CIS 5000 Final Exam 12 December 2024

Name (printed): _____

Pennkey (login id): _____

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

Signature: _____ Date: _____

Directions:

- You have 120 minutes to complete the exam.
- There are 120 total points
- There is an appendix that contains relevant definitions that may be useful Imp, Hoare Logic, and STLC questions.
- Please write your PennKey (login ID, e.g. `stevez`) in the blanks provided (so that we can find things if the staple fails!)

1. **Grab Bag** (14 points)

(6 points) Mark the following statements as True or False.

(a) To ensure consistency, every *tactic* used to interactively construct a Coq proof term must terminate.

☐ True ☐ False

(b) In Coq, the term `fun (X : Type) ⇒ fun (f : X → X) ⇒ fun (a : X) ⇒ f (f (f a))` has type `∀ X, X → X`.

☐ True ☐ False

(c) In Coq, the type `Bool → False` is inhabited (i.e., there is a closed value of that type).

☐ True ☐ False

(d) Using the definition of STLC (with pairs) and its operational semantics shown in the Appendix, there is an STLC term `t` such that for every `t'` such that `t ⟶* t'`, `t'` can also take a step.

☐ True ☐ False

(e) Recall that propositional extensionality states that

```
∀ (P Q : Prop), (P ↔ Q) → P = Q
```

This is provable without an axiom Coq.

☐ True ☐ False

(f) Recall that the excluded middle property states that

```
∀ (P : Prop), P ∨ ~P
```

This is provable without an axiom Coq.

☐ True ☐ False

(g) (2 points) Which of the following are *dependent types*? (choose all that apply)

☐ `nat → Prop`

☐ `nat → bool`

☐ `fun (x:nat) ⇒ x = 3`

☐ `∀ (x:nat), x = 3`

☐ `∀ (x:nat), bool`

(h) (4 points) Recall the definition of the `le` relation, which is this inductively defined proposition:

```
Inductive le : nat → nat → Prop :=
| le_0 (n : nat)                    : le 0 n
| le_S_both (n m : nat) (H: le n m) : le (S n) (S m).
```

Write the *type* for each of the following terms:

- `le_0` : _____
- `(le_S_both 4)` : _____

Recall the `ev` predicate which characterizes the set of *even* natural numbers:

```
Inductive ev : nat → Prop :=
| ev_0                    : ev 0
| ev_SS (n : nat) (H : ev n) : ev (S (S n)).
```

(2 points) For the lemma below, indicate what tactics, besides `intros` and `apply` are needed to finish the proof, or mark it "not provable" if that is the case. (choose one)

```
Lemma ev_not_S: ∀ n, ev n → ~ ev (S n).
```

☐ Not provable.

☐ Provable without using `inversion` or `induction`.

☐ Provable using only `inversion` (but not `induction`).

☐ Provable only by using `induction`.

2. **Imp semantics and Hoare Logic** (23 points total)

The following questions refer to the Imp semantics and Hoare logic, whose definitions are summarized in Appendix A.

(a) For each pair of Imp commands `c1` and `c2` below, indicate whether they are behaviorally equivalent (as defined by `cequiv`). If they are not, write a Hoare-logic assertion describing a starting state that distinguishes them. We have done the first one for you.

i.

```
c1:                          c2:
   X := Y; X := 0               X := Y
```

☐ equivalent

☒ inequivalent in a state satisfying: `{{ Y <> 0 }}`

ii. (2 points)

```
c1:                          c2:
   skip                         while true do skip end
```

☐ equivalent

☐ inequivalent in a state satisfying: _____

iii. (2 points)

```
c1:                          c2:
   while true do                while X > 0 do
     X := 3                        X := X + 1
   end                          end
```

☐ equivalent

☐ inequivalent in a state satisfying: _____

iv. (2 points)

```
c1:                          c2:
   if X > Y then                Y := X
     while X <> Y do
       Y := Y + 1
     end
   else
     while X <> Y do
       Y := Y - 1
     end
   end
```

