CIS 500 — Software Foundations

Final Exam

(Standard and advanced versions together)

December 18, 2014

Answer key
1. Properties of Imp Relations

The propositions below concern basic properties of the Imp language. For each proposition, indicate whether it is true or false by circling either True or False. For reference, the definition of Imp, its evaluation semantics, and program equivalence (cequiv) starts on page 16.

(a) The evaluation relation for Imp is deterministic.

Answer: True

(b) The cequiv relation is symmetric.

Answer: True

(c) The command WHILE BFalse DO SKIP END is not equivalent to any other command.

Answer: False

(d) There is an Imp command c that terminates for some input states and diverges for others.

Answer: True

(e) If cequiv c1 c2 then cequiv (SKIP ;; c1) (SKIP ;; c2).

Answer: True

(f) For all arithmetic expressions a1 and a2, we can show

cequiv (X ::= a1 ;; Y ::= a2) (Y ::= a2 ;; X ::= a1)

Answer: False

(g) If SKIP / st || st' then we know that st = st'.

Answer: True
2. **Hoare Logic**

The following Imp program (slowly) computes $X + Y$, placing the answer into $Z$.

\[ \begin{align*}
Z & ::= Y; \\
\text{WHILE } X <> 0 \text{ DO} \\
Z & ::= Z + 1; \\
X & ::= X - 1
\end{align*} \]

Below, add appropriate annotations in the provided spaces. You will need to give the outermost pre- and post-conditions; these assertions should show that the program works as described above. Use informal notations for mathematical formulae and assertions, but be completely precise 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 $\rightarrow$.

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

- **forall** $a$, $b$, $c$, $b <> 0 \rightarrow (a + 1) + (b - 1) = a + b$

The Hoare rules and the rules for well-formed decorated programs are provided on pages 18 and 19.

\[
\begin{align*}
\{\{ X = m \}\} & \rightarrow \\
\{\{ Y + X = m + Y \}\} & \\
Z & ::= Y \\
\{\{ Z + X = m + Y \}\} & \\
\text{WHILE } X <> 0 \text{ DO} \\
\{\{ Z + X = m + Y \land (X <> 0) \}\} & \rightarrow \\
\{\{ (Z + 1) + (X - 1) = m + Y \}\} & \\
Z & ::= Z + 1; \\
\{\{ Z + (X - 1) = m + Y \}\} & \\
X & ::= X - 1 \\
\{\{ Z + X = m + Y \}\} & \\
\end{align*} \]

Grading scheme:

- 1 point per each assignment back propagation
- 1 point for including $X <> 0$ and $(X <> 0)$ at appropriate parts
- 1 point for circling the correct implication
- 4 points for specifying the program with an appropriate pre/post condition
3. **Coq programming - Small-step semantics**

This problem refers to the Coq version of the small-step relation (\texttt{step}) for the Simply-typed Lambda Calculus with booleans, shown on page 22.

Because the \texttt{step} relation is deterministic, we can write a Coq function, called \texttt{next_step}, that computes what each term steps to (if any). The next page shows (part of) the definition of this function; you will need to complete the definition. Your implementation should satisfy the following correctness lemmas that state that \texttt{next_step} exactly corresponds to the \texttt{step} relation. (You do not need to prove these lemmas).

\begin{align*}
\text{Lemma next_step_correct1} : \forall t t', \\
\quad \text{step} t t' \iff \text{next_step} t = \text{Some} t'.
\end{align*}

\begin{align*}
\text{Lemma next_step_correct2} : \forall t, \\
\quad \text{normal_form} \text{ step} t \iff \text{next_step} t = \text{None}.
\end{align*}

Below, the following example demonstrates the evaluation of the \texttt{next_step} function.

\begin{verbatim}
(* Identity function for booleans *)
Definition idB := tabs x TBool (tvar x).
Example ex0 : next_step (tapp idB ttrue) = Some ttrue.
\end{verbatim}

(a) Fill in the blanks for the examples below. Some of these examples use \texttt{idB} defined above. Your answers should be consistent with the correctness lemmas.

\begin{verbatim}
Example ex1 : next_step ttrue = None.
Example ex2 : next_step (tapp ttrue tfalse) = None.
Example ex3 : next_step (tapp idB (tif ttrue tfalse ttrue)) = \text{Some} (tapp idB tfalse).
Example ex4 : next_step (tif ttrue (tapp idB ttrue) tfalse) = \text{Some} (tapp idB ttrue).
\end{verbatim}

*Grading scheme: 2 points per blank*
(b) Now complete the implementation of the \texttt{next_step} function. The first few cases of this implementation have been given for you.

