

# CIS 500 — Software Foundations

## Final

(Advanced version)

April 30, 2013

Answer key

1. (12 points) Recall the definition of the `fold` function on lists (given in the appendix, page ??). Use it to define a function `sum_list_list` that, given a list `(list nat)`, returns the sum of all the `nats` in it. E.g.

```
sum_list_list [[1,3],[5,6]] = 15
```

(Note that the first line we've given you starts with `Definition`, not `Fixpoint`: your solution should be non-recursive.)

```
Definition sum_list_list (l : list (list nat)) : nat :=
```

*Answer:*

```
fold (fun z => fun h => fold plus h z) 0 l.
```

*Grading scheme: 6 points off if not recursive or missing fold, 3 points off if not accumulating, 1,2 or 3 points off for other errors or redundancies.*

2. (12 points) Complete the definition at the bottom of the page of an inductive proposition `count` that counts the number of elements of a list satisfying some predicate `P`. For example, if we define

```
Definition iszero (n : nat) : Prop :=  
  n = 0.
```

then the propositions

```
count iszero [] 0  
count iszero [1,2,3] 0  
count iszero [0,1,2,3] 1  
count iszero [1,0,0,2,3,0] 3
```

should all be provable, whereas the propositions

```
count iszero [1,2,3] 3  
count iszero [0,0] 4  
count iszero [] 1
```

should not be provable.

```
Inductive count {X : Type} (P : X -> Prop) : list X -> nat -> Prop :=
```

Answer:

```
R0 : count P nil 0
| Rno : forall l n x, count P l n -> ~(P x) -> count P (x::l) n
| Ryes : forall l n x, count P l n -> P x -> count P (x::l) (S n).
```

Grading scheme: Roughly 4 points for each correct constructor. 1 or 2 points taken off on each one for minor mistakes.

3. (10 points) For each of the types below, write a Coq expression that has that type. (For example, if the type given were `nat->nat`, you might write `fun x:nat, x+5`.) The definitions of `/\` and `\/` are given in the appendix (page ??) for reference.

(a) `forall X Y, (X -> Y) -> X -> list Y`

*Possible answers:*

```
fun X (f: X -> Y) (x : X) => [f x]
fun X (f: X -> Y) (x : X) => nil
```

...

(b) `forall (X : Type), (X -> Prop) -> Prop`

*Possible answers:*

```
fun (X : Type) => fun (P : X -> Prop) => True
fun (X : Type) => fun (P : X -> Prop) => forall x : X, P x
```

...

(c) `forall (X Y : Prop), X -> Y -> (X \/ Y) /\ Y`

*Possible answers:*

```
fun X Y (HX : X) (HY : Y) =>
  conj (X \/ Y) Y (or_introl X Y HX) HY.
```

...

(d) `forall (X Y W Z : Prop), (X /\ Y) \/ (W /\ Z) -> (X \/ Z)`

*Possible answers:*

```
fun X Y (H : (X /\ Y) \/ (W /\ Z)) =>
  match H with
  | or_introl HXY => ( match HXY with
    | conj HX HY => or_introl X Z HX
    end)
  | or_intror HWZ => ( match HWZ with
    | conj HW HZ => or_intror X Z HZ
    end)
  end
```

...

*Grading scheme: 2 points for each one of (a) and (b), 3 points for the other ones. One point taken off for forgetting to bind the types, additional points taken off for other mistakes.*

4. (12 points) The following Imp program sets R to the sum of the initial values of X and Y modulo the initial value of Z:

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

Your task is to fill in assertions to make this a well-decorated program (see page ?? of the appendix), relative to an appropriate and post-condition. Please fill in annotations in the spaces provided below.

```
{ { True } } ->>
{ { (X + Y) mod Z = (X + Y) mod Z } }
R ::= X + Y;
{ { R mod Z = (X + Y) mod Z } }
WHILE Z <= R DO
  { { R mod Z = (X + Y) mod Z /\ Z <= R } } ->> (1)
  { { (R - Z) mod Z = (X + Y) mod Z } }
  R ::= R - Z
  { { R mod Z = (X + Y) mod Z } }
END
{ { R mod Z = (X + Y) mod Z /\ ~(Z <= R) } } ->> (2)
{ { R = (X + Y) mod Z } }
```