☐ equivalent

☐ inequivalent in a state satisfying: _____

v. (2 points)

```
c1:                          c2:
   if X > Y then                if X < Y then
     while true do skip end       Y := X
   else                         else
     Y := X                       while true do skip
   end                          end
```

☐ equivalent

☐ inequivalent in a state satisfying: _____

(b) (5 points) Consider the following Hoare triple, with a missing precondition ???

```
{{ ??? }}
   while X < 3 do
      X := X + 1;
      Y := Y - 1
   end
{{ X = 3 ∧ Y = 0 }}
```

Which of the following assertions could fill in for ??? so that the Hoare triple is valid? (choose all that apply)

☐ `True`

☐ `False`

☐ `3 < X ∧ Y = 3`

☐ `X < 3 ∧ 4 < Y`

☐ `X < 0 ∧ Y < 2`

(c) (5 points) Consider the following Hoare triple, with a missing postcondition ???

```
{{ True }}
  skip; X := 5; Y := 5
{{ ??? }}
```

Which of the following assertions could fill in for ??? so that the Hoare triple is valid? (choose all that apply)

☐ `True`

☐ `False`

☐ `X = 5`

☐ `Y = 5`

☐ `X = 5 ∧ Y = 5`

(d) (5 points) Consider the following Hoare triple, with a missing command ???

```
{{ X = 42 }}
   while 0 < X do
     X := X + 1;
     ???;
   end
{{ X = 42 }}
```

Which of the following commands could fill in for ??? so that the Hoare triple is valid? (choose all that apply)

☐ `Y := 3`

☐ `X := Y`

☐ `while X < 0 do skip end`

☐ `skip`

☐ `X := X - 2`

3. **Simply-typed Lambda Calculus with Pairs** (24 points total)

Appendix B contains the syntax, small-step operational semantics, and typing relation for a variant of the simply-typed lambda calculus (STLC) with `Bool` and pair types.

(a) (2 points) Of the 12 rules that define the small step semantics, how many of them are *congruence* rules?

Fill in the blank with 0–12: _____

Note: The following questions are *independent* (i.e., the proposed changes to STLC are *not* cumulative).

(b) (4.5 points) Suppose we replace the typing rule `T_If` with the one shown below (leaving all other rules unchanged):

```
Gamma ⊢ t1 ∈ T1
Gamma ⊢ t2 ∈ T2
------------------------------------------- (T_If')
Gamma ⊢ <{ if true then t1 else t2 }> ∈ T1
```

What will happen to each of the following properties? Give a term that provides a counterexample for those that fail:

Preservation: ☐ Still holds
☐ Fails with counterexample: _____

Progress: ☐ Still holds
☐ Fails with counterexample: _____

Determinism: ☐ Still holds
☐ Fails with counterexample: _____

(c) (4.5 points) Suppose we replace the typing rule `T_If` with the one shown below (leaving all other rules unchanged):

```
Gamma ⊢ t1 ∈ T1
Gamma ⊢ t2 ∈ T2
------------------------------------------- (T_If'')
Gamma ⊢ <{ if true then t1 else t2 }> ∈ T2
```

What will happen to each of the following properties? Give a term that provides a counterexample for those that fail:

Preservation: ☐ Still holds
☐ Fails with counterexample: _____

Progress: ☐ Still holds
☐ Fails with counterexample: _____

Determinism: ☐ Still holds
☐ Fails with counterexample: _____

(d) (4.5 points) Suppose we add a new typing rule shown below (leaving all other rules unchanged):

```
------------------------------------ (T_FstTrue)
Gamma ⊢ <{ true.fst }> ∈ Bool * Bool
```

What will happen to each of the following properties? Give a term that provides a counterexample for those that fail:

Preservation: ☐ Still holds
☐ Fails with counterexample: _____

Progress: ☐ Still holds
☐ Fails with counterexample: _____

Determinism: ☐ Still holds
☐ Fails with counterexample: _____

(e) (4.5 points) Suppose we replace the step rule `ST_Pair2` with the one shown below (leaving all other rules unchanged):

```
       t2 ⟶ t2'
------------------------ (ST_Pair2')
(t1, t2) ⟶ (true, t2')
```

What will happen to each of the following properties? Give a term that provides a counterexample for those that fail:

Preservation: ☐ Still holds
☐ Fails with counterexample: _____

Progress: ☐ Still holds
☐ Fails with counterexample: _____

Determinism: ☐ Still holds
☐ Fails with counterexample: _____

(f) (4 points) Suggest a change to STLC such that *Preservation* and *Progress* still hold, but *Determinism* fails.

