CIS 500: Software Foundations

Final Exam

December 15, 2017

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.” Please do not waste time (yours or ours) answering questions intended for the other track.

If you are taking the exam as to satisfy the WPE-I requirement for the CIS PhD program, follow the instructions for the Advanced track.

Before beginning, please write your name or WPE-I ID at the top of each even-numbered page.
1. **Inductive relations** (11 points)

Complete the definition at the bottom of the page of an *Inductive* relation `count` that relates a list to the number of elements of a list satisfying some predicate \( P \) — that is, `count P l n` should be true when the number of elements of `l` satisfying \( P \) is \( n \). For example, if we define

\[
\text{Definition iszero } (n : \text{nat}) : \text{Prop} := (n = 0).
\]

then the propositions

\[
\begin{align*}
\text{count iszero } [] &\text{ 0} \\
\text{count iszero } [1,2,3] &\text{ 0} \\
\text{count iszero } [0,1,2,3] &\text{ 1} \\
\text{count iszero } [1,0,0,2,3,0] &\text{ 3}
\end{align*}
\]

should all be provable, whereas the propositions

\[
\begin{align*}
\text{count iszero } [1,2,3] &\text{ 3} \\
\text{count iszero } [0,0] &\text{ 4} \\
\text{count iszero } [] &\text{ 1}
\end{align*}
\]

should not be provable.

\[
\text{Inductive count } (X : \text{Type}) (P : X \rightarrow \text{Prop}) : \text{list X} \rightarrow \text{nat} \rightarrow \text{Prop} :=
\]
2. **Termination in Coq** (12 points)

We know that every **Fixpoint** definition in Coq must pass a “termination checker,” to ensure that all definable functions terminate on all inputs.

2.1 For example, here is the definition of the function **app** which concatenates (appends) two lists:

```coq
Fixpoint app (l1 l2 : natlist) : natlist :=
  match l1 with
  | nil => l2
  | h :: t => h :: (app t l2)
  end.
```

In one or two sentences, explain how Coq is able to automatically deduce that this function always terminates.

2.2 Fill in a Fixpoint definition (it doesn’t have to make sense) that would not pass Coq’s termination checker.

```coq
Fixpoint nonsense (l1 l2 : natlist) : natlist :=
```

2.3 Although functions defined with **Fixpoint** must terminate in Coq, this isn’t true in Coq’s tactic language **Ltac**. Indeed, it is possible to write tactics that will loop forever.

Should we be worried about this, from the perspective of logical consistency? Briefly explain why or why not.
3. Program equivalence (12 points)

Consider the Imp language from chapter Imp.v of Software Foundations (summarized on pages ?? and ?? in the appendix). The definition of equivalence for Imp programs can be rephrased as follows:

- Programs $c_1$ and $c_2$ are called equivalent on state $st$ if, for every state $st'$, $c_1 / st \setminus st'$ if and only if $c_2 / st \setminus st'$.
- Programs $c_1$ and $c_2$ are equivalent if they are equivalent on every state.
- Programs $c_1$ and $c_2$ are called somewhat inequivalent if there is some state $st_1$ on which they are equivalent and another state $st_2$ on which they are not equivalent.
- Programs $c_1$ and $c_2$ are called very inequivalent if there is no state $st$ on which they are equivalent.

For each of the following pairs of programs $c_1$ and $c_2$, check the appropriate box to indicate whether they are equivalent, somewhat inequivalent, or very inequivalent. (Keep in mind that Imp variables hold natural numbers ($\geq 0$), not integers.) If you choose somewhat inequivalent, give an example of a starting state $st$ on which $c_1$ and $c_2$ behave differently.

