CIS 500: Software Foundations — Final Exam

December 20, 2016

Solutions

1 (12 points) (Functional programming in Coq) For each type below, give either a term which inhabits that type, or write "uninhabited."

1.1 `forall (X Y : Type), Y -> X`

Answer: uninhabited (otherwise Coq would be inconsistent!)

1.2 `forall (A B : Type), option (A -> B) -> option A -> B`

Answer: uninhabited, because we might not be provided a function or an initial A

1.3 `forall (A B : Type), option A -> option B`

Answer: `fun A B : Type => fun x : option A => @None B`

1.4 `forall (A B C : Type), (A * B -> C) -> A -> B -> C`

Answer: `fun A B C : Type => fun f : (A*B -> C) => fun a => fun b => f (a, b)`

1.5 `forall (A B C : Type), ((A -> B) -> C) -> (A * B -> C)`

Answer: `fun A B C : Type => fun f : ((A->B)->C) => fun ab : A*B => match ab with (a,b) => f (fun x => b) end`

2 (11 points) (Functional programming in Coq) We've seen the `fold` function over lists:

```
Fixpoint fold_list {A B : Type} (f : A -> B -> B) (b : B) (l : list A) : B :=
  match l with
  | [] => b
  | h :: t => f h (fold_list f b t)
  end.
```

The same idea can be instantiated for a wide variety of inductive data types. For instance, consider the type of binary trees (annotated with data items at their internal nodes).

```
Inductive bin (X : Type) :=
  | Node : bin X -> X -> bin X -> bin X
  | Leaf : bin X
```

2.1 (6 points) A "fold function" for this type would have the following type:

```
forall A B, (B -> A -> B -> B) -> B -> bin A -> B
```

Implement `fold_bin`. Here is the header:

```
Fixpoint fold_bin {A B : Type} (f : B -> A -> B -> B) (b : B) (t : bin A) : B :=

  match t with
  | Leaf => b
  | Node l x r => f (fold_bin f b l) x (fold_bin f b r)
```

Here is type of `fold_bin` again:

```
forall (A B : Type), (B -> A -> B -> B) -> B -> bin A -> B
```

Now consider a hypothetical inductive type of *truffula trees*. A truffula tree has two kinds of internal nodes: `Q` nodes with one data element and two children, and `P` nodes with two data elements and three children. Its leaves are also of two varieties: they can each be a `R` with four data elements or a plain `S` leaf with no payload.

```
Inductive truffula (X : Type) :=
  | P : X -> X -> truffula X -> truffula X -> truffula X -> truffula X
  | Q : X -> truffula X -> truffula X -> truffula X
  | R : X -> X -> X -> X -> truffula X
  | S : truffula X
```

2.2 (5 points) What should be the type of a fold function over truffula trees?

```
fold_truffula :

  forall (A B : Type),
    (A -> A -> B -> B -> B -> B) ->
    (A -> B -> B -> B) ->
    (A -> A -> A -> A -> B) ->
    B -> truffula A -> B
```

3 (15 points) (Program equivalence in Imp) For each part, circle *True* or *False*. Some parts also ask for examples, counterexamples, or explanations.

3.1 (1 point) The following programs are equivalent.

```
Y ::= X * 2
```

and

```
Y ::= 0;;
WHILE X > 0 DO
  Y ::= Y + 2;;
  X ::= X - 1;;
DONE
```

Answer: False (missing `X::= 0`*).*

3.2 (1 point) The following programs are equivalent.

```
X ::= 0
```

and

```
IF X = 0 THEN
  SKIP
ELSE
  WHILE True DO SKIP DONE
FI
```

Answer: False

3.3 (3 points) The following programs are equivalent for *all* choices of subcommand `c`. If you choose *False*, give a counterexample (a `c` that shows that they are not equivalent).

`WHILE True DO c DONE` and `c;; WHILE True DO c DONE`

Answer: True

3.4 (3 points) The following programs are equivalent for *some* choice of subcommand `c`. If you choose *True*, give an example of such a `c`.

`c ;; c` and `c`

Answer: True. For instance for `c = SKIP`*.*

3.5 (3 points) The following programs are equivalent for *some* choice of subcommand `c`. If you choose *True*, give an example of such a `c`.