Finally, please mark (by circling the associated arrow  $\rightarrow$  and writing either 1 or 2 or both, as appropriate) any places where the following facts about modular arithmetic are needed to validate the correctness of the decorations:

$$(1) \quad a \bmod z = (a + z) \bmod z$$

$$(2) \quad a < z \rightarrow (a \bmod z) = a$$

*Grading scheme: One point for each pair of consistent annotations, one point for each correctly applied lemma, plus 4 points for finding the correct loop invariant*

5. (10 points) A *triangular number* is one that can be expressed as a sum of the form

$$1 + 2 + 3 + \dots + n$$

for some  $n$ . In Coq notation:

```
Fixpoint tri (n : nat) : nat :=
  match n with
  0 => 0
```

```
| S n' => n + (tri n')
end.
```

```
Definition triangular (t : nat) : Prop :=
  exists n, t = tri n.
```

The following Imp program will terminate only when the initial value of the variable T is a triangular number.

```
{ { True } }
X ::= 0;
A ::= 0;
WHILE A <> T DO
  X ::= X + 1;
  A ::= A + X;
  { { I } }
END
{ { triangular T } }
```

In the space below, write a suitable invariant for this loop — i.e., an assertion I such that, if we add I as the annotation at the end of the loop as shown, we can fill in annotations on the rest of the program to make it well decorated. (The spaces for these additional annotations are not shown, and you do not need to provide them.)

*Answer:*     I        =        A = tri X

*Grading scheme:* 2 to 6 points for wrong invariant. 1 or 2 points off for redundant invariants.

6. (14 points) Consider the following two programs:

```
P1 = WHILE b DO c END
P2 = IFB b THEN c ELSE SKIP FI; P1
```

Give a careful informal proof that P1 and P2 are equivalent.

*Answer:*

By the definition of program equivalence, we need to show that

forall s s', P1 / s || s' <-> P2 s || s'

There are two cases to consider:

- Suppose we have P1 / s || s' for some states s, s'. Inverting that hypothesis gives us two possible cases:
  - beval s b = false and s = s'. By the E\_Skip rule, we know that SKIP / s || s. Thus, the E\_IfFalse rule gives us IFB b THEN c ELSE SKIP FI / s || s. Again, we can apply the E\_WhileEnd rule to show that WHILE b DO c END / s || s. Both combined with the E\_Seq rule imply P2 / s || s, which is what we wanted.

- Otherwise, `beval s b = true`, and there is some `s''` such that `c / s || s''` and `P1 / s'' || s'`. The first and second assumptions together with the `E_IfTrue` rule imply `IFB b THEN c ELSE SKIP s || s''`. Once again, combining this with our third assumption and the `E_Seq` rule yields `P2 / s || s'`, which is what we wanted
- Now, let's prove the other direction. Suppose that `P2 / s || s'`. By the shape of `P2`, inverting this assumption yields a state `s''` such that `IFB b THEN c ELSE SKIP / s || s''` and `P1 / s'' || s'`. We can invert the first assumption again, which leads us to two cases to consider:
  - `beval s b = true` and `c / s || s''`. We need to show `P1 / s || s'`, which can be obtained by applying the `E_WhileLoop` rule with `s''` as an intermediate state.
  - `beval s b = false` and `SKIP / s || s''`. The second assumption allows us to conclude that `s = s''` by inversion. Thus, we know that `P1 / s || s'`. But this is exactly what we wanted to show.

*Grading scheme: 13-14 points for perfect or nearly perfect proofs. 9-12 points for mostly correct proofs with more significant deficiencies. 1-8 points for proofs with major issues.*

7. (15 points) Suppose  $L$  is some variant of STLC with the same syntax but in which the typing and single-step reduction rules have been changed in some way (by adding, removing, or changing rules). Consider the following schematic statement:

“If the (A) theorem (B) for  $L$ , then changing  $L$ 's (C) relation by (D) a rule might cause it to (E).”

If we substitute “preservation” for (A), “holds” for (B), “typing” for (C), “adding” for (D), and “fail” for (E), we obtain the statement “If the *preservation* theorem *holds* for  $L$ , then changing  $L$ 's *typing* relation by *adding* a rule might cause it to *fail*,” which happens to be true: adding a rule to the typing relation of a language like the STLC can sometimes destroy the preservation property.