Your code may use the following helper function in your answer.

\begin{verbatim}
(* Determine whether the given term is a value *)
Fixpoint is_value (t : tm) : bool :=
  match t with
  | tabs x T u => true
  | ttrue     => true
  | tfalse    => true
  | _         => false
end.

(* Calculate the next (small-)step for this term, if one exists *)
Fixpoint next_step (t : tm) : option tm :=
  match t with
  | tif ttrue t2 t3 => Some t2
  | tif tfalse t2 t3 => Some t3
  | tif t1 t2 t3 => match next_step t1 with
    | Some t1' => Some (tif t1' t2 t3)
    | None     => None
  end
  | tapp (tabs x T t1) t2 =>
    if is_value t2 then Some (subst x t2 t1)
    else match next_step t2 with
      | Some t2' => Some (tapp t1 t2')
      | None     => None
    end
  | tapp t1 t2 => match next_step t1 with
    | Some t1' => Some (tapp t1' t2)
    | None     => None
  end
  | _         => None
end.
\end{verbatim}

\textit{Grading scheme:}

\begin{itemize}
  \item 3 points for sending all values to None
  \item 3 points for checking if \(t1\) steps and return Some if so and None if \(t1\) is not a value
  \item 3 points for checking if \(t1\) is a value and \(t2\) steps and returning Some if so and None if \(t2\) is not a value
  \item 3 points for beta reduction when \(t1\) and \(t2\) are both values
  \item Various errors about pattern matching in Coq at discretion
\end{itemize}
4. Inductive Definitions and Scoping

Consider the following Coq definitions for a simple language of expressions with constants, variables, and options.

Inductive tm : Type :=
| tnum : nat -> tm (* Constants 0, 1, 2, ... *)
| tvar : id -> tm (* Variables X Y Z ... *)
| tsome : tm -> tm (* Some t1 *)
| tnone : tm (* None *)
| tmatch : tm -> tm -> id -> tm -> tm (* match t1 with
| None => t2
| Some x => t3 *)

For example, we might encode the Coq expression

match x with
| None => Some 0
| Some y => None
end

as

tmatch (tvar x) (tsome (tnum 0)) y tnone

The tmatch construct follows the usual variable scoping rules. That is, in the expression tmatch t1 t2 X t3 the variable X is bound in t3.

Note that a variable X appears free in a term t if there is an occurrence of X that is not bound by a corresponding tmatch. 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. You may use the next page if you need more space.

Answer:

Inductive afi : id -> tm -> Prop :=
| afi_var : forall x, afi x (tvar x)
| afi_somel : forall x t1,
  afi x t1 ->
  afi x (tsome t1)
| afi_match1 : forall x y t1 t2 t3,
  afi x t1 ->
  afi x (tmatch t1 t2 y t3)
| afi_match2 : forall x y t1 t2 t3,
  afi x t2 ->
  afi x (tmatch t1 t2 y t3)
| afi_match3 : forall x y t1 t2 t3,
  x <> y ->
afi x t3 ->
afi x (tmatch t1 t2 y t3)

Grading scheme:
• 2 pts - afi_var
• 2 pts - afi_some1
• 2 pts - afi_match1
• 2 pts - afi_match2
• 4 pts - afi_match3
• 2 pts - not having afi_none or afi_num (if other constructors are reasonable)
5. [Standard] Simply-typed Lambda Calculus

This problem again considers the simply-typed lambda calculus with booleans. This language is type safe, a fact that can be proved using the standard preservation and progress proofs, and evaluation is deterministic.