`WHILE True DO c DONE` and `c`

Answer: True. For instance for `c = WHILE True DO SKIP DONE`*.*

3.6 (4 points) For all boolean expressions `b` and for all subcommands `c`, there exists some subcommand `d` such that the following programs are equivalent. Briefly explain your answer.

```
WHILE b DO                          IF b THEN
   c;;              and                WHILE True DO SKIP DONE
   d                                  ELSE
DONE                                    SKIP
                                    FI
```

Answer: True. If not b, then trivial. If b, then in the loop body we know that b has at least one satisfying state; d needs to re-establish a satisfying state for b.

4 **[Standard Only]** (4 points) (Hoare logic) Circle *True* or *False*.

4.1 The Hoare rule $\{\!\{ True \}\!\}$ `X ::= 2` $\{\!\{ X = 2 \}\!\}$ can be proven directly using just the `hoare_asgn` rule.

Answer: False

4.2 Everything that can be proven using the following rule can also be proven using the `hoare_if` rule.

$$\frac{\begin{array}{c}\{\!\{ P \}\!\}\ \texttt{c1}\ \{\!\{ Q \}\!\} \\ \{\!\{ P \}\!\}\ \texttt{c2}\ \{\!\{ Q \}\!\}\end{array}}{\{\!\{ P \}\!\}\ \texttt{IFB b THEN c1 ELSE c2 FI}\ \{\!\{ Q \}\!\}}\ ()$$

Answer: True

5 **[Standard Only]** (14 points) (Hoare logic)

5.1

```
WHILE X > 0 DO
  X ::= X - 1;;
  IF X = Y THEN X ::= X + 1
END
```

Check the box next to each assertion that is an *invariant* of this loop.

- ☒ `True`
- ☐ `X > 0`
- ☒ `X = Y + 1`
- ☐ `X > Y + 1`
- ☒ `X < Y + 1`
- ☒ `False`

5.2

```
R ::= 0;;
X ::= 0;;
WHILE X < Y DO
  R ::= R + Y;;
  X ::= X + 1
END
```

Check all assertions that are valid *postconditions* of this program.

☒ `True`

☐ `R = X + Y`

☒ `R = X * Y`

☐ `R = 2 * Y`

☒ `R = Y * Y`

☐ `False`

5.3

```
R ::= 0;;
X ::= 0;;
WHILE X < Y DO
  R ::= R + Z;;
  X ::= X + 1
END
```

Check assertions that are valid *postconditions* of this program.

☐ `R = Y + Z`

☒ `R = Y * Z`

Check assertions that are *invariants* of this loop and are *sufficient* to prove the postcondition(s) you checked.

☐ `True`

☐ `R = Y * (Z-X)`

☐ `X <= Y /\ R = Y * (Z-X)`

☐ `R = X * Z`

☒ `X <= Y /\ R = X * Z`

☐ `False`

6 **[Advanced Only]** (16 points) (Hoare logic) Recall the Hoare logic rule for WHILE loops:

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

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

Proof: Suppose `st` is a state satisfying `P` and that `(WHILE b DO c END) / st` $\Downarrow$ `st'`. Proceed by induction on a derivation of `(WHILE b DO c END) / st` $\Downarrow$ `st'`. Because of the form of the program, there are just two cases to consider:

1. Suppose `(WHILE b DO c END) / st` $\Downarrow$ `st'` by rule `E_WhileEnd`, with `st' = st` and with `beval st b = false`. We know `P st'` by assumption, and the assertion `(~b) st` follows by definition from the fact that `beval st b = false`, so `st'` satisfies the required postcondition.

2. Suppose `(WHILE b DO c END) / st ⇓ st'` by rule `E_WhileLoop`, with `beval st b = true` and `c /st ⇓ st1` and `(WHILE b DO c END) / st1 ⇓ st'`. By the first premise (using the fact that `beval st b = true` implies the assertion `b st`, plus the assumption that `P` holds for `st` and the definition of validity for Hoare triples), we have `P st1`. Now, by the induction hypothesis, the assertion `P /\ ~b` holds for the state `st'`, as required.

