CIS 500 — Software Foundations
Final Exam

May 3, 2010

Name: _____

	Score
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	
Total	

Instructions

- This is a closed-book exam: you may not use any books or notes.
- You have 120 minutes to answer all of the questions.
- The exam is worth 120 points. However, questions vary significantly in difficulty, and the point value of a given question is not always exactly proportional to its difficulty. Do not spend too much time on any one question.
- Partial credit will be given. All correct answers are short. The back side of each page may be used as a scratch pad.
- Good luck!

Induction

1. (10 points) Three natural numbers are said to *frobnosticate* if they are in the following inductively defined relation:

```
Inductive frob : nat → nat → nat → Prop :=
  | frob_base : frob 0 0 0
  | frob_inc  : forall n1 n2 n3, frob n1 n2 n3 → frob n1 n2 (S n3)
  | frob1     : forall n1 n2 n3, frob n1 n2 n3 → frob (S n1) n2 (S n3)
  | frob2     : forall n1 n2 n3, frob n1 n2 n3 → frob n1 (S n2) (S n3).
```

   Give a careful informal proof that, if `x`, `y`, and `z` frobnosticate, then $\texttt{x} + \texttt{y} \leq \texttt{z}$.

Logic in Coq

2. (12 points) State the inductive definitions of **and**, **or**, and **exists**, as we gave them in **Logic.v**. (Don't worry about the exact names of the constructors. Just make up your own names if you don't remember the ones we used in Logic.v.)

```
Inductive and (P Q : Prop) : Prop :=
```

```
Inductive or (P Q : Prop) : Prop :=
```

```
Inductive ex (X:Type) (P : X→Prop) : Prop :=
```

Hoare Logic

3. (9 points) Recall the list-reversal function **rev**...

```
Fixpoint rev {X:Type} (l:list X) : list X :=
  match l with
  | nil      => []
  | cons h t => snoc (rev t) h
  end.
```

...and its imperative realization in IMP:

```
WHILE (BIsCons (AId X)) DO
  Y ::= ACons (AHead (AId X)) (AId Y);
  X ::= ATail (AId X)
END
```

Suppose that we want to carry out a Hoare-logic proof of correctness for this program. We begin by annotating it with pre- and post-conditions, plus a candidate invariant for the loop:

```
                          {{ X = l ∧ Y = [] }}
WHILE (BIsCons (AId X)) DO
  Y ::= ACons (AHead (AId X)) (AId Y);
  X ::= ATail (AId X)
                          {{ X ++ rev Y = rev l }}    // Invariant?
END
                          {{ Y = rev l }}
```

Is this invariant correct — i.e., are we going to be able to finish decorating the program so that all the local constraints imposed by the Hoare-logic rules are satisfied? If not, propose a correct invariant.

4. (9 points) For each of the following Hoare triples, give the *weakest precondition* that makes the triple valid.

(a)
```
    {{ ? }}
WHILE Y <= X DO
  X := X - 1
END
    {{ Y > X }}
```

(b)
```
    {{ ? }}
IF X > 3 THEN Z := X ELSE Z := Y FI
    {{ Z = W }}
```

(c)
```
    {{ ? }}
WHILE IsCons X DO
   N := N + 1;
   X := Tail(X)
DONE
    {{ X = [] ∧ N = length l }}
```

Simply Typed Lambda Calculus

The next two problems concern the STLC extended with natural numbers, pairs, and fixpoints, defined formally on page ?? in the Appendix.

5. (8 points) For each of the following assertions, write down a type `T` that makes the assertion true, or else state that there exists no such type.

   (a) `empty |- (\p : T. p.fst (p.snd 42)) : T → A`

   (b)
   ```
   exists U, exists V,
      empty |- (\f : U. \g : V. \x : A. g (f x)) : T
   ```

   (c) `empty |- fix (\n : Nat. pred n) : T`

   (d) `exists S, empty |- (\x:T. x 42 x) : S`

6. (6 points) Recall the typing and reduction rules for the **fix** operator in the simply typed lambda calculus.