3.1 $c_1 = \text{WHILE } \text{False DO } X ::= X + 1 \text{ END}$
$c_2 = \text{WHILE } X > 0 \text{ DO } X ::= X + 1 \text{ END}$

For example:

3.2 $c_1 = \text{WHILE } X > 0 \text{ DO } X ::= X + 1 \text{ END}$
$c_2 = \text{WHILE } X > 0 \text{ DO } Y ::= Y + 1 \text{ END}$

For example:

3.3 $c_1 = X ::= X + 1 ;; Y ::= X$
$c_2 = Y ::= X + 1$

For example:

3.4 $c_1 = \text{IFB } X = Y \text{ THEN } X ::= X - Y \text{ ELSE } X ::= 0 \text{ FI}$
$c_2 = \text{SKIP}$

For example:
\[ c_1 = \begin{cases} \text{IF} & X = Y \text{ THEN } X ::= X - Y \text{ ELSE } X ::= 0 \text{ FI} \\ \\
\end{cases} \]
\[ c_2 = \text{WHILE } X > 0 \text{ DO } X ::= X - 1 \text{ END} \]

\[ \begin{array}{ccc}
\text{□ Equivalent} & \text{□ Somewhat inequivalent} & \text{□ Very inequivalent} \\
\end{array} \]

For example:
4. [Standard Only] Hoare triples (12 points)

Each of the following Hoare triples contain a variable c, representing an arbitrary Imp program (that is, a com). For each triple, check the appropriate box to indicate whether the triple is always valid (valid for all choices of c), sometimes valid (valid for some, but not all, choices of c), or never valid (invalid for all choices of c). If you choose sometimes valid, give an example of a command c1 that makes the triple valid and a command c2 that makes the triple invalid.

4.1

\[
\{\{ X = 3 \}\} \\
\text{WHILE True DO c END}\ \\
\{\{ X = 3 \}\}
\]

\[\begin{array}{ccc}
\square \text{ Always valid } & \square \text{ Sometimes valid } & \square \text{ Never valid } \\
\end{array}\]

\[\begin{array}{l}
c1 = \\
c2 = \\
\end{array}\]

4.2

\[
\{\{ X = 1 \}\} \\
\text{WHILE } X > 0 \text{ DO c END }; \\
\text{WHILE } X = 0 \text{ DO c END}\ \\
\{\{ \text{False} \}\}
\]

\[\begin{array}{ccc}
\square \text{ Always valid } & \square \text{ Sometimes valid } & \square \text{ Never valid } \\
\end{array}\]

\[\begin{array}{l}
c1 = \\
c2 = \\
\end{array}\]

4.3

\[
\{\{ \text{True} \}\} \\
\text{WHILE } X > 0 \text{ DO c ;; Y ::= 1 END ;; } \\
\text{WHILE } X = 0 \text{ DO c ;; Y ::= 0 END}\ \\
\{\{ Y = 0 \}\}
\]

\[\begin{array}{ccc}
\square \text{ Always valid } & \square \text{ Sometimes valid } & \square \text{ Never valid } \\
\end{array}\]

\[\begin{array}{l}
c1 = \\
c2 = \\
\end{array}\]
\[ \{\{ Y = X \}\} \]

\[
\text{IFB } X = 0 \text{ THEN}
\]

\[
\text{SKIP}
\]

\[
\text{ELSE}
\]

\[
\text{WHILE } X > 0 \text{ DO } c \text{ END}
\]

\[
\{\{ Y > 0 \}\}
\]

\begin{tabular}{ccc}
\text{Always valid} & \text{Sometimes valid} & \text{Never valid} \\
\end{tabular}

\[
c_1 = \\
\]

\[
c_2 = \\
\]

\[ \{\{ X = Y \}\} \]

\[
c ;; c \\
\]

\[
\{\{ X = Y + 1 \}\}
\]

\begin{tabular}{ccc}
\text{Always valid} & \text{Sometimes valid} & \text{Never valid} \\
\end{tabular}

\[
c_1 = \\
\]

\[
c_2 = \\
\]
5. **Loop invariants** (12 points)

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

\[
\{\{ P \}\} \text{WHILE } b \text{ DO } c \text{ END } \{\{ Q \}\}
\]

Specifically, you should decide whether Inv satisfies each of the specific constraints

- **Implied by precondition:** $P \rightarrow \text{Inv}$
- **Preserved by loop body:** $\{\{ \text{Inv} \land b \}\} c \{\{ \text{Inv} \}\}$$
- **Implies postcondition:** $(\text{Inv} \land \sim b) \rightarrow \text{Q}$

from the Hoare rule for WHILE.