7 (14 points) (Operational semantics) In this problem, we consider an alternate formulation of the small-step operational semantics for the simply-typed lambda calculus with booleans (without subtyping and no products until part (d)).

One annoying thing about the operational semantics is the number of "structural" rules (`ST_App1`, `ST_App2`, `ST_If`) that we have to deal with. An alternate formulation of the operational semantics is to give a syntax of "evaluation contexts" `E` (of type `ectx`) and "primitive steps" `s` (which are just particular terms—those that are "ready to take a top-level step") like this:

```
(* Evaluation contexts  E : ectx *)          (* prim_step : tm -> Prop *)
E ::= []                    (* hole *)       s ::= (\x:T.t) v
    | E t                   (* ST_App1 *)        | if true then t1 else t2
    | v E                   (* ST_App2 *)        | if false then t1 else t2
    | if E then t1 else t2 (* ST_If *)
```

Here we use informal syntax rather than formal Coq syntax, to make things easier to read. We use the usual convention the `v` stands for a term that is a value. The idea is that `E` describes a term with a single "hole" `[]` in it, into which we can place an arbitrary term. We define the function that fills the hole by pattern matching on the `E` like this:

```
Fixpoint fill (t:tm) (E:ectx) : tm :=
  match E with
   | []   => t
   | E t1 => (fill t E) t1
   | v E  => v (fill t E)
   | if E then t1 else t2 => if (fill t E) then t1 else t2
  end.
```

(Again, this is "Coq pseudocode": the syntax of the patterns is informal.)

Each non-hole `E` corresponds to one of the structural rules, which lets us use one evaluation rule `ST_Hole` for all of them. We also include one rule for each primitive step, like this:

```
      s ==> t                      (\x:T.t) v ==> [x:=v]t              (ST_AppAbs)
---------------------- (ST_Hole)
fill s E ==> fill t E              if true then t1 else t2 ==> t1      (ST_IfTrue)

                                   if false then t1 else t2 ==> t2     (ST_IfFalse)
```

These rules replace the old definition of the small-step semantics.

7.1 (3 pts.) If we use these evaluation contexts to prove soundness, we need a couple of different helper lemmas.

The first lemma says that we can always decompose a well-typed term if it is not a value:

```
Lemma decompose : forall (t:tm) (T:ty) ,
  ⊢ t : T  ->
  value t \/
  exists (E:ectx), exists (s:tm),
     prim_step s /\
     t = fill s E.
```

Which of the following proofs would *directly* require this `decompose` lemma? (If A needs B and B needs the lemma, mark only B.)

- ☐ canonical forms
- ☐ preservation
- ☒ progress
- ☐ context invariance
- ☐ substitution

7.2 (4 pts.) We also need a kind of substitution lemma that relates to `fill`. A bad attempt at stating it might be something like this:

```
Lemma ectx_substitution:  forall (E:ectx) (x:id) (T U:ty) (t:tm) Γ,
  Γ, x:T ⊢ (fill x E) ∈ U ->
  ⊢ t ∈ T ->
  Γ ⊢ (fill t E) ∈ U.
```

Unfortunately, the lemma above is not provable (indeed it is false!). Briefly explain why.

Answer: The context `E` itself might contain `x` as a free variable and so it won't be well-typed when we remove `x` from the context.

7.3 (3 pts.) A better way to state the substitution principal is:

```
Lemma ectx_substitution:  forall (E:ectx) (T U:ty) (s t:tm) Γ,
  Γ ⊢ (fill s E) ∈ U ->   (* Hyp1 *)
  ⊢ s ∈ T ->             (* Hyp2 *)
  ⊢ t ∈ T ->             (* Hyp3 *)
  Γ ⊢ (fill t E) ∈ U.
```

It would be easiest to prove this fact by induction on which of the following? (Choose one.)

☒ E	☐ T	☐ U	☐ s
☐ t	☐ Hyp1	☐ Hyp2	☐ Hyp3

7.4 (4 pts.) What would we need to add to the definition of `E` to support products? (The usual rules for products are given in the handout. You may need to add more than one clause.)

```
E ::= ...
    |

    | (E, t)
    | (v, E)
    | E.fst
    | E.snd
```

8 (15 points) (Simply typed lambda-calculus) 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 constants `0` and successor `S`. You can find the syntax, typing rules, and small-step evaluation rules for this part of the language in the handout. Note that, for this problem, we do not consider subtyping, `fix`, or any other extensions to the STLC.