The change is to (choose one option and fill in the blank):

☐ remove the typing rule (rule name): _____

☐ remove the step rule (rule name):_____

☐ add a `step` rule: (use the space below)

☐ add a new typing rule: (use the space below)

new rule:

4. **STLC + Void** (12 points total)

Recall that Coq has types like `False` that have no values. Suppose we add such a type, called `Void`, to the STLC from Appendix B as shown below:

```
T := ...
   | Void

t := ...
   | explode t as T


              t ⟶ t'
-----------------------------------     (ST_Explode)
explode t as T ⟶ explode t' as T


Gamma ⊢ t ∈ Void
--------------------------  (T_Explode)
Gamma ⊢ explode t as T ∈ T
```

The rest of the definitions remain unchanged. Note that: (1) there are *no* values of type `Void`, and (2) the term `explode t as T` has *any* type `T`, as long as `t` is a term of type `Void`. (This is analogous to how, in Coq, we can prove *anything* using the "principle of explosion," hence the name.)

(a) (8 points) For each term below, fill in the blank with its type according to this new type system, or write "ill typed" if there is no way to assign the term a valid type.

`empty ⊢ \x:Void, (explode x as Bool) ∈` _____

`empty ⊢ \x:Void, ((explode x as (Void → Void)) x) ∈` _____

`empty ⊢ \x:Bool, ((explode x as (Bool → Void)) true) ∈` _____

`empty ⊢ \x:Bool, \f:(Bool → Void), (f x) ∈` _____

(b) As noted above, this system satisfies the following lemma:

```
Lemma void_not_value :
  ∀ t, <{ empty ⊢ t ∈ Void }> → ~(value t).
```

(i) (2 points) Suppose we start the proof of this fact by doing `intros t H C`. Which of the following tactics will complete the proof? (choose one)

☐ `induction t; inversion H.`

☐ `induction H; inversion C.`

☐ `induction C; inversion H.`

☐ `induction C; inversion t.`

(ii) (2 points) The `void_not_value` lemma is most useful to prove which of the following properties of STLC+Void? (choose one)

☐ `step_deterministic`

☐ `progress`

☐ `preservation`

☐ `substitution_preserves_typing`

5. **STLC + Poly** (25 points total)

This problem considers one way to extend the simply-typed lambda calculus with *ad-hoc polymorphism*. The idea is to use a Coq-level *function* of type `ty → tm` as a way to represent a lambda calculus term that can analyze an STLC type—that is, we create STLC terms that behave like functions on types.

For example, consider the follow Coq function, which does case analysis on the given STLC type `(T:ty)` to determine whether it is `Bool`, returning an STLC term `<{ true }>` or `<{ false }>` as appropriate:

```
Definition is_Bool : ty → tm :=
  fun (T:ty) ⇒
    match T with
    | <{{ Bool }}> ⇒ <{ true }>
    | _ ⇒ <{ false }>
    end.
```

Just as we can embed Coq-level natural numbers as STLC values using a constructor `tm_nat : nat → tm`, we can embed such a Coq function as an STLC term using a constructor `tm_poly : (ty → tm) → tm`. So `tm_poly is_Bool` is a valid STLC term of type `tm`. We introduce the notation `LAM $(is_Bool)` as nicer syntax, where `LAM` indicates that term is intended to act like a function and `$` reminds us that the embedded value is a Coq-level function.

