CIS 500 — Software Foundations

Final Exam

(Advanced version)

December 13, 2013

Name: _____

Pennkey (e.g. `bcpierce`): _____

Scores:

1		12
2		16
3		16
4		16
5		20
6		20
7		10
Total:		110

1. (12 points) Multiple Choice — Coq Programming
   Circle the correct answer. (Each question has one unique correct answer.)

   (a) What is the type of the Coq term: `(fun n:nat => n = 0)`?

      i. `nat -> nat`
      ii. `Prop`
      iii. `nat -> Prop`
      iv. `forall n:nat, n = 0`
      v. ill typed

   (b) What is the type of the Coq term: `(forall n:nat, n = 0)`?

      i. `nat -> nat`
      ii. `Prop`
      iii. `nat -> Prop`
      iv. `forall n:nat, n = 0`
      v. ill typed

   (c) What is the type of the Coq term: `(fun (P:nat -> Prop) (n:nat) (Q:P n) => Q)`?

      i. `nat -> Prop -> nat -> Prop -> Prop`
      ii. `forall (P : nat -> Prop), nat -> Prop`
      iii. `(nat -> Prop) -> (n:nat -> P n) -> Q`
      iv. `forall (P : nat -> Prop) (n : nat), P n -> P n`
      v. ill typed

   (d) What is the type of the Coq term: `((fun P:Prop => P) (3 = 4))`

      i. `Prop`
      ii. `nat`
      iii. `nat -> Prop`
      iv. `Prop -> nat`
      v. ill typed

2. (16 points) Multiple Choice — Imp Equivalence

Circle *all* correct answers. There may be zero or more than one. For reference, the definition of Imp, its evaluation semantics, and program equivalence (`cequiv`) start on page 14.

(a) Consider the Imp program:

```
IF X > 0 THEN
  WHILE X > 0 DO SKIP END
ELSE
  SKIP
FI
```

Which of the following are equivalent to it, according to `cequiv`?

i.
```
WHILE X > 0 DO
  SKIP
END
```

ii.
```
SKIP
```

iii.
```
X ::= 0
```

iv.
```
IF X > 1 THEN
  WHILE X > 0 DO SKIP END
ELSE
  WHILE X > 0 DO SKIP END
FI
```

(b) Consider the Imp program:

```
X ::= 0;;
Y ::= X + 1;;
```

Which of the following are equivalent to it, according to `cequiv`?

i.
```
Y ::= 1;;
X ::= 0;;
```

ii.
```
Y ::= 0;;
X ::= Y + 1;;
```

iii.
```
Y ::= 1;;
WHILE X <> 0 DO
  X ::= X - 1
END
```

iv.
```
Y ::= 2;;
X ::= X - X;;
Y ::= Y - 1;;
```

(c) Consider an arbitrary Imp command `c`. Which of the following are equivalent to `c`, according to `cequiv`?

i.
```
c ;; c
```

ii.
```
X := 1;;
WHILE X > 0 DO
  c ;; X ::= X - 1
DONE
```

iii.
```
IF X > 0 THEN
   c
ELSE SKIP FI
```

iv.
```
IF X < 0 THEN
  SKIP
ELSE c FI
```

(d) Which of the following propositions are provable?

i. `forall c, (exists st:state, c / st || st) -> cequiv c SKIP`

ii.
```
forall c1 c2 st1 s3,
((c1 ;; c2) / st1 || st3) -> (exists st2:state, c1 / st1 || st2 /\ c2 / st2 || st3).
```

iii. `forall c1 c2, (cequiv c1 c2) -> (cequiv (c1;;c2) (c2;;c1))`

iv. `forall c, (forall st, c / st || st) -> cequiv c SKIP.`

3. (16 points) Hoare Logic

The following Imp program computes `m * n`, placing the answer into `Z`.

```
  {{ True }}
X ::= 0 ;;
Z ::= 0 ;;
WHILE X <> n DO
  Y ::= 0 ;;
  WHILE Y <> m DO
    Z ::= Z + 1 ;;
    Y ::= Y + 1 ;;
  END
  X ::= X + 1 ;;
END
  {{ Z = m * n }}
```

On the next page, add appropriate annotations to the program in the provided spaces to show that the Hoare triple given by the outermost pre- and post-conditions is valid. Use informal notations for mathematical formulae and assertions, but please be completely precise and pedantic in the way you apply the Hoare rules — i.e., write out assertions in *exactly* the form given by the rules (rather than logically equivalent ones). The provided blanks have been constructed so that, if you work backwards from the end of the program, you should only need to use the rule of consequence in the places indicated with `->>`.