### 5.1
\[
\{\{ X = 3 \}\}
\]

Inv: $X \neq 4$

WHILE $X \cdot X < 10$ DO
\[
\begin{align*}
X & := X + 2 \\
\end{align*}
\]
END
\[
\{\{ X = 5 \}\}
\]

(a) **Implied by precondition**
- [ ] True
- [ ] False

(b) **Preserved by loop body**
- [ ] True
- [ ] False

(c) **Implies postcondition**
- [ ] True
- [ ] False

### 5.2
\[
\{\{ X = 0 \}\}
\]

Inv: $X = 0$

WHILE $X < Y$ DO
\[
\begin{align*}
X & := Y - X \\
Y & := Y - X
\end{align*}
\]
END
\[
\{\{ Y = 0 \}\}
\]

(a) **Implied by precondition**
- [ ] True
- [ ] False

(b) **Preserved by loop body**
- [ ] True
- [ ] False

(c) **Implies postcondition**
- [ ] True
- [ ] False
5.3 \(\{\{ X = 0 \}\}\) Inv: True

WHILE X = 0 DO
  SKIP
END
\(\{\{ X <> 0 \}\}\)

(a) Implied by precondition
  □ True □ False

(b) Preserved by loop body
  □ True □ False

(c) Implies postcondition
  □ True □ False

5.4 \(\{\{ X > 0 \}\}\) Inv: False

WHILE X > 0 DO
  X ::= X + 1
END
\(\{\{ X > 0 \}\}\)

(a) Implied by precondition
  □ True □ False

(b) Preserved by loop body
  □ True □ False

(c) Implies postcondition
  □ True □ False
6. [Advanced Only] Hoare logic (16 points)

In Hoare.v, we experimented with extending Imp with a REPEAT command, written either CRepeat b c or REPEAT c UNTIL b END. The meaning of this command is to execute the command c one or more times, until the condition b is satisfied. That is, after each time c is executed, we check if b holds; if it does hold, we halt the loop.

The ceval relation (repeated on page 2 of the appendix) is extended with the following rules:

| E_RepeatEnd : forall st st' b1 c1, ceval st c1 st' -> beval st' b1 = true -> ceval st (CRepeat c1 b1) st' |
| E_RepeatLoop : forall st st' st'' b1 c1, ceval st c1 st' -> beval st' b1 = false -> ceval st' (CRepeat c1 b1) st'' -> ceval st (CRepeat c1 b1) st'' |

Here is one possible Hoare-logic rule for REPEAT loops:

\[
\begin{align*}
\{ P \} \ c \ \{ P \} \\
\{ P \} \ \text{REPEAT} \ c \ \text{UNTIL} \ b \ \text{END} \ \{ P \land b \} \\
\end{align*}
\] (hoare_repeat)

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

Please use full, grammatical sentences. If you use induction, be explicit about the induction hypothesis. If you run out of space use the following blank page and write “continued” at the bottom of this page so we know to look for the rest (Gradescope only displays one page at a time).
7. [Standard Only] Small-step reduction (12 points)

Consider the simply typed lambda-calculus with booleans and conditionals, defined on pages ?? and ??.

Here are three examples of STLC terms. The following questions ask you to decide which of these terms are values, and then to answer how each of the three terms steps.

Coq notation: Informal notation:
Example e1 := tabs x TBool (tapp (tvar x) (tvar x)) \x:T. x x
Example e2 := tapp e1 e1 e1 e1 e1 e1
Example e3 := tif (tapp e1 ttrue) e2 e2 if e1 true then e2 else e2 e2

(None of these terms are actually well-typed, but this does not prevent us from reasoning about their reduction behavior, which is all we are interested in here.)

7.1 Which of the above terms are values?

☐ e1  ☐ e2  ☐ e3

7.2 List all the terms that e1 multi-steps to.

7.3 What term (if any) does e2 single-step to?

7.4 List all the terms that e3 multi-steps to.
8. [Advanced Only] Big-step and small-step semantics (16 points)

Again, consider the simply typed lambda-calculus with booleans and conditionals (pages ?? and ?? in the appendix). An alternative big-step evaluation relation for the same language is given on page ??.

Give careful informal proofs of the following theorem and the easy corollary on the next page. The beginning of each proof is given; please fill in the requested cases. Use full, grammatical sentences. Again, if you run out of space use the following blank page and write “continued” at the bottom of this page.