$$\frac{\Gamma \vdash t_1 : T_1 \to T_1}{\Gamma \vdash \mathtt{fix}\ t_1 : T_1} \quad \text{(T\_Fix)}$$

$$\frac{t_1 \longrightarrow t_1'}{\mathtt{fix}\ t_1 \longrightarrow \mathtt{fix}\ t_1'} \quad \text{(ST\_Fix1)}$$

$$\frac{}{\mathtt{fix}\ (\lambda x{:}T_1.t_2) \longrightarrow [(\mathtt{fix}(\lambda x{:}T_1.t_2))/x]\ t_2} \quad \text{(ST\_FixAbs)}$$

Consider the following Coq **Fixpoint** definition of exponentiation:

```
Fixpoint pow (base:Nat) (exp:Nat) : Nat :=
  match exp with
  | O      => 1
  | S exp' => base * pow base exp'
  end.
```

Translate this into a term in the STLC (with natural numbers and **fix**).

References

The definition of the STLC extended with references can be found on page ?? of the Appendix, together with critical auxiliary definitions such as the `well_typed_store` and `extends` relations.

7. (12 points) For each of the following stores `s`:

   - First, write down a store typing corresponding to the given store. For example, the store typing corresponding to the store `[6,7]` would be `[Nat, Nat]`.
   - Then write down a term `t` in the simply typed lambda calculus with references, such that

     ```
     t / [] -->* v / s
     ```

     for some value `v`. For example, if the desired ending store were

     ```
     [6, 7]
     ```

     one possible program to build this store would be:

     ```
     let x = ref 6 in
     let y = ref 7 in
     unit
     ```

     (We've used the `let` derived form here, to make the program more readable. Feel free to do the same.)

(a) `[5, loc 0]`

(b) `[5, loc 2, 4]`

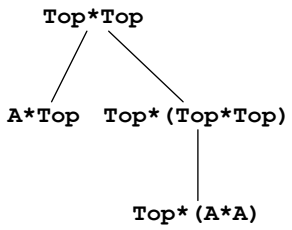
(c) `[(\ x : Unit . !(loc 0) unit)]`

8. (8 points) Write out the *statements* of the progress and preservation theorems for the STLC with references. (Just the statements—no proofs.)

Subtyping

The remaining questions on the exam concern the simply typed lambda calculus extended with products and subtyping, described formally on page ?? in the Appendix.

9. (6 points) The subtyping relations among a collection of types can be visuaized compactly in tree form: we draw the tree so that `S <: T` iff `S` is below `T` (either directly or indirectly). For example, a tree for the types `Top*Top`, `A*Top`, `Top*(Top*Top)`, and `Top*(A*A)` would look like this:



Draw a tree for the following six types.

```
(Top → B) → A
Top
A → Top
(Top → B) → Top
A → (B → B)
(B → B) → Top
```

10. (10 points) Suppose we wanted to add *both* subtyping and references to the simply typed lambda calculus. Following the pattern we've used for the other type constructors (arrow, products, etc.), we'd need to think about what subtyping rule we'd want for reference types.

    Here are two *incorrect* versions of a subtyping rule for reference types. For each rule, explain in one or two sentences why it is incorrect, and give an example of a program that would be well typed using this rule but would get stuck when executed.

    (a)
    ```
        T2 <: T1
    ----------------          (S_Ref_Wrong1)
    Ref T1 <: Ref T2
    ```

    (b)
    ```
        T1 <: T2
    ----------------          (S_Ref_Wrong2)
    Ref T1 <: Ref T2
    ```

11. (12 points) This problem asks you to consider the possible consequences if we add to this language (STLC with products and subtyping) a reduction rule of the form

```
----------                    (ST_Unit)
unit --> ?
```

where `?` is some term.

For each of the following three properties, either give a term which can be put in the place of the `?` in order to *break* the given property, or explain why there is no such term.

(a) Preservation

(b) Progress

(c) Normalization of well-typed terms

12. (18 points) Recall the *progress* theorem for the STLC with products and subtyping.

> **Theorem:** For any term `t` and type `T`, if `empty |- t : T` then either `t` is a value or else `t --> t'` for some term `t'`.

Write a careful informal proof of this theorem in the space below. You may use the following lemmas:

> **Lemma [Canonical forms of arrow types]:** If `Gamma |- s : T1→T2` and `s` is a value, then `s = tm_abs x S1 s2` for some `x`, `S1`, and `s2`.