In order to use a term like `LAM $(is_Bool)`, we also add a "type application" term `t @ T`, which calls a polymorphic function `t` with the type argument `T`. The intended operational semantics are illustrated by the examples below, where, if we apply `LAM $(is_Bool)` to the type `Bool` the term reduces to STLC-value `true`, and if we apply it instead to the type `Bool → Bool` it reduces to `false`:

```
Lemma example1 : <{ LAM $(is_Bool) @ Bool }> ⟶* <{ true }>.

Lemma example2 : <{ LAM $(is_Bool) @ (Bool → Bool) }> ⟶* <{ false }>.
```

We can add this feature to STLC in a type-safe way. As usual, we will do this by (a) extending the syntax, (b) extending the operational semantics, (c) extending the type system, and (d) proving the metatheory properties. Each of these steps is covered below.

(a) Syntax The informal definitions below summarize the changes to the term and type syntax:

```
t ::=                Terms (tm)
    ...
    | LAM $(coq_fn)     embedded Coq function
    | t @ T             type application

T ::=                Types (ty)
    | T → T                 arrow type
    | Bool                  Boolean type
    | T * T                 product type
    | Poly T                polymorphic type
```

(2 points) What type should be filled in for the `???` below to add type application `t @ T` syntax to the definition of `tm`? (choose one)

```
Inductive tm : Type :=
  ...
  | tm_poly   : (ty → tm) → tm
  | tm_ty_app : ???
```

☐ `tm → tm`

☐ `(ty → tm) → ty → tm`

☐ `tm → ty → tm`

☐ `ty → ty → tm`

(b) Operational Semantics To add polymorphic terms to STLC we need to decide what are the new values, adjust substitution, and define the `step` relation.

i. (4 points) We adjust the definition of values in Appendix B by adding one new case:

```
Inductive value : tm → Prop :=
  ...
  | v_ty_fun : ∀ (f : ty → tm), value <{ LAM $(f) }>.
```

Which of the following Coq terms are values according to this definition? (check all that apply)

☐ `LAM $(is_Bool)`

☐ `(LAM $(is_Bool)) @ Bool`

☐ `\x:Bool, (LAM $(is_Bool) x)`

☐ `LAM $((fun f ⇒ fun T ⇒ f T) (fun T ⇒ <{ true }>))`

ii. (4 points) We adjust the substitution function from Appendix B by adding cases for `LAM` and type application:

```
[x:=s](LAM $(f)) = LAM $(f)
[x:=s](t @ T)    = ([x:=s]t) @ T
```

For this definition to be coherent with the *substitution lemma*, we need to impose some constraints. What does this definition imply about the occurrence of *free variables* in these terms? (check all true statements)

☐ There are no free STLC variables in an STLC type `(T:ty)`.

☐ The variable `x` may *not* appear free in `t` for the STLC term `t @ T`.

☐ *Only* the variable `x` may appear free in `t` for the STLC term `t @ T`.

☐ The term `LAM $(f)` is considered to be *closed* (i.e. it contains no free variables).

iii. (4 points) We modify the `step` relation by adding two new cases, as shown below. Note that `ST_LAM_App1` simply calls the Coq-level embedded function `f` on `(T:ty)`.

```
| ST_LAM_App1 : ∀ (f:ty → tm) (T:ty),
  <{ (LAM $(f)) @ T }> ⟶ f T

| ST_LAM_App2 : ∀ t0 t0' T,
  (t0 ⟶ t0') →
  <{ t0 @ T }> ⟶ <{ t0' @ T }>
```

Which of the following properties must fail if we omitted `ST_LAM_App1`? (check all that apply)

☐ *determinism* would fail

☐ *progress* would fail

☐ *canonical forms* would fail

☐ *preservation* would fail

(c) Typing Rules To adjust the typing rules for STLC+Poly, we need to ensure that the Coq-level embedded functions somehow still obey a typing discipline. We have added the type form `Poly T`, which should, intuitively, characterize values of polymorphic type that when called on an STCL type `U` return STLC terms of type `T`. We can capture this in the following typing rule:

```
| T_TApp : ∀ Gamma t U T,
    <{ Gamma ⊢ t ∈ Poly T }> →
    <{ Gamma ⊢ t @ U ∈ T }>
```