Fill in the last column of the following table to indicate whether the statement obtained by similarly substituting the values in columns (A) to (E) is true or false. (We've done the first one for you.)

(A)	(B)	(C)	(D)	(E)	T/F
preservation	holds	typing	adding	fail	T
preservation	holds	single-step reduction	removing	fail	F
preservation	holds	typing	removing	fail	T
preservation	holds	single-step reduction	adding	fail	T
progress	holds	typing	removing	fail	F
progress	holds	single-step reduction	removing	fail	T
progress	holds	typing	adding	fail	T
progress	holds	single-step reduction	adding	fail	F
preservation	fails	typing	removing	hold	T
preservation	fails	single-step reduction	removing	hold	T
preservation	fails	typing	adding	hold	T
preservation	fails	single-step reduction	adding	hold	F
progress	fails	typing	removing	hold	T
progress	fails	single-step reduction	removing	hold	F
progress	fails	typing	adding	hold	F
progress	fails	single-step reduction	adding	hold	T

8. (10 points) Consider the following propositions about typing in STLC. For each one, either give values for the existentially quantified variables to make the term typeable, or write “not typeable” and briefly explain why not.

- (a) There exist types  $T$  and  $T1$  such that

$$\text{empty} \vdash (\lambda x:T. \text{if } x \text{ then true else } x) \in T1$$

*Answers:*

True. E.g., we have  $T = \text{bool}$ ,  $T1 = \text{bool} \rightarrow \text{bool}$

- (b) For some types  $T2$  and  $T1$ ,

$$y:T2 \vdash (\lambda x:T1. x (y (y x) y)) \in \text{Bool}$$

*Answers:*

Not typeable.

- (c) For all types  $T$ , there exist types  $T1$  and  $T2$  such that

$$y:T2, x:T1 \vdash (y (x \text{ true}) x) \in T$$

*Answers:*

$T1 = \text{bool} \rightarrow \text{bool}$ ,  $T2 : \text{bool} \rightarrow (\text{bool} \rightarrow \text{bool}) \rightarrow T$

- (d) For all types  $T2$  and  $T1$ , there exists a type  $T$  such that

$$y:T2 \vdash (\lambda x:T1. \text{if } y \text{ then true else } x) \in T$$

*Possible answers:* Not typeable,  $T2 = \text{bool}$ ,  $T1 = \text{bool}$ .

- (e) There exist types  $T1$  and  $T2$  such that

`empty |- (λy:T2. λx:T1. if x then y (y (y x)) else x) ∈ Bool`

*Possible answers:* Not typeable.

*Grading scheme:* 2 points for each correct answer. No points for incorrect answer. One point for incomplete answer (Not typable, but not stating the reason.) One point off for other mistakes.

9. (20 points) Your job in this problem is to fill in part of the proof of a lemma needed for the Preservation theorem in the STLC. (Note that the language we are considering here is the *standard SLTC*, without subtyping—see page ?? of the appendix.)

The lemma in question is the one stating that substitution preserves typing in STLC:

```
Theorem substitution_preserves_typing :
  forall (t t' : tm) (x : id) (Γ : context) (T T' : ty),
    Γ, x:T' |- t ∈ T ->
    empty |- t' ∈ T' ->
    Γ |- [x := t'] t ∈ T.
```

On the next two pages, you will find an informal proof of this theorem, with two cases on the second page omitted. Fill in the argument for the missing cases.

Your may find it useful at some point(s) to invoke the following lemma (which you do not need to prove):

```
Lemma context_invariance : forall Γ Γ' t T,
  Γ |- t ∈ T ->
  (forall x, appears_free_in x t -> Γ x = Γ' x) ->
  Γ' |- t ∈ T.
```

**Proof:** By induction on  $t$ . We have the following cases to consider:

- $t = \text{true}$ . In this case, we have  $[x := t'] \text{true} = \text{true}$  by the definition of substitution. By the rule `T_True` the conclusion follows.
- $t = \text{false}$ . Similar.
- $t = t1 \ t2$ . In this case, we have

$$\begin{aligned} [x := t'] (t1 \ t2) \\ = ([x := t'] t1) ([x := t'] t2) \end{aligned}$$

by the definition of substitution. By inversion on `Gamma, x:T' |- t ∈ T` we know that there exists a `T1` such that

$$\Gamma, x:T' |- t1 \in T1 \rightarrow T \quad (1)$$

$$\Gamma, x:T' |- t2 \in T1 \quad (2)$$

In order to apply rule `T_App` to build the desired derivation, we have to show the following conditions:

- $\Gamma \vdash [x := t'] t1 \in T1 \rightarrow T$ . This follows from (1) by the IH.