> **Lemma [Canonical forms of product types]:** If `Gamma |- s : T1*T2` and `s` is a value, then `s = tm_pair s1 s2` for some `s1` and `s2`.

Appendix

IMP (with lists)

```
Inductive val : Type :=
| VNat : nat → val
| VList : list nat → val.

Definition state := id → val.

Definition empty_state : state := fun _ => VNat 0.
Definition update (st : state) (V:id) (n : val) : state :=
  fun V' => if beq_id V V' then n else st V'.

Definition asnat (v : val) : nat :=
  match v with
  | VNat n => n
  | VList _ => 0
  end.

Definition aslist (v : val) : list nat :=
  match v with
  | VNat n => []
  | VList xs => xs
  end.

Inductive aexp : Type :=
  | ANum : nat → aexp
  | AId : id → aexp
  | APlus : aexp → aexp → aexp
  | AMinus : aexp → aexp → aexp
  | AMult : aexp → aexp → aexp
  | AHead : aexp → aexp
  | ATail : aexp → aexp
  | ACons : aexp → aexp → aexp
  | ANil  : aexp.

Definition tail (l : list nat) :=
  match l with
  | x::xs => xs
  | [] => []
  end.

Definition head (l : list nat) :=
  match l with
  | x::xs => x
  | [] => 0
  end.
```

```
Fixpoint aeval (st : state) (e : aexp) : val :=
  match e with
  | ANum n => VNat n
  | AId i => st i
  | APlus a1 a2   => VNat (asnat (aeval st a1) + asnat (aeval st a2))
  | AMinus a1 a2  => VNat (asnat (aeval st a1) - asnat (aeval st a2))
  | AMult a1 a2   => VNat (asnat (aeval st a1) * asnat (aeval st a2))
  | ATail a => VList (tail (aslist (aeval st a)))
  | AHead a => VNat (head (aslist (aeval st a)))
  | ACons a1 a2 => VList (asnat (aeval st a1) :: aslist (aeval st a2))
  | ANil => VList []
  end.

Inductive bexp : Type :=
  | BTrue : bexp
  | BFalse : bexp
  | BEq : aexp → aexp → bexp
  | BLe : aexp → aexp → bexp
  | BNot : bexp → bexp
  | BAnd : bexp → bexp → bexp
  | BIsCons : aexp → bexp.

Fixpoint beval (st : state) (e : bexp) : bool :=
  match e with
  | BTrue       => true
  | BFalse      => false
  | BEq a1 a2   => beq_nat (asnat (aeval st a1)) (asnat (aeval st a2))
  | BLe a1 a2   => ble_nat (asnat (aeval st a1)) (asnat (aeval st a2))
  | BNot b1     => negb (beval st b1)
  | BAnd b1 b2  => andb (beval st b1) (beval st b2)
  | BIsCons a    => match aslist (aeval st a) with
                      | _::_ => true
                      | [] => false
                    end
  end.

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  (at level 10).
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).
```

```
Inductive ceval : state → com → state → Prop :=
  | E_Skip : forall st,
      SKIP / st ==> st
  | E_Ass  : forall st a1 n l,
      aeval st a1 = n →
      (l ::= a1) / st ==> (update st l 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 st c1 st').
```