8.1 Theorem 1: If \( t \Rightarrow s \) and \( s \Downarrow v \), then \( t \Downarrow v \).

Proof: By induction on a derivation of \( t \Rightarrow s \).

Give just the cases where the last rule in the derivation is ST_AppAbs or ST_App1. (The other cases are similar.)
8.2 Corollary 2: If \( t \Rightarrow \Rightarrow v \) and \( v \) is a value, then \( t \not\Rightarrow v \).

Proof: By induction on a derivation of \( t \Rightarrow \Rightarrow v \)—that is, a derivation of \( \text{multi step } t \Rightarrow v \), where \( \text{step} \) is the single-step reduction relation from page ?? and \( \text{multi} \) was defined as follows:

\[
\text{Inductive } \text{multi} \ {X:\text{Type}} \ (R: \text{relation } X) : \text{relation } X := \\
\mid \text{multi_refl} : \forall (x : X), \quad \text{multi } R \ x \ x \\
\mid \text{multi_step} : \forall (x \ y \ z : X), \\
\qquad R \ x \ y \rightarrow \\
\qquad \text{multi } R \ y \ z \rightarrow \\
\qquad \text{multi } R \ x \ z.
\]
9. **Properties of reduction and typing** (16 points)

Suppose we are given some new programming language — i.e., someone specifies

- a set of terms **Inductive tm**,
- a property **value** : **tm** -> **Prop** that picks out a subset of terms designated as values,
- a small-step reduction relation **Inductive stepsto** : **tm** -> **tm** -> **Prop**,
- a set of types **Inductive ty**,
- a typing relation **Inductive hastype** : **tm** -> **ty** -> **Prop**.

For simplicity, let’s suppose that this language doesn’t have any variable binders, so we don’t need any contexts, and typing is just a two-place relation. Please also assume that the typing relation makes no mention of the step relation, and vice versa.

We use lower-case variables like **t** to stand for terms (in **tm**) and upper-case variables like **T** to stand for types (in **ty**). We write **t1 ==> t2** to mean **stepsto t1 t2** and \( |- t \in T \) to mean **hastype t T**.

Further, suppose that we are told the following facts about this language:

- **Uniqueness of typing**: If **t** is a term and **T1**, **T2** are types such that \( |- t \in T1 \) and \( |- t \in T2 \), then \( T1 = T2 \).
- **Determinism of reduction**: If \( t \Rightarrow t1 \) and \( t \Rightarrow t2 \), then \( t1 = t2 \).
- **Progress**: If \( |- t \in T \), then either **t** is a value or else there is some **t’** such that \( t \Rightarrow t’ \).
- **Preservation**: If \( |- t \in T \) and \( t \Rightarrow t’ \), then \( |- t’ \in T \).
- **Values are normal forms**: If **t** is a value, then there is no **t’** such that \( t \Rightarrow t’ \).

In each of the following parts (on the next page), we ask you to consider how a proposed change to this language will affect these properties (without knowing anything more about the details of the language). If the proposed change will **definitely break** the corresponding property, check the box by “fails.” If the proposed change definitely **cannot break** the property, check the box next to “holds.” If this change **might or might not break** the property, depending on the details of the original language and/or exactly what is added or removed, choose “depends.”
If we restrict the typing relation (i.e., we take one or more pairs of a term \( t \) and a type \( T \) with \( |- t \in T \) and remove them from the relation), what happens to these properties?

- **Uniqueness of typing:**  □ holds □ fails □ depends
- **Determinism of reduction:** □ holds □ fails □ depends
- **Progress:** □ holds □ fails □ depends
- **Preservation:** □ holds □ fails □ depends

If we enlarge the typing relation (i.e., we add one or more new pairs \( |- t \in T \) that were not in the original relation), what happens?

- **Uniqueness of typing:** □ holds □ fails □ depends
- **Determinism of reduction:** □ holds □ fails □ depends
- **Progress:** □ holds □ fails □ depends
- **Preservation:** □ holds □ fails □ depends

If we restrict the reduction relation for well-typed terms (i.e., we take one or more pairs of terms \( t \) and \( t' \) with \( t \Rightarrow t' \) and \( |- t \in T \) for some type \( T \) and remove them from the reduction relation), what happens?