–  $\Gamma \vdash [x := t'] t_2 \in T_1$ . This follows from (2) by the IH.

•  $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$ . This case is similar to the one for application.

*Sample answer:*

•  $t = y$ . There are two cases to consider:

–  $y \langle \rangle x$ . In this case, we have  $[x := t'] y = y$  by the definition of substitution. Thus, we need to show that  $\Gamma \vdash y : T$ . We know that  $\Gamma, x:T' \vdash y : T$ . Hence, by inversion we get  $(\Gamma, x:T') y = \text{Some } T$ , but we also know that  $(\Gamma, x:T') y = \Gamma y$ . So, we can conclude using the `T_Var` rule.

–  $y = x$ . In this case, we have  $[x := t'] x = t'$  by the definition of substitution. Thus, we need to show that  $\Gamma \vdash t' \in T$ . Since `empty`  $\vdash t' \in T'$ , we know that  $t'$  has no free variables. Therefore, the hypothesis of the `context_invariance` lemma is trivially satisfied, and we can apply it to obtain our result.

•  $t = \lambda y:T_1. t_1$ . There are two cases to consider:

–  $y \langle \rangle x$ . In this case, we have  $[x := t'] \lambda y:T_1. t_1 = \lambda y:T_1. [x := t'] t_1$  by the definition of substitution. Thus, we need to show that  $\Gamma \vdash \lambda y:T_1. [x := t'] t_1 \in T$ . We know that  $\Gamma, x:T' \vdash \lambda y:T_1. t_1 \in T$ . By inversion on this we also know that  $T = T_1 \rightarrow T_2$  and  $\Gamma, x:T', y:T_1 \vdash t_1 \in T_2$ . Then, the fact that  $x \langle \rangle y$  implies that for all variables  $z$ ,

$$\begin{aligned} & (\Gamma, x:T', y:T_1) z \\ &= (\text{extend } (\text{extend } \Gamma \ x \ T') \ y \ T_1) z \\ &= (\text{extend } (\text{extend } \Gamma \ y \ T_1) \ x \ T') z \\ &= (\Gamma, y:T_1, x:T') z \end{aligned}$$

which in turn implies

$$\begin{aligned} & \Gamma, x:T', y:T_1 \\ &= \Gamma, y:T_1, x:T' \end{aligned}$$

by functional extensionality. Therefore,  $\Gamma, y:T_1, x:T' \vdash t_1 \in T_2$ , and our induction hypothesis allows us to conclude  $\Gamma, y:T_1 \vdash [x := t'] t_1 \in T_2$ . By applying the `T_Abs` rule we can conclude.

–  $y = x$ . In this case, we have  $[x := t'] \lambda x:T_1. t_1 = \lambda x:T_1. t_1$  by the definition of substitution. Thus, we need to show that  $\Gamma \vdash \lambda x:T_1. t_1 \in T$ . We know

$$\Gamma, x:T' \vdash \lambda x:T_1. t_1 \in T$$

and by inversion on this derivation we have

$$\Gamma, x:T', x:T_1 \vdash t_1 \in T_2$$

which implies  $\Gamma, x:T_1 \vdash t_1 \in T_2$ . Thus, by applying the rule `T_Abs` we can conclude.

*Grading scheme: 18-20 points for perfect or nearly perfect proofs. 13-17 points for mostly correct proofs with more significant deficiencies. 1-12 points for proofs with major issues.*

10. (5 points) The STLC with records and subtyping is summarized in the appendix for reference (page ??). Indicate whether each of the following claims is true or false for this language.



(a) If  $T <: S$ , then  $(T \rightarrow U) <: (S \rightarrow U)$ .

T  F

(b) The subtype relation contains an infinite descending chain — that is, there is an infinite sequence of types  $T_1, T_2, T_3, \dots$  such that, for each  $i$ , we have  $T_{i+1} <: T_i$  but not  $T_i <: T_{i+1}$ .

T F

(c) There is a record type  $T$  that is a supertype of every other record type (that is,  $S <: T$  for every record type  $S$ ).