STLC with Naturals, Pairs, and Fixpoints

```
Inductive ty : Type :=
  | ty_base  : id → ty
  | ty_arrow : ty → ty → ty
  | ty_unit  : ty
  | ty_prod  : ty → ty → ty
  | ty_nat   : ty.

Inductive tm : Type :=
  | tm_var  : id → tm
  | tm_app  : tm → tm → tm
  | tm_abs  : id → ty → tm → tm
  | tm_unit : tm
  | tm_pair : tm → tm → tm
  | tm_fst  : tm → tm
  | tm_snd  : tm → tm
  | tm_nat  : nat → tm
  | tm_succ : tm → tm
  | tm_pred : tm → tm
  | tm_mult : tm → tm → tm
  | tm_if0  : tm → tm → tm → tm
  | tm_fix  : tm → tm.

Inductive value : tm → Prop :=
  | v_abs : forall x T11 t12,
      value (tm_abs x T11 t12)
  | v_unit :
      value tm_unit
  | v_pair : forall v1 v2,
      value v1 →
      value v2 →
      value (tm_pair v1 v2)
  | v_nat : forall n1,
      value (tm_nat n1).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : forall x T11 t12 v2,
        value v2 →
        (tm_app (tm_abs x T11 t12) v2) --> (subst x v2 t12)
  | ST_App1 : forall t1 t1' t2,
        t1 --> t1' →
        (tm_app t1 t2) --> (tm_app t1' t2)
  | ST_App2 : forall v1 t2 t2',
        value v1 →
        t2 --> t2' →
        (tm_app v1 t2) --> (tm_app v1 t2')
  | ST_Pair1 : forall t1 t1' t2,
       t1 --> t1' →
       (tm_pair t1 t2) --> (tm_pair t1' t2)
  | ST_Pair2 : forall v1 t2 t2',
       value v1 →
       t2 --> t2' →
       (tm_pair v1 t2) --> (tm_pair v1 t2')
  | ST_Fst1 : forall t1 t1',
       t1 --> t1' →
       (tm_fst t1) --> (tm_fst t1')
```

```
  | ST_FstPair : forall v1 v2,
       value v1 →
       value v2 →
       (tm_fst (tm_pair v1 v2)) --> v1
  | ST_Snd1 : forall t1 t1',
       t1 --> t1' →
       (tm_snd t1) --> (tm_snd t1')
  | ST_SndPair : forall v1 v2,
       value v1 →
       value v2 →
       (tm_snd (tm_pair v1 v2)) --> v2
  | ST_Succ1 : forall t1 t1',
      t1 --> t1' →
      (tm_succ t1) --> (tm_succ t1')
  | ST_SuccNat : forall n1,
      (tm_succ (tm_nat n1)) --> (tm_nat (S n1))
  | ST_Pred : forall t1 t1',
      t1 --> t1' →
      (tm_pred t1) --> (tm_pred t1')
  | ST_PredNat : forall n1,
      (tm_pred (tm_nat n1)) --> (tm_nat (pred n1))
  | ST_Mult1 : forall t1 t1' t2,
      t1 --> t1' →
      (tm_mult t1 t2) --> (tm_mult t1' t2)
  | ST_Mult2 : forall v1 t2 t2',
      value v1 →
      t2 --> t2' →
      (tm_mult v1 t2) --> (tm_mult v1 t2')
  | ST_MultNats : forall n1 n2,
      (tm_mult (tm_nat n1) (tm_nat n2)) --> (tm_nat (mult n1 n2))
  | ST_If01 : forall t1 t1' t2 t3,
      t1 --> t1' →
      (tm_if0 t1 t2 t3) --> (tm_if0 t1' t2 t3)
  | ST_If0Zero : forall t2 t3,
      (tm_if0 (tm_nat 0) t2 t3) --> t2
  | ST_If0Nonzero : forall n t2 t3,
      (tm_if0 (tm_nat (S n)) t2 t3) --> t3
  | ST_Fix1 : forall t1 t1',
      t1 --> t1' →
      (tm_fix t1) --> (tm_fix t1')
  | ST_FixAbs : forall x T1 t12,
      (tm_fix (tm_abs x T1 t12)) -->
      (subst x (tm_fix (tm_abs x T1 t12)) t12)

where "t1 '-->' t2" := (step t1 t2).

Inductive has_type : context → tm → ty → Prop :=
  | T_Var : forall Gamma x T,
      Gamma x = Some T →
      has_type Gamma (tm_var x) T
  | T_Abs : forall Gamma x T11 T12 t12,
      has_type (extend Gamma x T11) t12 T12 →
      has_type Gamma (tm_abs x T11 t12) (ty_arrow T11 T12)
  | T_App : forall T1 T2 Gamma t1 t2,
      has_type Gamma t1 (ty_arrow T1 T2) →
      has_type Gamma t2 T1 →
```

```
    has_type Gamma (tm_app t1 t2) T2
| T_Unit : forall Gamma,
    has_type Gamma tm_unit ty_unit
| T_Pair : forall Gamma t1 t2 T1 T2,
    has_type Gamma t1 T1 →
    has_type Gamma t2 T2 →
    has_type Gamma (tm_pair t1 t2) (ty_prod T1 T2)
| T_Fst : forall Gamma t T1 T2,
    has_type Gamma t (ty_prod T1 T2) →
    has_type Gamma (tm_fst t) T1
| T_Snd : forall Gamma t T1 T2,
    has_type Gamma t (ty_prod T1 T2) →
    has_type Gamma (tm_snd t) T2
| T_Nat : forall Gamma n1,
    has_type Gamma (tm_nat n1) ty_nat
| T_Succ : forall Gamma t1,
    has_type Gamma t1 ty_nat →
    has_type Gamma (tm_succ t1) ty_nat
| T_Pred : forall Gamma t1,
    has_type Gamma t1 ty_nat →
    has_type Gamma (tm_pred t1) ty_nat
| T_Mult : forall Gamma t1 t2,
    has_type Gamma t1 ty_nat →
    has_type Gamma t2 ty_nat →
    has_type Gamma (tm_mult t1 t2) ty_nat
| T_If0 : forall Gamma t1 t2 t3 T1,
    has_type Gamma t1 ty_nat →
    has_type Gamma t2 T1 →
    has_type Gamma t3 T1 →
    has_type Gamma (tm_if0 t1 t2 t3) T1
| T_Fix : forall Gamma t1 T1,
    has_type Gamma t1 (ty_arrow T1 T1) →
    has_type Gamma (tm_fix t1) T1.
```