The Hoare rules and the rules for well-formed decorated programs are provided on pages 16 and 17, for reference.

Mark the implication step(s) in your decoration (by circling the `->>`) that rely on the following fact. You may use other arithmetic facts silently.

- `m * a + m = m * (a + 1)`

```
  {{ True }} ->>
    {{                                                          }}
X ::= 0;;
    {{                                                          }}
Z ::= 0;;
    {{                                                          }}
WHILE X <> n DO
    {{                                                          }} ->>
    {{                                                          }}
  Y ::= 0;;
    {{                                                          }}
  WHILE Y <> m DO
      {{                                                          }} ->>
      {{                                                          }}
    Z ::= Z + 1;;
      {{                                                          }}
    Y ::= Y + 1;;
      {{                                                          }}
  END
    {{                                                          }} ->>
    {{                                                          }}
  X ::= X + 1;;
    {{                                                          }}
END
  {{                                                        }} ->>
  {{ Z = m * n }}
```

4. (16 points) Inductive Definitions and Scoping

Consider the following Coq definitions for a simple language of arithmetic expressions with constants, variables, plus, and `let`.

```
Definition id := nat.
Inductive tm : Type :=
| tnum : nat -> tm             (* Constants 0, 1, 2, ...    *)
| tvar : id -> tm              (* Variables X Y Z ...       *)
| tplus : tm -> tm -> tm       (* Plus:  t1 + t2            *)
| tlet : id -> tm -> tm -> tm. (* Let:   let X = t1 in t2   *)
```

The `let` construct follows the usual variable scoping rules. That is, in `let X = t1 in t2`, written in Coq as `(tlet X t1 t2)`, the variable `X` is *bound* in `t2`.

Recall that a variable `X` *appears free* in a term `t` if there is an occurrence of `X` that is not bound by a corresponding `let`. Complete the following Coq definition of `afi` as an inductively defined relation such that `afi X t` is provable if and only if `X` appears free in `t`.

```
Inductive afi : id -> tm -> Prop :=
```

5. (20 points) Informal Proofs — Substitution

Your job in this problem is to prove a substitution lemma. First, we extend the `let` language from the previous problem by adding a unit constant. Informally, the grammar of the terms and types for the resulting language is given by:

```
t ::=  (* Terms *)                                T ::=  (* Types *)
    |  ()               (* Unit contstant *)          | Unit
    |  n                (* Natural numbers *)         | Nat
    |  X                (* Variables *)
    |  t + t            (* Sum *)                  v ::=  (* Values *)
    |  let X = t in t   (* Let *)                      | ()
                                                      | n
```

The type system is given by:

$$\frac{\Gamma\ \mathtt{X} = \mathtt{T}}{\Gamma \vdash \mathtt{X} \in \mathtt{T}}\ (\mathtt{T\_Var}) \qquad \frac{}{\Gamma \vdash \mathtt{n} \in \mathtt{Nat}}\ (\mathtt{T\_Nat}) \qquad \frac{}{\Gamma \vdash () \in \mathtt{Unit}}\ (\mathtt{T\_Unit})$$

$$\frac{\Gamma \vdash \mathtt{t1} \in \mathtt{Nat} \quad \Gamma \vdash \mathtt{t2} \in \mathtt{Nat}}{\Gamma \vdash \mathtt{t1 + t2} \in \mathtt{Nat}}\ (\mathtt{T\_Sum}) \qquad \frac{\Gamma \vdash \mathtt{t1} \in \mathtt{T1} \quad \Gamma, \mathtt{X{:}T} \vdash \mathtt{t1} \in \mathtt{T2}}{\Gamma \vdash \mathtt{let\ X = t1\ in\ t2} \in \mathtt{T2}}\ (\mathtt{T\_Let})$$