Rather than adding `if0` and the general recursion operator `fix` for writing programs over natural numbers, this extension follows Coq and adds 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 `!!` and where we write `2` as a shorthand for `S (S 0)`, etc.:

```
(\n:Nat. \m:Nat. nat_ind m (\x:Nat.\y:Nat. S y) n) 2 1
  ==>
(\m:Nat. nat_ind m (\x:Nat.\y:Nat. S y) 2) 1
  ==>
nat_ind 1 (\x:Nat.\y:Nat. S y) 2
  ==> !! (nonzero case)
(\x:Nat.\y:Nat. S y) 1 (nat_ind 1 (\x:Nat.\y:Nat. S y) 1)
  ==> !! (nonzero case)
(\x:Nat.\y:Nat. S y) 1 ((\x:Nat.\y:Nat. S y) 0 (nat_ind 1 (\x:Nat.\y:Nat. S y) 0))
  ==> !! (zero case)
(\x:Nat.\y:Nat. S y) 1 ((\x:Nat.\y:Nat. S y) 0 1)
  ==>
(\x:Nat.\y:Nat. S y) 1 ((\y:Nat. S y) 1)
  ==>
(\x:Nat.\y:Nat. S y) 1 2
  ==>
(\y:Nat. S y) 2
  ==>
3
```

Intuitively, the small-step operational semantics of `nat_ind` should work like this:

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

8.1 (5 points) First, let's complete the small-step operational semantics for `nat_ind`. There are three congruence rules that evaluate the arguments to `nat_ind` in order from left-to-right. The first is:

$$\frac{\texttt{tz ==> tz'}}{\texttt{nat\_ind tz ts tn ==> nat\_ind tz' ts tn}}$$

Write the other two structural rules below. Use the `value` predicate as appropriate.

$$\frac{\texttt{value vz ts ==> ts'}}{\texttt{nat\_ind vz ts tn ==> nat\_ind vz ts' tn}}$$

$$\frac{\texttt{value vz tn ==> tn'} \qquad \texttt{value vs}}{\texttt{nat\_ind vz vs tn ==> nat\_ind vz vs tn'}}$$

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 ==> vz

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

8.2 (4 points) 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 above should be well-typed according to your rule.

```
Gamma |- tz : T     Gamma |- ts : Nat -> T -> T    Gamma |- tn : Nat
-------------------------------------------------------------------
                  Gamma |- nat_ind tz ts tn : T
```

8.3 (4 points) Above, we used `nat_ind` to define the `Nat_plus` function. Now, use `Nat_plus` and `nat_ind` to define *multiplication* of two numbers.

```
Nat_mult  :  Nat -> Nat -> Nat

Nat_mult  =

   \n:Nat. \m:Nat. nat_ind 0 (\x:Nat. \p:Nat. Nat_plus m p) n
```

9 (25 points) (Simply typed lambda-calculus with subtyping) The syntax, operational semantics, and typing rules for the simply-typed lambda calculus with subtyping and booleans are given in the handout.

For each variant of this system described below, indicate which of the properties remain true in the presence of this rule. For each one, circle either "*Remains true*" or else "*Becomes false.*" If a property becomes false, give a counterexample.

9.1 Consider a variant in which we add the following new subtyping rule:

```
-----------------------    S_ProdArrow)
Top * Top <: Top -> Top
```

- Determinism of `==>`
  Answer: Remains true.

- Progress

  Answer: Becomes false. For example, `(true,true) true` *is typeable, but stuck.*
- Preservation

  Answer: Remains true.