STLC with References

```
Inductive ty : Type :=
  | ty_nat   : ty
  | ty_arrow : ty → ty → ty
  | ty_unit  : ty
  | ty_ref   : ty → ty.

Inductive tm  : Type :=
  | tm_var    : id → tm
  | tm_app    : tm → tm → tm
  | tm_abs    : id → ty → tm → tm
  | tm_nat    : nat → tm
  | tm_succ   : tm → tm
  | tm_pred   : tm → tm
  | tm_mult   : tm → tm → tm
  | tm_if0    : tm → tm → tm → tm
  | tm_unit   : tm
  | tm_ref    : tm → tm
  | tm_deref  : tm → tm
  | tm_assign : tm → tm → tm
  | tm_loc    : nat → tm.

Inductive value : tm → Prop :=
  | v_abs  : forall x T t,
      value (tm_abs x T t)
  | v_nat : forall n,
      value (tm_nat n)
  | v_unit :
      value tm_unit
  | v_loc : forall l,
      value (tm_loc l).

Inductive step : tm * store → tm * store → Prop :=
  | ST_AppAbs : forall x T t12 v2 st,
        value v2 →
        tm_app (tm_abs x T t12) v2 / st --> subst x v2 t12 / st
  | ST_App1 : forall t1 t1' t2 st st',
        t1 / st --> t1' / st' →
        tm_app t1 t2 / st --> tm_app t1' t2 / st'
  | ST_App2 : forall v1 t2 t2' st st',
        value v1 →
        t2 / st --> t2' / st' →
        tm_app v1 t2 / st --> tm_app v1 t2'/ st'
  | ST_SuccNat : forall n st,
        tm_succ (tm_nat n) / st --> tm_nat (S n) / st
  | ST_Succ : forall t1 t1' st st',
        t1 / st --> t1' / st' →
        tm_succ t1 / st --> tm_succ t1' / st'
  | ST_PredNat : forall n st,
        tm_pred (tm_nat n) / st --> tm_nat (pred n) / st
  | ST_Pred : forall t1 t1' st st',
        t1 / st --> t1' / st' →
        tm_pred t1 / st --> tm_pred t1' / st'
  | ST_MultNats : forall n1 n2 st,
        tm_mult (tm_nat n1) (tm_nat n2) / st --> tm_nat (mult n1 n2) / st
  | ST_Mult1 : forall t1 t2 t1' st st',
```

```
        t1 / st --> t1' / st' →
        tm_mult t1 t2 / st --> tm_mult t1' t2 / st'
  | ST_Mult2 : forall v1 t2 t2' st st',
        value v1 →
        t2 / st --> t2' / st' →
        tm_mult v1 t2 / st --> tm_mult v1 t2' / st'
  | ST_If0 : forall t1 t1' t2 t3 st st',
        t1 / st --> t1' / st' →
        tm_if0 t1 t2 t3 / st --> tm_if0 t1' t2 t3 / st'
  | ST_If0_Zero : forall t2 t3 st,
        tm_if0 (tm_nat 0) t2 t3 / st --> t2 / st
  | ST_If0_Nonzero : forall n t2 t3 st,
        tm_if0 (tm_nat (S n)) t2 t3 / st --> t3 / st
  | ST_RefValue : forall v1 st,
        value v1 →
        tm_ref v1 / st --> tm_loc (length st) / snoc st v1
  | ST_Ref : forall t1 t1' st st',
        t1 / st --> t1' / st' →
        tm_ref t1 /  st --> tm_ref t1' /  st'
  | ST_DerefLoc : forall st l,
        l < length st →
        tm_deref (tm_loc l) / st --> store_lookup l st / st
  | ST_Deref : forall t1 t1' st st',
        t1 / st --> t1' / st' →