The substitution operation is defined by:

```
[X:=s]() = ()
[X:=s]n  =  n
[X:=s]X  =  s
[X:=s]Y  =  Y   (if X <> Y)
[X:=s](t1 + t2)  =  ([X:=s]t1 + [X:=s]t2)
[X:=s](let X = t1 in t2)  =  (let X = [X:=s]t1 in t2)
[X:=s](let Y = t1 in t2)  =  (let Y = [X:=s]t1 in [X:=s]t2)      (if x <> Y)
```

Also recall the following lemma, which makes use of the concept of "appears free in" from the previous problem. You do *not* need to prove this lemma.

Lemma: Context Invariance

Suppose $\Gamma \vdash \mathtt{t} \in \mathtt{T}$ and that, for all `X`, if `X` appears free in `t` then $\Gamma'\ \mathtt{X} = \Gamma\ \mathtt{X}$. Then $\Gamma' \vdash \mathtt{t} \in \mathtt{T}$.

Give a careful, informal proof of the *substitution* lemma for this language. You may freely use functional extensionality to reason about context equivalence, and you may find it helpful to invoke the context invariance lemma at some point(s).

Lemma: Substitution Preserves Typing
For all `t`, `v`, `X`, Γ, `T`, and `U`, if Γ, `X:U` ⊢ `t` ∈ `T` and ⊢ `v` ∈ `U` then Γ ⊢ `[X:=v]t` ∈ `T`.

6. (20 points) STLC with Natural Number Induction

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 constant zero `0` and successor `S`.

You can find the syntax, typing rules, and small-step evaluation rules for this part of the language beginning on page 19. Note: for this problem we *do not* consider STLC with subtyping, `fix`, or other extensions.

(a) Recall that we can draw typing derivations as "trees" where each node is a judgment of the form $\Gamma \vdash \texttt{t} \in \texttt{T}$. The root of the tree (pictured at the bottom of the drawing) is the desired conclusion, and each premise is a subtree that instantiates a typing rule. For example, the following is a legal typing derivation:

```
    ----------------  T_Zero
     x:Nat ⊢ 0 ∈ Nat
   ------------------  T_Succ
   x:Nat ⊢ S 0 ∈ Nat
---------------------------- T_Abs
⊢ \x:Nat.(S 0) ∈ Nat -> Nat
```

Complete the typing derivation given below. Label the inference rule used at each node of the tree. Note that the type of the root judgment needs to be filled in.

```
------------------------------------------------------------ (T_Abs)
x:Nat ⊢ \y:(Nat -> Nat). S (y (S x))  ∈
```

(b) Rather than adding `if0` and the general recursion operator `fix`, here we follow Coq and add 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 `!!`:

```
(\n:Nat. \m:Nat. nat_ind m (\x:Nat.\y:Nat.S y) n) (S S 0) (S 0)
==>
(\m:Nat. nat_ind m (\x:Nat.\y:Nat.S y) (S S 0)) (S 0)
==>
nat_ind (S 0) (\x:Nat.\y:Nat.S y) (S S 0)
==> !!  inductive case
(\x:Nat.\y:Nat.S y) (S 0) (nat_ind (S 0) (\x:Nat.\y:Nat.S y) (S 0))
==> !!  inductive case
(\x:Nat.\y:Nat.S y) (S 0) ((\x:Nat.\y:Nat.S y) 0 (nat_ind (S 0) (\x:Nat.\y:Nat.S y) 0))
==> !!  base case
(\x:Nat.\y:Nat.S y) (S 0) ((\x:Nat.\y:Nat.S y) 0 (S 0))
==>
(\x:Nat.\y:Nat.S y) (S 0) ((\y:Nat.S y) (S 0))
==>
(\x:Nat.\y:Nat.S y) (S 0) (S (S 0))
==>
(\y:Nat.S y) (S (S 0))
==>
S (S (S 0))
```

In general, the small step semantics of `nat_ind` should work like:

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

where we write `3` as a shorthand for `S S S 0`, etc.