Type checking the `LAM $(f)` form is a bit more delicate. The returned STLC terms must be *closed* and all agree with the type of `Poly T` as described above. The shape of the typechecking rule is thus given by:

```
| T_Poly : ∀ Gamma (f:ty → tm) T,
    ??? →
    <{ Gamma ⊢ LAM $(f) ∈ Poly T }>
```

(3 points) Which clause should fill in for `???` above so that *preservation* will be possible to prove? (choose one)

☐ `<{ empty ⊢$(f T) ∈T }>`

☐ `(∀(U:ty), <{ Gamma ⊢$(f U) ∈T }>)`

☐ `(∀(U:ty), <{ empty ⊢$(f U) ∈T }>)`

☐ `(∀(U:ty), <{ Gamma ⊢$(f U) ∈U }>)`

☐ `(∀(U:ty), <{ empty ⊢$(f U) ∈U }>)`

(e) Comparison with Coq Recall that Coq is based on the *calculus of inductive constructions*, in which types `(T : Type)` are first-class values. This means that Coq, like STCL+Poly, can express functions whose inputs are types.

i. (2 points) What is the type of the Coq term `(fun (T:Type) ⇒ fun (x:T) ⇒ x)`? (choose one)

☐ `Type → Type`

☐ `∀(T:Type), T → T`

☐ `Type → Type → Type`

☐ `∀(T:Type), (T → T) → T`

ii. (2 points) What is the type of the Coq term `(fun (T:Type) ⇒ ((fun (x:nat) ⇒ x) 3))`? (choose one)

☐ `Type → Type`

☐ `∀(T:Type), T`

☐ `Type → nat`

☐ `Type → nat → nat`

(iii) (4 points) Interestingly, STLC+Poly, as we have defined it, can express different functions than possible in Coq. Which of the following are correct statements about the difference? (mark all correct statements)

☐ All types in Coq are inhabited, whereas STLC+Poly has a *void* (uninhabited) type.

☐ All types in STLC+Poly are inhabited, whereas Coq has a *void* (uninhabited) type.

☐ A Coq term can do case analysis by pattern matching on the type argument, whereas an STLC+Poly term cannot.

☐ An STLC+Poly term can do case analysis by pattern matching on the type argument, whereas a Coq term cannot.

6. **Subtyping** (22 points total)

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

(a) For each of the terms `t` provided below, mark *all* types `T` such that `empty ⊢ t ∈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.

i. (5 points) `empty ⊢ \x : Top, x ∈ T`

☐ `Bool → Top`

☐ `Top → Top`

☐ `(Bool → Bool) → Top`

☐ `Top`

☐ Some other type(s)

☐ Not typeable

ii. (5 points) `empty ⊢ \x : Bool → Top, (x x) ∈ T`

☐ `(Bool → Top) → Top`

☐ `(Bool → Top) → Bool`

☐ `Top → Top`

☐ `Top`

☐ Some other type(s)

☐ Not typeable

iii. (5 points) `empty ⊢ \x : Top → Bool, \y : Bool, x y ∈ T`

☐ `(Top → Bool) → Bool → Bool`

☐ `(Top → Bool) → Bool → Top`

☐ `(Top → Top) → Top`

☐ `(Top → Bool) → Top`

☐ Some other type(s)

☐ Not typeable

b. (4 points)

Suppose we extend the subtyping relation from the appendix to include the new rule subtyping below, but leave the rest of STLC semantics alone:

```
-------------------  (S_Bogus)
Bool <: Bool → Top
```

Which of the following properties would fail? (Check all that apply)

☐ *determinism* would fail

☐ *progress* would fail

☐ *canonical forms* would fail

☐ *preservation* would fail

c. (3 points) Consider STLC+Poly from Problem 5 with the additions from Appendix C. We want to add a new subtyping rule `S_Poly` to reason about our type operator `Poly`.

```
---------------------------  (S_Poly)
      Poly S <: Poly T
```

Check *every* premise that should appear above the line. If none should appear, you should not check any.