        tm_deref t1 / st --> tm_deref t1' / st'
  | ST_Assign : forall v2 l st,
        value v2 →
        l < length st →
        tm_assign (tm_loc l) v2 / st --> tm_unit / replace l v2 st
  | ST_Assign1 : forall t1 t1' t2 st st',
        t1 / st --> t1' / st' →
        tm_assign t1 t2 / st --> tm_assign t1' t2 / st'
  | ST_Assign2 : forall v1 t2 t2' st st',
        value v1 →
        t2 / st --> t2' / st' →
        tm_assign v1 t2 / st --> tm_assign v1 t2' / st'

where "t1 '/' st1 '-->' t2 '/' st2" := (step (t1,st1) (t2,st2)).

Inductive has_type : context → store_ty → tm → ty → Prop :=
  | T_Var : forall Gamma ST x T,
      Gamma x = Some T →
      has_type Gamma ST (tm_var x) T
  | T_Abs : forall Gamma ST x T11 T12 t12,
      has_type (extend Gamma x T11) ST t12 T12 →
      has_type Gamma ST (tm_abs x T11 t12) (ty_arrow T11 T12)
  | T_App : forall T1 T2 Gamma ST t1 t2,
      has_type Gamma ST t1 (ty_arrow T1 T2) →
      has_type Gamma ST t2 T1 →
      has_type Gamma ST (tm_app t1 t2) T2
  | T_Nat : forall Gamma ST n,
      has_type Gamma ST (tm_nat n) ty_nat
  | T_Succ : forall Gamma ST t1,
      has_type Gamma ST t1 ty_nat →
      has_type Gamma ST (tm_succ t1) ty_nat
  | T_Pred : forall Gamma ST t1,
```

```
      has_type Gamma ST t1 ty_nat →
      has_type Gamma ST (tm_pred t1) ty_nat
  | T_Mult : forall Gamma ST t1 t2,
      has_type Gamma ST t1 ty_nat →
      has_type Gamma ST t2 ty_nat →
      has_type Gamma ST (tm_mult t1 t2) ty_nat
  | T_If0 : forall Gamma ST t1 t2 t3 T,
      has_type Gamma ST t1 ty_nat →
      has_type Gamma ST t2 T →
      has_type Gamma ST t3 T →
      has_type Gamma ST (tm_if0 t1 t2 t3) T
  | T_Unit : forall Gamma ST,
      has_type Gamma ST tm_unit ty_unit
  | T_Loc : forall Gamma ST l,
      l < length ST →
      has_type Gamma ST (tm_loc l) (ty_ref (store_ty_lookup l ST))
  | T_Ref : forall Gamma ST t1 T1,
      has_type Gamma ST t1 T1 →
      has_type Gamma ST (tm_ref t1) (ty_ref T1)
  | T_Deref : forall Gamma ST t1 T11,
      has_type Gamma ST t1 (ty_ref T11) →
      has_type Gamma ST (tm_deref t1) T11
  | T_Assign : forall Gamma ST t1 t2 T11,
      has_type Gamma ST t1 (ty_ref T11) →
      has_type Gamma ST t2 T11 →
      has_type Gamma ST (tm_assign t1 t2) ty_unit.

Definition store_well_typed (Gamma:context) (ST:store_ty) (st:store) :=
  length ST = length st ∧
  (forall l,
     l < length st →
     has_type Gamma ST (store_lookup l st) (store_ty_lookup l ST)).

Inductive extends : store_ty → store_ty → Prop :=
  | extends_nil  : forall ST', extends ST' nil
  | extends_cons : forall x ST' ST, extends ST' ST → extends (x::ST') (x::ST).
```