- **Uniqueness of typing:** □ holds □ fails □ depends
- **Determinism of reduction:** □ holds □ fails □ depends
- **Progress:** □ holds □ fails □ depends
- **Preservation:** □ holds □ fails □ depends

If we enlarge the reduction relation for well-typed terms (i.e., we add one or more new pairs \( t \Rightarrow t' \), where \( |- t \in T \) is in the original typing relation for some type \( T \)), what happens?

- **Uniqueness of typing:** □ holds □ fails □ depends
- **Determinism of reduction:** □ holds □ fails □ depends
- **Progress:** □ holds □ fails □ depends
- **Preservation:** □ holds □ fails □ depends
10. Simply Typed Lambda Calculus (10 points)

10.1 Is it possible to find a term \( t \) and a type \( T \) such that \( |- t t \in T \) in the ordinary simply typed lambda-calculus with booleans and pairs (without subtyping) (pages ?? to 9 in the appendix)? If so, give an example. If not, explain why this cannot occur.

☐ Yes ☐ No

10.2 Is it possible to find a term \( t \) and a type \( T \) such that \( |- t t \in T \) in the simply typed lambda-calculus with booleans, pairs, and subtyping (pages ?? to 10 in the appendix)? If so, give an example. If not, explain why this cannot occur.

☐ Yes ☐ No
11. [Standard Only] (Subtyping) (8 points)

For each of the following pairs of types, S and T, mark the appropriate description of how they are ordered by the subtype relation. (In the last two, {} is the empty record type—a record type with no fields.)

11.1 \( S = (\text{Top} \to \text{Bool}) \to (\text{Nat} \to \text{Top}) \)
\( T = \text{Top} \to (\text{Top} \to \text{Bool}) \)

- [ ] \( S <: T \)
- [ ] \( T <: S \)
- [ ] equivalent (both \( S <: T \) and \( T <: S \))
- [ ] unrelated (neither \( S <: T \) nor \( T <: S \))

11.2 \( S = \{x: \text{Bool}, y: \text{Top}\} \)
\( T = \{y: \text{Top} \to \text{Top}, x: \text{Top}\} \)

- [ ] \( S <: T \)
- [ ] \( T <: S \)
- [ ] equivalent
- [ ] unrelated

11.3 \( S = \{x: \{\}, y: \text{Top}\} \)
\( T = \{y: \{\}\} \)

- [ ] \( S <: T \)
- [ ] \( T <: S \)
- [ ] equivalent
- [ ] unrelated

11.4 \( S = \{\} \)
\( T = \text{Top} \)

- [ ] \( S <: T \)
- [ ] \( T <: S \)
- [ ] equivalent
- [ ] unrelated
12. **Subtyping** (15 points)

In this problem, we again consider the simply typed lambda-calculus with booleans, products, and subtyping (pages ?? to 10 in the appendix).

### 12.1
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:

### 12.2
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:

### 12.3
Is there a type with only finitely many subtypes?

- [ ] Yes
- [ ] No

If Yes, give an example and list *all* of its subtypes:

### 12.4
Is there a type with only finitely many supertypes?

- [ ] Yes
- [ ] No

If Yes, give an example and list *all* of its supertypes:

### 12.5
Is there a term $t$ that has infinitely many types in the empty context (i.e., such that the set of $T$ with $|- t \in T$ is infinite)?

- [ ] Yes
- [ ] No

If Yes, give an example:
For Reference

Formal definitions for Imp

Syntax

Inductive aexp : Type :=
| ANum : nat -> aexp
| AId : string -> 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 : string -> aexp -> com
| CSeq : com -> com -> com
| CIf : bexp -> com -> com -> com
| CWhile : bexp -> com -> com.

Notation "'SKIP'" :=
CSkip.
Notation "l '::=' a" :=
(CAss l a).
Notation "c1 ;; c2" :=
(CSeq c1 c2).
Notation "'WHILE' b 'DO' c 'END'" :=
(CWhile b c).
Notation "'IFB' e1 'THEN' e2 'ELSE' e3 'FI'" :=
(CIf e1 e2 e3).
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).