9.2 Consider a variant in which we add a new term `loop` with the following reduction and typing rules:

```
------------- (ST_Loop)        ---------------- (T_Loop)
loop ==> loop                   Γ ⊢ loop ∈ T
```

- Determinism of `==>`

  Answer: Remains true. Each term has at most one rule that applies.
- Progress

  Answer: Remains true. Every well-typed term can still take a step, as can `loop`*.*
- Preservation

  Answer: Remains true. `loop` *can have any type.*

9.3 Suppose instead that we add the following typing rule:

```
Γ ⊢ t1 ∈ U -> T
----------------      (T_App')
 Γ ⊢ t1 t2 ∈ T
```

- Determinism of `==>`

  Answer: Remains true.
- Progress

  Answer: Remains true.
- Preservation

  Answer: Becomes false: `(\x:(Bool->Bool). x true) true` *can be given the type* `Bool` *but it steps to* `true true` *which is ill-typed. (The substitution lemma does not apply.)*

9.4 Instead, suppose that we add a new term `guess T` (where `T` is a type) with the following reduction rule:

```
--------------------------------      (ST_GArr)
guess (T -> U) ==> \x:T. guess U
```

and the following typing rule:

```
----------------      (T_Guess)
Γ ⊢ guess T ∈ T
```

- Determinism of `==>`
  Answer: Remains true.
- Progress
  Answer: Becomes false. The term `guess Bool` *is stuck.*
- Preservation
  Answer: Remains true.

9.5 Instead, suppose that we add a new reduction rule:

```
        t ==> t'
--------------------      (ST_LCong)
\x:T. t ==> \x:T. t'
```

- Determinism of `==>`
  Answer: Becomes false. The term `(\x:T. t) v` *reduces in two ways, provided* `t` *can reduce*
- Progress
  Answer: Remains true.
- Preservation
  Answer: Remains true.

10 **[Standard Only]** (10 points) (Subtyping) For each of the following pairs of types, `S` and `T`, circle the appropriate description of how they are ordered by the subtype relation.

10.1

```
S = Top -> Top
T = (Top -> Top) -> Top
```

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

10.2

```
S = {x: Bool, y: Bool -> Top}
T = {x: Top, y: Top -> Bool}
```

- S<:T
- T<:S
- equivalent
- [unrelated]

10.3

```
S = {x: Bool -> Top}
T = {y: Nat, x: Top -> Top}
```

- S<:T
- **T<:S**
- equivalent
- unrelated

10.4

```
S = (Bool -> Top) -> (Nat -> Nat)
T = (Top -> Bool) -> Top
```

- **S<:T**
- T<:S
- equivalent
- unrelated

10.5

```
S = {x: Nat, y: Bool} -> Nat
T = {y: Bool, x: Nat} -> Nat
```

- S<:T
- T<:S
- **equivalent**
- unrelated

11 **[Advanced Only]** (12 points) (Subtyping) Give a (careful and detailed) *informal* proof of the following lemma about the simply typed lambda-calculus with subtyping (a slight variation on one we saw in the Subtyping chapter of *Software Foundations*):

> **Lemma**: If $\Gamma \vdash \mathtt{s} \in \mathtt{T}_1 \to \mathtt{T}_2$ and $\mathtt{s}$ is a value, then there exist $\mathtt{x}$, $\mathtt{S}_1$, and $\mathtt{s}_2$ such that:
>
> - $\mathtt{s} = \backslash \mathtt{x}{:}\mathtt{S}_1.\, \mathtt{s}_2$
> - $\Gamma,\ \mathtt{x}{:}\mathtt{S}_1 \vdash \mathtt{s}_2 \in \mathtt{S}_2$
> - $\mathtt{T}_1 <: \mathtt{S}_1$
> - $\mathtt{S}_2 <: \mathtt{T}_2$

Proof. By induction on the given derivation of $\Gamma \vdash \mathtt{s} \in \mathtt{T}_1 \to \mathtt{T}_2$. Since $\mathtt{s}$ is a value of arrow type, the last rule in this derivation must be either `T_Abs` or `T_Sub`.