STLC with Products and Subtyping

```
Inductive ty : Type :=
  | ty_top   : ty
  | ty_arrow : ty → ty → ty
  | ty_unit  : ty
  | ty_prod  : ty → ty → ty.

Inductive tm : Type :=
  | tm_var : id → tm
  | tm_app : tm → tm → tm
  | tm_abs : id → ty → tm → tm
  | tm_unit : tm
  | tm_pair : tm → tm → tm
  | tm_fst : tm → tm
  | tm_snd : tm → tm.

Inductive value : tm → Prop :=
  | v_abs : forall x T t,
      value (tm_abs x T t)
  | v_unit :
      value tm_unit
  | v_pair : forall v1 v2,
      value v1 →
      value v2 →
      value (tm_pair v1 v2).

Inductive step : tm → tm → Prop :=
  | ST_AppAbs : forall x T t12 v2,
         value v2 →
         (tm_app (tm_abs x T t12) v2) --> (subst x v2 t12)
  | ST_App1 : forall t1 t1' t2,
         t1 --> t1' →
         (tm_app t1 t2) --> (tm_app t1' t2)
  | ST_App2 : forall v1 t2 t2',
         value v1 →
         t2 --> t2' →
         (tm_app v1 t2) --> (tm_app v1  t2')
  | ST_Pair1 : forall t1 t1' t2,
        t1 --> t1' →
        (tm_pair t1 t2) --> (tm_pair t1' t2)
  | ST_Pair2 : forall v1 t2 t2',
        value v1 →
        t2 --> t2' →
        (tm_pair v1 t2) --> (tm_pair v1 t2')
  | ST_Fst1 : forall t t',
        t --> t' →
        tm_fst t --> tm_fst t'
  | ST_FstPair : forall v1 v2,
        value v1 →
        value v2 →
        (tm_fst (tm_pair v1 v2)) --> v1
  | ST_Snd1 : forall t t',
        t --> t' →
        tm_snd t --> tm_snd t'
  | ST_SndPair : forall v1 v2,
        value v1 →
```

```
        value v2 →
        tm_snd (tm_pair v1 v2) --> v2

where "t1 '-->' t2" := (step t1 t2).

Inductive subtype : ty → ty → Prop :=
  | S_Refl : forall T,
    subtype T T
  | S_Trans : forall S U T,
    subtype S U →
    subtype U T →
    subtype S T
  | S_Top : forall S,
    subtype S ty_top
  | S_Arrow : forall S1 S2 T1 T2,
    subtype T1 S1 →
    subtype S2 T2 →
    subtype (ty_arrow S1 S2) (ty_arrow T1 T2)
  | S_Pair : forall S1 S2 T1 T2,
    subtype S1 T1 →
    subtype S2 T2 →
    subtype (ty_prod S1 S2) (ty_prod T1 T2).

Inductive has_type : context → tm → ty → Prop :=
  | T_Var : forall Gamma x T,
      Gamma x = Some T →
      has_type Gamma (tm_var x) T
  | T_Abs : forall Gamma x T11 T12 t12,
      has_type (extend Gamma x T11) t12 T12 →
      has_type Gamma (tm_abs x T11 t12) (ty_arrow T11 T12)
  | T_App : forall T1 T2 Gamma t1 t2,
      has_type Gamma t1 (ty_arrow T1 T2) →
      has_type Gamma t2 T1 →
      has_type Gamma (tm_app t1 t2) T2
  | T_Unit : forall Gamma,
      has_type Gamma tm_unit ty_unit
  | T_Pair : forall Gamma t1 t2 T1 T2,
      has_type Gamma t1 T1 →
      has_type Gamma t2 T2 →
      has_type Gamma (tm_pair t1 t2) (ty_prod T1 T2)
  | T_Fst : forall Gamma t T1 T2,
      has_type Gamma t (ty_prod T1 T2) →
      has_type Gamma (tm_fst t) T1
  | T_Snd : forall Gamma t T1 T2,
      has_type Gamma t (ty_prod T1 T2) →
      has_type Gamma (tm_snd t) T2
  | T_Sub : forall Gamma t S T,
      has_type Gamma t S →
      subtype S T →
      has_type Gamma t T.
```