☐ `T <: S` ☐ `S = Top` ☐ `S <: T` ☐ `T = Top` ☐ `S != Void` ☐ `S = T`

CIS 5000 2024 Final Exam Appendices

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

Appendix A: Imp Semantics and Hoare Logic Rules

Imp Large Step Semantics

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

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

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

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

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

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

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

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

Imp Hoare Logic Rules

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

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

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

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

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

                   {{P'}} c {{Q'}}
                      P →ₐ P'
                      Q' →ₐ Q
             ---------------------------   (hoare_consequence)
                   {{P}} c {{Q}}
```

Appendix B: Simply-typed Lambda Calculus

This appendix contains the syntax, small-step operational semantics, and typing relation for a variant of the simply-typed lambda calculus with `Bool` and pair types.

Syntax and Types

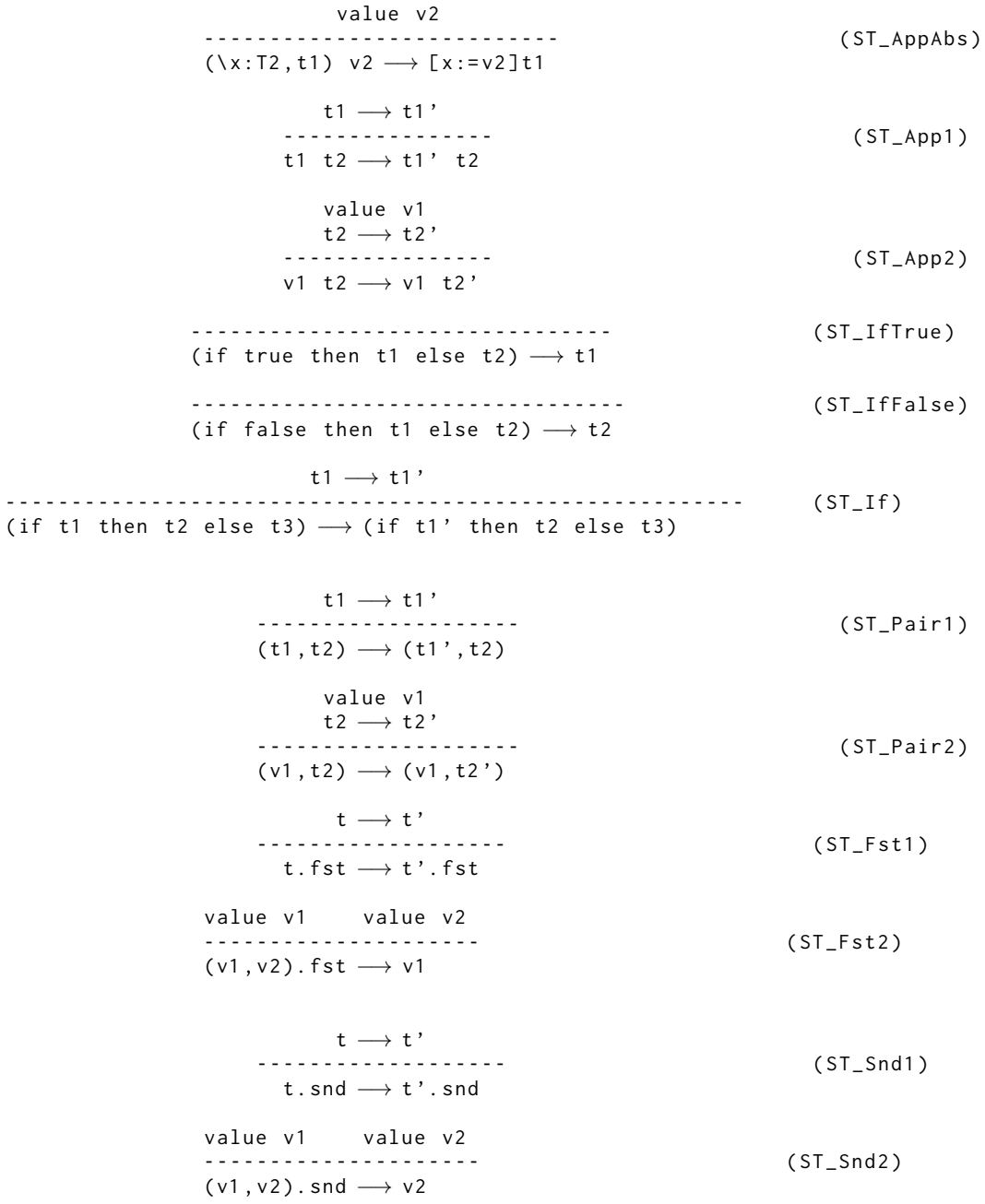
```
t ::=                Terms
    | x                      variable
    | \x:T,t                 abstraction
    | t t                    application
    | true                   constant true
    | false                  constant false
    | if t then t else t     conditional
    | (t, t)                 pair
    | t.fst                  first projection
    | t.snd                  second projection