- Suppose the last rule is `T_Abs`, with $\mathtt{s} = \backslash \mathtt{x}{:}\mathtt{T}_1.\, \mathtt{s}_2$ and $\Gamma,\ \mathtt{x}{:}\mathtt{T}_1 \vdash \mathtt{s}_2 \in \mathtt{T}_2$. All four of the required conditions are satisfied, with $\mathtt{S}_1 = \mathtt{T}_1$ and $\mathtt{S}_2 = \mathtt{T}_2$ (and noting that $\mathtt{T}_1 <: \mathtt{T}_1$ and $\mathtt{T}_2 <: \mathtt{T}_2$ by reflexivity of subtyping).
- Suppose the last rule is `T_Sub`, with $\Gamma \vdash \mathtt{s} \in U$ for some $\mathtt{U}$ with $\mathtt{U} <: \mathtt{T}_1 \to \mathtt{T}_2$. By lemma `sub_inversion_arrow` from `Sub.v`, $\mathtt{U} = \mathtt{U}_1 \to \mathtt{U}_2$, where $\mathtt{T}_1 <: \mathtt{U}_1$ and $\mathtt{U}_2 <: \mathtt{T}_2$. By the induction hypothesis, $\mathtt{s} = \backslash \mathtt{x}{:}\mathtt{S}_1.\, \mathtt{s}_2$ for some $\mathtt{x}$, $\mathtt{S}_1$, and $\mathtt{s}_2$ with $\Gamma,\ \mathtt{x}{:}\mathtt{S}_1 \vdash \mathtt{s}_2 \in \mathtt{S}_2$, $\mathtt{U}_1 <: \mathtt{S}_1$, and $\mathtt{S}_2 <: \mathtt{U}_2$. By transitivity, $\mathtt{T}_1 <: \mathtt{S}_1$ and $\mathtt{S}_2 <: \mathtt{T}_2$, as required.

For Reference

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

1. `SKIP` is locally consistent if its precondition and postcondition are the same:

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

2. 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 }}
```

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

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

4. 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 }}
```

5. 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 }}
```

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

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

Small Step Semantics

```
Reserved Notation " t '/' st '==>a' t' " (at level 40, st at level 39).

Inductive astep : state -> aexp -> aexp -> Prop :=
  | AS_Id : forall st i,
      AId i / st ==>a ANum (st i)
  | AS_Plus : forall st n1 n2,
      APlus (ANum n1) (ANum n2) / st ==>a ANum (n1 + n2)
  | AS_Plus1 : forall st a1 a1' a2,
      a1 / st ==>a a1' ->
      (APlus a1 a2) / st ==>a (APlus a1' a2)
  | AS_Plus2 : forall st v1 a2 a2',
      aval v1 ->
      a2 / st ==>a a2' ->
      (APlus v1 a2) / st ==>a (APlus v1 a2')
  | AS_Minus : forall st n1 n2,
      (AMinus (ANum n1) (ANum n2)) / st ==>a (ANum (minus n1 n2))
  | AS_Minus1 : forall st a1 a1' a2,
      a1 / st ==>a a1' ->
      (AMinus a1 a2) / st ==>a (AMinus a1' a2)
  | AS_Minus2 : forall st v1 a2 a2',
      aval v1 ->
      a2 / st ==>a a2' ->
      (AMinus v1 a2) / st ==>a (AMinus v1 a2')
  | AS_Mult : forall st n1 n2,
      (AMult (ANum n1) (ANum n2)) / st ==>a (ANum (mult n1 n2))
  | AS_Mult1 : forall st a1 a1' a2,
      a1 / st ==>a a1' ->
      (AMult a1 a2) / st ==>a (AMult a1' a2)
```

```
  | AS_Mult2 : forall st v1 a2 a2',
      aval v1 ->
      a2 / st ==>a a2' ->
      (AMult v1 a2) / st ==>a (AMult v1 a2')
    where " t '/' st '==>a' t' " := (astep st t t').

Reserved Notation " t '/' st '==>' t' '/' st' "
                  (at level 40, st at level 39, t' at level 39).