T F

(d) There is a type  $T$  that is a subtype of every other type (that is,  $T <: S$  for every type  $S$ ).

T  F

(e) There is a record type  $T$  that is a subtype of every other record type (that is,  $T <: S$  for every record type  $S$ ).

T  F

*Grading scheme: One point for each correct item.*

## For Reference...

### Some functions on lists

```
Fixpoint map {X Y:Type} (f:X->Y) (l:list X) : list Y :=
  match l with
  | []      => []
  | h :: t => (f h) :: (map f t)
  end.
```

```
Fixpoint fold {X Y:Type} (f:X->Y->Y) (y:Y) (l:list X) : Y :=
  match l with
  | []      => y
  | h :: t => f h (fold f y t)
  end.
```

## Definitions of logical connectives in Coq

```
Inductive and (P Q : Prop) : Prop :=  
  conj : P -> Q -> (and P Q).
```

```
Inductive or (P Q : Prop) : Prop :=  
  | or_introl : P -> or P Q  
  | or_intror : Q -> or P Q.
```

```
Notation "P /\ Q" := (and P Q) : type_scope.
```

```
Notation "P \/ Q" := (or P Q) : type_scope.
```

## 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 "X '::=' a" :=
  (CAss X 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)  
 (at level 90, c at next level)  
 : hoare\_spec\_scope.

## 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).

## Hoare logic rules

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

$$\frac{}{\{ P \} \text{ SKIP } \{ P \}} \text{ (hoare\_skip)}$$

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

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

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

$$\frac{\{ P' \} c \ \{ Q' \} \quad P \rightarrow P' \quad Q' \rightarrow Q}{\{ P \} c \ \{ Q \}} \text{ (hoare\_consequence)}$$

## Decorated programs

A decorated program consists of the program text interleaved with assertions. To check that a decorated program represents a valid proof, we check that each individual command is *locally* consistent with its accompanying assertions in the following sense:

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

```

{{ P }}
SKIP
{{ P }}

```

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

```

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

```

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

```

{{ P where a is substituted for X }}
X ::= a
{{ P }}

```

- A conditional is locally consistent (with respect to assertions  $P$  and  $Q$ ) if the assertions at the top of its “then” and “else” branches are exactly  $P \wedge \mathbf{b}$  and  $P \wedge \sim\mathbf{b}$  and if its “then” branch is locally consistent (with respect to  $P \wedge \mathbf{b}$  and  $Q$ ) and its “else” branch is locally consistent (with respect to  $P \wedge \sim\mathbf{b}$  and  $Q$ ):

```

{{ P }}
IFB b THEN
  {{ P ∧ b }}
  c1
  {{ Q }}
ELSE
  {{ P ∧ ~b }}
  c2
  {{ Q }}
FI
{{ Q }}

```

- A while loop is locally consistent if its postcondition is  $P \wedge \sim\mathbf{b}$  (where  $P$  is its precondition) and if the pre- and postconditions of its body are exactly  $P \wedge \mathbf{b}$  and  $P$ :

```

{{ P }}
WHILE b DO
  {{ P ∧ b }}
  c1
  {{ P }}
END
{{ P ∧ ~b }}

```

- A pair of assertions separated by  $\Rightarrow$  is locally consistent if the first implies the second (in all states):

$\{\{ P \}\} \Rightarrow$   
 $\{\{ Q \}\}$



## STLC with booleans

### Syntax

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

### Small-step operational semantics

$\frac{\text{value } v2}{\text{-----}}$	$(\lambda x:T.t12)\ v2 \Rightarrow [x:=v2]t12$	$(\text{ST\_AppAbs})$
$\frac{t1 \Rightarrow t1'}{\text{-----}}$	$t1\ t2 \Rightarrow t1'\ t2$	$(\text{ST\_App1})$
$\frac{\text{value } v1 \quad t2 \Rightarrow t2'}{\text{-----}}$	$v1\ t2 \Rightarrow v1\ t2'$	$(\text{ST\_App2})$
$\text{-----}$	$(\text{if true then } t1 \text{ else } t2) \Rightarrow t1$	$(\text{ST\_IfTrue})$
$\text{-----}$	$(\text{if false then } t1 \text{ else } t2) \Rightarrow t2$	$(\text{ST\_IfFalse})$
$\frac{t1 \Rightarrow t1'}{\text{-----}}$	$(\text{if } t1 \text{ then } t2 \text{ else } t3) \Rightarrow (\text{if } t1' \text{ then } t2 \text{ else } t3)$	$(\text{ST\_If})$