Define the small-step operational semantics for `nat_plus`. There are three "structural" rules that evaluate the arguments to `nat_ind` in order from left-to-right. The first such rule is:

```
              tz ==> tz'
-----------------------------
nat_ind tz ts tn ==> nat_ind tz' ts 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`.

```
                    value vz            value vs
-------------------------------------------------------------------
nat_ind vz vs 0  ==>
```

```
          value vz            value vs           value vn
-------------------------------------------------------------------
nat_ind vz vs (S vn) ==>
```

(c) 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 in part (b) should be well-typed according to your rule.

$\Gamma \vdash$ `tz` $\in$

$\Gamma \vdash$ `ts` $\in$

$$\frac{\Gamma \vdash \texttt{tn} \in \texttt{Nat}}{\Gamma \vdash \texttt{nat\_ind tz ts tn} \in \texttt{T}}$$

(d) Part (b) used `nat_ind` to define the `Nat_plus` function. Use `Nat_plus` and `nat_ind` to define multiplication of two numbers. We have provided the type of `Nat_mult` to get you started:

```
(* Nat_mult : Nat -> Nat -> Nat *)
```

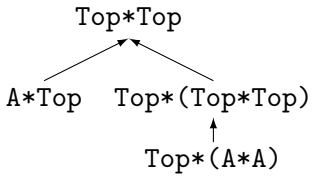
(e) A harder function to define using `nat_ind` is natural number equality, a function `Nat_eq` of type `Nat -> Nat -> Nat` such that `Nat_eq n m ==>* 0` if `n` and `m` are different natural numbers and `nat_eq n m ==>* 1` if they are the same.

We have started the definition. Fill in the two blanks to complete it.

```
(* Nat_eq : Nat -> Nat -> Nat *)


\n:Nat. nat_ind  (__________________________________________) (* base case *)




                 ___________________________________________ (* inductive case *)

                 n  (* do induction on n *)
```

7. (10 points) Subtyping

The rules for STLC with pairs and subtyping are given on page 21 for your reference. The subtyping relations among a collection of types can be visualized compactly in picture form: we draw a graph so that `S <: T` iff we can get from `S` to `T` by following arrows in the graph (either directly or indirectly). For example, a picture for the types `Top*Top`, `A*Top`, `Top*(Top*Top)`, and `Top*(A*A)` would look like this (it happens to form a tree, but that is not necessary in general):



Suppose we have defined types `A` and `B` so that `A <: B`. Draw a picture for the following six types.