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

Hoare logic rules

\[ \begin{array}{c}
\{ \text{assn_sub X a Q} \} X := a \{ Q \} \quad (\text{hoare_asgn}) \\
\{ P \} \text{SKIP} \{ P \} \quad (\text{hoare_skip}) \\
\{ P \} c_1 \{ Q \} \\
\{ Q \} c_2 \{ R \} \\
\{ P \} c_1 ;; c_2 \{ R \} \quad (\text{hoare_seq}) \\
\{ P \land b \} c_1 \{ Q \} \\
\{ P \land \neg b \} c_2 \{ Q \} \\
\{ P \} \text{IFB b THEN c_1 ELSE c_2 FI} \{ Q \} \quad (\text{hoare_if}) \\
\{ P \} \text{WHILE b DO c END} \{ P \land \neg b \} \quad (\text{hoare_while}) \\
\{ P' \} c \{ Q' \} \\
P \rightarrow P' \\
Q' \rightarrow Q \\
\{ P \} c \{ Q \} \quad (\text{hoare_consequence}) \\
\{ P' \} c \{ Q \} \\
P \rightarrow P' \\
\{ P \} c \{ Q \} \quad (\text{hoare_consequence_pre}) \\
\{ P \} c \{ Q' \} \\
Q' \rightarrow Q \\
\{ P \} c \{ Q \} \quad (\text{hoare_consequence_post}) \\
\end{array} \]
STLC with booleans

Syntax

\[ T ::= \text{Bool} \quad t ::= x \quad v ::= \text{true} \]

\[ | \quad T \to T \quad | \quad t \quad t \quad | \quad \text{false} \]

\[ | \quad \forall x:T. \quad t \quad | \quad \forall x:T. \quad t \]

\[ | \quad \text{true} \quad | \quad \text{false} \quad | \quad \text{if } t \text{ then } t \text{ else } t \]

Small-step operational semantics

\[
\text{value } v \\
\text{------------------------ (ST_AppAbs)} \\
(\forall x:T. t) v \Rightarrow [x:=v)t
\]

\[
t1 \Rightarrow t1' \\
\text{--------------------- (ST_App1)} \\
t1 \ t2 \Rightarrow t1' \ t2
\]

\[
\text{value } v1 \\
\text{t2} \Rightarrow t2' \\
\text{------------------------ (ST_App2)} \\
v1 \ t2 \Rightarrow v1 \ t2'
\]

\[
\text{------------------------ (ST_IfTrue)} \\
(\text{if } \text{true} \text{ then } t1 \text{ else } t2) \Rightarrow t1
\]

\[
\text{------------------------ (ST_IfFalse)} \\
(\text{if } \text{false} \text{ then } t1 \text{ else } t2) \Rightarrow t2
\]

\[
t1 \Rightarrow t1' \\
\text{-------------------------- (ST_If)} \\
(\text{if } t1 \text{ then } t2 \text{ else } t3) \Rightarrow (\text{if } t1' \text{ then } t2 \text{ else } t3)
\]
Typing

\[ \text{Gamma} \ x = T \]
\[ \text{-------------------------- (T_Var)} \]
\[ \text{Gamma} \vdash x \in T \]

\[ \text{Gamma, } x:T_11 \vdash t_{12} \in T_{12} \]
\[ \text{--------------------------- (T.Abs)} \]
\[ \text{Gamma} \vdash \\lambda x:T_11.t_{12} \in T_{11} \to T_{12} \]

\[ \text{Gamma} \vdash t_1 \in T_{11} \to T_{12} \]
\[ \text{Gamma} \vdash t_2 \in T_{11} \]
\[ \text{-------------------------- (T_App)} \]
\[ \text{Gamma} \vdash t_1 \ t_2 \in T_{12} \]

\[ \text{------------------------ (T_True)} \]
\[ \text{Gamma} \vdash \text{true} \in \text{Bool} \]

\[ \text{------------------------ (T.False)} \]
\[ \text{Gamma} \vdash \text{false} \in \text{Bool} \]

\[ \text{Gamma} \vdash t_1 \in \text{Bool} \quad \text{Gamma} \vdash t_2 \in T \quad \text{Gamma} \vdash t_3 \in T \]
\[ \text{----------------------------- (T.If)} \]
\[ \text{Gamma} \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \in T \]