## Typing

$$\frac{\Gamma \ x = T}{\Gamma \vdash x \in T} \quad (\text{T\_Var})$$
$$\frac{\Gamma, \ x:T11 \vdash t12 \in T12}{\Gamma \vdash \lambda x:T11. t12 \in T11 \rightarrow T12} \quad (\text{T\_Abs})$$
$$\frac{\Gamma \vdash t1 \in T11 \rightarrow T12 \quad \Gamma \vdash t2 \in T11}{\Gamma \vdash t1 \ t2 \in T12} \quad (\text{T\_App})$$
$$\frac{}{\Gamma \vdash \text{true} \in \text{Bool}} \quad (\text{T\_True})$$
$$\frac{}{\Gamma \vdash \text{false} \in \text{Bool}} \quad (\text{T\_False})$$
$$\frac{\Gamma \vdash t1 \in \text{Bool} \quad \Gamma \vdash t2 \in T \quad \Gamma \vdash t3 \in T}{\Gamma \vdash \text{if } t1 \text{ then } t2 \text{ else } t3 \in T} \quad (\text{T\_If})$$

## Properties of STLC

Theorem preservation :

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

Theorem progress :

forall  $t \ T$ ,  
empty  $\vdash t \in T \rightarrow$   
value  $t \ \backslash/ \ \text{exists } t', t \ ==> t'$ .

## STLC with booleans, records and subtyping

### Syntax

$$\begin{array}{lll}
 T ::= \dots & t ::= \dots & v ::= \dots \\
 | \{i1:T1, \dots, in:Tn\} & | \{i1=t1, \dots, in=tn\} & | \{i1=v1, \dots, in=vn\} \\
 | \text{Top} & | t.i &
 \end{array}$$

### Small-step operational semantics

$$\begin{array}{l}
 \dots \\
 \frac{t_i \Rightarrow t_i'}{\{i1=v1, \dots, im=vm, in=t_i, \dots\} \Rightarrow \{i1=v1, \dots, im=vm, in=t_i', \dots\}} \quad (\text{ST\_Rcd}) \\
 \frac{t_1 \Rightarrow t_1'}{t_1.i \Rightarrow t_1'.i} \quad (\text{ST\_Proj1}) \\
 \frac{}{\{\dots, i=vi, \dots\}.i \Rightarrow vi} \quad (\text{ST\_ProjRcd})
 \end{array}$$

### Typing

$$\begin{array}{l}
 \dots \\
 \frac{\Gamma \vdash t_1 \in T_1 \quad \dots \quad \Gamma \vdash t_n \in T_n}{\Gamma \vdash \{i1=t_1, \dots, in=t_n\} \in \{i1:T_1, \dots, in:T_n\}} \quad (\text{T\_Rcd}) \\
 \frac{\Gamma \vdash t \in \{\dots, i:T, \dots\}}{\Gamma \vdash t.i \in T} \quad (\text{T\_proj}) \\
 \frac{\Gamma \vdash t \in S \quad S <: T}{\Gamma \vdash t \in T} \quad (\text{T\_Sub})
 \end{array}$$

## Subtyping

$\frac{S <: U \quad U <: T}{S <: T}$	(S_Trans)
$\frac{}{T <: T}$	(S_Ref1)
$\frac{}{S <: \text{Top}}$	(S_Top)
$\frac{T1 <: S1 \quad S2 <: T2}{S1 \rightarrow S2 <: T1 \rightarrow T2}$	(S_Arrow)
$\frac{n > m}{\{i1:T1\dots in:Tn\} <: \{i1:T1\dots im:Tm\}}$	(S_RcdWidth)
$\frac{S1 <: T1 \quad \dots \quad Sn <: Tn}{\{i1:S1\dots in:Sn\} <: \{i1:T1\dots in:Tn\}}$	(S_RcdDepth)
$\frac{\{i1:S1\dots in:Sn\} \text{ is a permutation of } \{i1:T1\dots in:Tn\}}{\{i1:S1\dots in:Sn\} <: \{i1:T1\dots in:Tn\}}$	(S_RcdPerm)