```
Top -> (A * B)
Top -> (A * A)
(B * A) -> (B * A)
(A * B) -> (B * A)
(B * A) -> Top
Top
```

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

$$\frac{}{\{\!\!\{\texttt{assn\_sub X a } Q\}\!\!\}\ \texttt{X := a}\ \{\!\!\{Q\}\!\!\}} \quad (\texttt{hoare\_asgn})$$

$$\frac{}{\{\!\!\{P\}\!\!\}\ \texttt{SKIP}\ \{\!\!\{P\}\!\!\}} \quad (\texttt{hoare\_skip})$$

$$\frac{\{\!\!\{P\}\!\!\}\ \texttt{c1}\ \{\!\!\{Q\}\!\!\} \qquad \{\!\!\{Q\}\!\!\}\ \texttt{c2}\ \{\!\!\{R\}\!\!\}}{\{\!\!\{P\}\!\!\}\ \texttt{c1; c2}\ \{\!\!\{R\}\!\!\}} \quad (\texttt{hoare\_seq})$$

$$\frac{\{\!\!\{P \wedge b\}\!\!\}\ \texttt{c1}\ \{\!\!\{Q\}\!\!\} \qquad \{\!\!\{P \wedge \sim b\}\!\!\}\ \texttt{c2}\ \{\!\!\{Q\}\!\!\}}{\{\!\!\{P\}\!\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\!\!\{Q\}\!\!\}} \quad (\texttt{hoare\_if})$$

$$\frac{\{\!\!\{P \wedge b\}\!\!\}\ \texttt{c}\ \{\!\!\{P\}\!\!\}}{\{\!\!\{P\}\!\!\}\ \texttt{WHILE b DO c END}\ \{\!\!\{P \wedge \sim b\}\!\!\}} \quad (\texttt{hoare\_while})$$

$$\frac{\{\!\!\{P'\}\!\!\}\ \texttt{c}\ \{\!\!\{Q'\}\!\!\} \qquad P \twoheadrightarrow P' \qquad Q' \twoheadrightarrow Q}{\{\!\!\{P\}\!\!\}\ \texttt{c}\ \{\!\!\{Q\}\!\!\}} \quad (\texttt{hoare\_consequence})$$

$$\frac{\{\!\!\{P'\}\!\!\}\ \texttt{c}\ \{\!\!\{Q\}\!\!\} \qquad P \twoheadrightarrow P'}{\{\!\!\{P\}\!\!\}\ \texttt{c}\ \{\!\!\{Q\}\!\!\}} \quad (\texttt{hoare\_consequence\_pre})$$

$$\frac{\{\!\!\{P\}\!\!\}\ \texttt{c}\ \{\!\!\{Q'\}\!\!\} \qquad Q' \twoheadrightarrow Q}{\{\!\!\{P\}\!\!\}\ \texttt{c}\ \{\!\!\{Q\}\!\!\}} \quad (\texttt{hoare\_consequence\_post})$$

Decorated programs

(a) `SKIP` is locally consistent if its precondition and postcondition are the same:

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

(b) The sequential composition of `c1` and `c2` is locally consistent (with respect to assertions `P` and `R`) if `c1` is locally consistent (with respect to `P` and `Q`) and `c2` is locally consistent (with respect to `Q` and `R`):

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

(c) An assignment is locally consistent if its precondition is the appropriate substitution of its postcondition:

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

(d) 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 /\ b` and `P /\ ~b` and if its "then" branch is locally consistent (with respect to `P /\ b` and `Q`) and its "else" branch is locally consistent (with respect to `P /\ ~b` and `Q`):

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

(e) A while loop with precondition `P` is locally consistent if its postcondition is `P /\ ~b` and if the pre- and postconditions of its body are exactly `P /\ b` and `P`:

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

(f) A pair of assertions separated by `->>` is locally consistent if the first implies the second (in all states):

```
{{ P }} ->>
{{ P' }}
```

STLC with Natural Numbers

Syntax

```
(* Types *)              (* Terms *)                  (* Values *)
T ::= Nat               t  ::= x                     v ::= 0
    | T -> T                 | t t                        | S v
                             | \x:T. t                    | \x:T. t
                             | 0
                             | S t
```

Small-step operational semantics

$$\frac{\text{value } v_2}{(\lambda x{:}T.\, t_{12})\ v_2 \Rightarrow [x := v_2]t_{12}} \quad \text{(ST\_AppAbs)}$$

$$\frac{t_1 \Rightarrow t_1'}{t_1\ t_2 \Rightarrow t_1'\ t_2} \quad \text{(ST\_App1)}$$

$$\frac{\text{value } v_1 \qquad t_2 \Rightarrow t_2'}{v_1\ t_2 \Rightarrow v_1\ t_2'} \quad \text{(ST\_App2)}$$

$$\frac{t_1 \Rightarrow t_1'}{S\ t_1 \Rightarrow S\ t_1'} \quad \text{(ST\_Succ)}$$

Typing

$$\frac{\Gamma\ x = T}{\Gamma \vdash x \in T}\ (T\_Var)$$

$$\frac{\Gamma,\ x{:}T11 \vdash t12 \in T12}{\Gamma \vdash \lambda x{:}T11.t12 \in T11 \to T12}\ (T\_Abs)$$

$$\frac{\Gamma \vdash t1 \in T11 \to T12 \qquad \Gamma \vdash t2 \in T11}{\Gamma \vdash t1\ t2 \in T12}\ (T\_App)$$

$$\frac{}{\Gamma \vdash 0 \in Nat}\ (T\_Zero)$$

$$\frac{\Gamma \vdash t \in Nat}{\Gamma \vdash S\ t \in Nat}\ (T\_Succ)$$

STLC with pairs and subtyping (excerpt)

Types

```
T ::=  ...
     | Top
     | T -> T
     | T * T
```

Subtyping relation

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

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

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

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

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