Properties of STLC

Theorem preservation: for all \( t \), \( t' \), \( T \),
\[ \text{empty} \vdash t \in T \quad \Rightarrow \quad t \Rightarrow t' \quad \Rightarrow \quad \text{empty} \vdash t' \in T. \]

Theorem progress: for all \( t \), \( T \),
\[ \text{empty} \vdash t \in T \quad \Rightarrow \quad \text{value } t \quad \lor \quad \text{exists } t', t \Rightarrow t'. \]
STLC with products

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

\[
T ::= \ldots \\
| T \times T \\
| (t, t) \\
| (v, v) \\
| t.\text{fst} \\
| t.\text{snd}
\]

Small-step operational semantics (added to STLC rules)

\[
t1 \Rightarrow t1' \\
\hline \hline (t1, t2) \Rightarrow (t1', t2) \\
\hline (t2 \Rightarrow t2') \\
\hline (v1, t2) \Rightarrow (v1, t2') \\
\hline (t1 \Rightarrow t1') \\
\hline t1.\text{fst} \Rightarrow t1'.\text{fst} \\
\hline (v1, v2).\text{fst} \Rightarrow v1 \\
\hline (t1 \Rightarrow t1') \\
\hline t1.\text{snd} \Rightarrow t1'.\text{snd} \\
\hline (v1, v2).\text{snd} \Rightarrow v2
\]

Typing (added to STLC rules)

\[
\Gamma \vdash t1 \in T1 \\
\Gamma \vdash t2 \in T2 \\
\hline \hline \Gamma \vdash (t1, t2) \in T1 \times T2 \\
\hline \hline \Gamma \vdash t1 \in T11 \times T12 \\
\hline \hline \Gamma \vdash t1.\text{fst} \in T11 \\
\hline \hline \Gamma \vdash t1 \in T11 \times T12 \\
\hline \hline \Gamma \vdash t1.\text{snd} \in T12
\]
STLC with Booleans and Subtyping

Extend the language from pages ?? to 9 with the type Top (terms and values remain unchanged):

\[
T ::= \ldots
| \text{Top}
\]

Add these rules that characterize the subtyping relation:

\[
\begin{align*}
S & <: U \quad U <: T \\
\phantom{\text{S <: U}} & <: T & (S\_Trans)
\end{align*}
\]

\[
\begin{align*}
\phantom{\text{S <: U}} & <: T & (S\_Refl)
\end{align*}
\]

\[
\begin{align*}
\phantom{\text{S <: U}} & <: T & (S\_Top)
\end{align*}
\]

\[
\begin{align*}
S1 & <: T1 \quad S2 <: T2 \\
\phantom{\text{S1 <: T1}} & <: T1 * T2 & (S\_Prod)
\end{align*}
\]

\[
\begin{align*}
\phantom{\text{S1 <: T1}} & <: T1 \quad \phantom{\text{S2 <: T2}} & <: T1 \rightarrow T2 & (S\_Arrow)
\end{align*}
\]

And add this to the typing relation:

\[
\begin{align*}
\Gamma & \vdash t \in S \quad S <: T \\
\phantom{\Gamma \vdash t \in S} & \vdash t \in T & (T\_Sub)
\end{align*}
\]
Big-step evaluation relation for STLC + Booleans

\[
\begin{align*}
\text{value } v \\
\hspace{1cm} v \backslash\backslash v \\
\end{align*}
\]

\[
\begin{align*}
t1 \backslash\backslash \\langle x:T.t1' \rangle & \quad t2 \backslash\backslash v2 \quad [x:=v2]t1' \backslash\backslash v \\
\end{align*}
\]

\[
\begin{align*}
t1 \quad t2 \quad \backslash\backslash v \\
\end{align*}
\]

\[
\begin{align*}
t0 \quad \\langle \text{true} \rangle & \quad t1 \quad \backslash\backslash v \\
\end{align*}
\]

\[
\begin{align*}
(\text{if } t0 \text{ then } t1 \text{ else } t2) \quad \backslash\backslash v \\
\end{align*}
\]

\[
\begin{align*}
t0 \quad \langle \text{false} \rangle & \quad t2 \quad \backslash\backslash v \\
\end{align*}
\]

\[
\begin{align*}
(\text{if } t0 \text{ then } t1 \text{ else } t2) \quad \backslash\backslash v \\
\end{align*}
\]