent (with respect to $P \wedge b$ and Q) and its "else" branch is locally consistent (with respect to $P \wedge \sim b$ and Q):

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

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

```

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

```

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

```

  {{ P }}  $\rightarrow$ 
  {{ P' }}

```

Small Step Semantics

Reserved Notation " t ' / ' st ' \Rightarrow_a ' t " (at level 40, st at level 39).

Inductive `astep` : `state` \rightarrow `aexp` \rightarrow `aexp` \rightarrow `Prop` :=

```

| AS_Id : forall st i,
  AId i / st  $\Rightarrow_a$  ANum (st i)
| AS_Plus : forall st n1 n2,
  APlus (ANum n1) (ANum n2) / st  $\Rightarrow_a$  ANum (n1 + n2)
| AS_Plus1 : forall st a1 a1' a2,
  a1 / st  $\Rightarrow_a$  a1'  $\rightarrow$ 
  (APlus a1 a2) / st  $\Rightarrow_a$  (APlus a1' a2)
| AS_Plus2 : forall st v1 a2 a2',
  aval v1  $\rightarrow$ 
  a2 / st  $\Rightarrow_a$  a2'  $\rightarrow$ 
  (APlus v1 a2) / st  $\Rightarrow_a$  (APlus v1 a2')
| AS_Minus : forall st n1 n2,
  (AMinus (ANum n1) (ANum n2)) / st  $\Rightarrow_a$  (ANum (minus n1 n2))
| AS_Minus1 : forall st a1 a1' a2,
  a1 / st  $\Rightarrow_a$  a1'  $\rightarrow$ 
  (AMinus a1 a2) / st  $\Rightarrow_a$  (AMinus a1' a2)
| AS_Minus2 : forall st v1 a2 a2',
  aval v1  $\rightarrow$ 
  a2 / st  $\Rightarrow_a$  a2'  $\rightarrow$ 
  (AMinus v1 a2) / st  $\Rightarrow_a$  (AMinus v1 a2')
| AS_Mult : forall st n1 n2,
  (AMult (ANum n1) (ANum n2)) / st  $\Rightarrow_a$  (ANum (mult n1 n2))
| AS_Mult1 : forall st a1 a1' a2,
  a1 / st  $\Rightarrow_a$  a1'  $\rightarrow$ 
  (AMult a1 a2) / st  $\Rightarrow_a$  (AMult a1' a2)

```

```

| AS_Mult2 : forall st v1 a2 a2',
  aval v1 ->
  a2 / st ==>a a2' ->
  (AMult v1 a2) / st ==>a (AMult v1 a2')
  where " t '/' st '==>a' t' " := (astep st t t').

```

Reserved Notation " t '/' st '==>' t' '/' st' "
 (at level 40, st at level 39, t' at level 39).

```

Inductive cstep : (com * state) -> (com * state) -> Prop :=
| CS_AssStep : forall st i a a',
  a / st ==>a a' ->
  (i ::= a) / st ==> (i ::= a') / st
| CS_Ass : forall st i n,
  (i ::= (ANum n)) / st ==> SKIP / (t_update st i n)
| CS_SeqStep : forall st c1 c1' st' c2,
  c1 / st ==> c1' / st' ->
  (c1 ;; c2) / st ==> (c1' ;; c2) / st'
| CS_SeqFinish : forall st c2,
  (SKIP ;; c2) / st ==> c2 / st
| CS_IfTrue : forall st c1 c2,
  IFB BTrue THEN c1 ELSE c2 FI / st ==> c1 / st
| CS_IfFalse : forall st c1 c2,
  IFB BFalse THEN c1 ELSE c2 FI / st ==> c2 / st
| CS_IfStep : forall st b b' c1 c2,
  b / st ==>b b' ->
  IFB b THEN c1 ELSE c2 FI / st
  ==> (IFB b' THEN c1 ELSE c2 FI) / st
| CS_While : forall st b c1,
  (WHILE b DO c1 END) / st
  ==> (IFB b THEN (c1;; (WHILE b DO c1 END)) ELSE SKIP FI) / st
  where " t '/' st '==>' t' '/' st' " := (cstep (t,st) (t',st')).

```

STLC with booleans

Syntax

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

Small-step operational semantics

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

Typing

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

Properties of STLC

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

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

STLC with products

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

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

Small-step operational semantics (added to STLC rules)

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

Typing (added to STLC rules)

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

Subtyping

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

$T ::= \dots$
| `Top`

Add these rules that characterize the subtyping relation:

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

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

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

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

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

Typing (added to STLC with products)

All of the ordinary typing rules, plus:

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