Inductive cstep : (com * state) -> (com * state) -> Prop :=
  | CS_AssStep : forall st i a a',
      a / st ==>a a' ->
      (i ::= a) / st ==> (i ::= a') / st
  | CS_Ass : forall st i n,
      (i ::= (ANum n)) / st ==> SKIP / (t_update st i n)
  | CS_SeqStep : forall st c1 c1' st' c2,
      c1 / st ==> c1' / st' ->
      (c1 ;; c2) / st ==> (c1' ;; c2) / st'
  | CS_SeqFinish : forall st c2,
      (SKIP ;; c2) / st ==> c2 / st
  | CS_IfTrue : forall st c1 c2,
      IFB BTrue THEN c1 ELSE c2 FI / st ==> c1 / st
  | CS_IfFalse : forall st c1 c2,
      IFB BFalse THEN c1 ELSE c2 FI / st ==> c2 / st
  | CS_IfStep : forall st b b' c1 c2,
      b / st ==>b b' ->
          IFB b THEN c1 ELSE c2 FI / st
      ==> (IFB b' THEN c1 ELSE c2 FI) / st
  | CS_While : forall st b c1,
          (WHILE b DO c1 END) / st
      ==> (IFB b THEN (c1;; (WHILE b DO c1 END)) ELSE SKIP FI) / st
  where " t '/' st '==>' t' '/' st' " := (cstep (t,st) (t',st')).
```

STLC with booleans

Syntax

```
T ::= Bool              t ::= x                   v ::= true
    | T -> T                | t t                     | false
                            | \x:T. t                 | \x:T. t
                            | true
                            | false
                            | if t then t else t
```

Small-step operational semantics

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

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

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

$$\frac{}{(\text{if true then } t_1 \text{ else } t_2) \Longrightarrow t_1} \text{(ST\_IfTrue)}$$

$$\frac{}{(\text{if false then } t_1 \text{ else } t_2) \Longrightarrow t_2} \text{(ST\_IfFalse)}$$

$$\frac{t_1 \Longrightarrow t_1'}{(\text{if } t_1 \text{ then } t_2 \text{ else } t_3) \Longrightarrow (\text{if } t_1' \text{ then } t_2 \text{ else } t_3)} \text{(ST\_If)}$$

Typing

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

$$\frac{\Gamma,\ \texttt{x:T11} \vdash \texttt{t12} \in \texttt{T12}}{\Gamma \vdash \texttt{\textbackslash x:T11.t12} \in \texttt{T11->T12}}\ (\texttt{T\_Abs})$$

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

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

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

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

Properties of STLC

```
Theorem preservation : forall t t' T,
    empty ⊢ t ∈ T  ->
    t ==> t'  ->
    empty ⊢ t' ∈ T.

Theorem progress : forall t T,
    empty ⊢ t ∈ T ->
    value t \/ exists t', t ==> t'.
```

STLC with products

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

```
T ::= ...                t ::=  ...                  v ::= ...
    | T * T                 | (t,t)                      | (v, v)
                            | t.fst
                            | t.snd
```

Small-step operational semantics (added to STLC rules)

$$\frac{t1 \Longrightarrow t1'}{(t1,t2) \Longrightarrow (t1',t2)} \quad \text{(ST\_Pair1)}$$

$$\frac{t2 \Longrightarrow t2'}{(v1,t2) \Longrightarrow (v1,t2')} \quad \text{(ST\_Pair2)}$$

$$\frac{t1 \Longrightarrow t1'}{t1.fst \Longrightarrow t1'.fst} \quad \text{(ST\_Fst1)}$$

$$\frac{}{(v1,v2).fst \Longrightarrow v1} \quad \text{(ST\_FstPair)}$$

$$\frac{t1 \Longrightarrow t1'}{t1.snd \Longrightarrow t1'.snd} \quad \text{(ST\_Snd1)}$$

$$\frac{}{(v1,v2).snd \Longrightarrow v2} \quad \text{(ST\_SndPair)}$$

Typing (added to STLC rules)

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

$$\frac{\Gamma \vdash t1 \in T11*T12}{\Gamma \vdash t1.fst \in T11} \quad \text{(T\_Fst)}$$

$$\frac{\Gamma \vdash t1 \in T11*T12}{\Gamma \vdash t1.snd \in T12} \quad \text{(T\_Snd)}$$

Subtyping

Extend the language above with the type `Top` (terms and values remain unchanged):

```
T ::= ...
    | Top
```

Add these rules that characterize the subtyping relation:

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

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

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

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

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

Typing (added to STLC with products)

All of the ordinary typing rules, plus:

$$\frac{\Gamma \vdash t \in S \quad S <: T}{\Gamma \vdash t \in T} \text{(T\_Sub)}$$