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

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

Substitution

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

STLC: small step operational semantics



STLC: typing relation

$$\frac{\texttt{Gamma x = T1}}{\texttt{Gamma} \vdash \texttt{x} \in \texttt{T1}} \quad \text{(T\_Var)}$$

$$\frac{\texttt{x} \mapsto \texttt{T2 ; Gamma} \vdash \texttt{t1} \in \texttt{T1}}{\texttt{Gamma} \vdash \texttt{\textbackslash x:T2,t1} \in \texttt{T2} \to \texttt{T1}} \quad \text{(T\_Abs)}$$

$$\frac{\texttt{Gamma} \vdash \texttt{t1} \in \texttt{T2} \to \texttt{T1} \qquad \texttt{Gamma} \vdash \texttt{t2} \in \texttt{T2}}{\texttt{Gamma} \vdash \texttt{t1 t2} \in \texttt{T1}} \quad \text{(T\_App)}$$

$$\frac{}{\texttt{Gamma} \vdash \texttt{true} \in \texttt{Bool}} \quad \text{(T\_True)}$$

$$\frac{}{\texttt{Gamma} \vdash \texttt{false} \in \texttt{Bool}} \quad \text{(T\_False)}$$

$$\frac{\texttt{Gamma} \vdash \texttt{t1} \in \texttt{Bool} \qquad \texttt{Gamma} \vdash \texttt{t2} \in \texttt{T1} \qquad \texttt{Gamma} \vdash \texttt{t3} \in \texttt{T1}}{\texttt{Gamma} \vdash \texttt{if t1 then t2 else t3} \in \texttt{T1}} \quad \text{(T\_If)}$$

$$\frac{\texttt{Gamma} \vdash \texttt{t1} \in \texttt{T1} \qquad \texttt{Gamma} \vdash \texttt{t2} \in \texttt{T2}}{\texttt{Gamma} \vdash \texttt{(t1,t2)} \in \texttt{T1*T2}} \quad \text{(T\_Pair)}$$

$$\frac{\texttt{Gamma} \vdash \texttt{t1} \in \texttt{T1 * T2}}{\texttt{Gamma} \vdash \texttt{t1.fst} \in \texttt{T1}} \quad \text{(T\_Fst)}$$

$$\frac{\texttt{Gamma} \vdash \texttt{t1} \in \texttt{T1 * T2}}{\texttt{Gamma} \vdash \texttt{t1.snd} \in \texttt{T2}} \quad \text{(T\_Snd)}$$

Key Lemmas for STLC

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

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

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

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

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

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

Theorem step_deterministic:
  deterministic step.
```

Appendix C: Subtyping

Additions to the type system:

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

$$\frac{\Gamma \vdash t_1 \in T_1 \qquad T_1 <: T_2}{\Gamma \vdash t_1 \in T_2} \quad \text{(T\_Sub)}$$

Subtyping relation:

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

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

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

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{S_1 * S_2 <: T_1 * T_2} \quad \text{(S\_Prod)}$$

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2} \quad \text{(S\_Arrow)}$$