Which of these properties are broken after each of the following modifications to STLC. (These modifications are made independently from one another.) In each case, circle each either “Remains true” or “Becomes false.” For each one that becomes false, give a counterexample.

Grading scheme: 2 points per subcase

(a) Suppose that we add a new term \texttt{foo} with the following reduction rules:

\[
\begin{align*}
\text{(ST_Foo1)} & \quad (\lambda x : A. x) \Rightarrow \texttt{foo} \\
\text{(ST_Foo2)} & \quad \texttt{foo} \Rightarrow \texttt{true}
\end{align*}
\]

i. \texttt{step} is deterministic

\textit{Answer: Becomes false}

ii. Progress

\textit{Answer: Remains true}

iii. Preservation

\textit{Answer: Becomes false}

Determinism fails because \((\lambda x : \text{Bool}. x) \texttt{true} \) steps to \texttt{foo true}. Preservation fails because \((\lambda x : \text{Bool}. x) \) steps to \texttt{foo}, which does not have a type.

(b) Suppose instead that we add the following new rule to the typing relation:

\[
\begin{align*}
\Gamma & \vdash t_1 \in \text{Bool} \\
\Gamma & \vdash t_2 \in \text{Bool}
\end{align*}
\]

\[
\text{(T_FunnyApp)}
\]

\[
\Gamma \vdash t_1 \, t_2 \in \text{Bool}
\]

i. \texttt{step} is deterministic

\textit{Answer: Remains true}

ii. Progress

\textit{Answer: Becomes false}

iii. Preservation

\textit{Answer: Remains true}
Progress breaks because \texttt{true false} doesn’t step, but isn’t a value.

(c) Suppose instead that we remove the rule $T_{\text{If}}$ from the typing relation.

i. \texttt{step} is deterministic
   \textit{Answer: Remains true}

ii. Progress
   \textit{Answer: Remains true}

iii. Preservation
   \textit{Answer: Remains true}
6. Subtyping

The subtyping rules for STLC extended with pairs and records are given on page 25 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 $\text{Top*Top}$, $\text{A*Top}$, $\text{Top*(Top*Top)}$, and $\text{Top*(A*A)}$ would look like this (it happens to form a tree, but that is not necessary in general):

```
Top*Top
   / \         /
  /   \       /   \\
/     \     /     \    \
\A*Top \   \Top*(Top*Top)\
  \     \   \       \   
    \   \    \       
     \Top*(A*A)\
```

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

{ }
{ m : A }
{ m : B }
{ k : B }
{ m : Top }
{ m : A , k : B }
Top
{ m : A } -> Top

Grading scheme: -2 points per mistake (i.e. arrow out of place). Answer:
7. [Standard] Subtyping

For each question, circle whether it is true or false. Briefly justify your answer.
Grading scheme: 2 points for true/false, 2 points for justification

(a) In STLC with subtyping (see the rules on page 24) there exists a type \( T \) such that \((\forall x:T. \ x \ x)\) is typeable.

\[ \text{Answer: True} \]
\[ \text{Why: } T = \text{Top} \rightarrow \text{Top} \]

(b) In STLC with subtyping, if we know \( \Gamma \vdash \forall x:U.t \in T \), then \( T \) must be equal to \( U \rightarrow S \) where \( \Gamma, x:U \vdash t \in S \).

\[ \text{Answer: False} \]
\[ \text{Why: Subsumption} \]

(c) In STLC with subtyping, if \( A \) is not equal to \( \text{Top} \), then the type \( A \rightarrow A \) is a subtype of \( \text{Top} \rightarrow A \).

\[ \text{Answer: False} \]
\[ \text{Why: It is a supertype.} \]

(d) The largest type \( U \) that makes the assertion below true is \( (A \rightarrow A \times B \rightarrow B) \). (Recall that if \( S <: T \), then we say that \( T \) is larger than \( S \).)

\[ \text{empty} \vdash (\forall p:(A \rightarrow A \times B \rightarrow B). \ p((\forall z:A.z), (\forall z:B.z))) \in U \]

\[ \text{Answer: False} \]
\[ \text{Why: False: that is the smallest type. The largest type is Top.} \]
8. [Advanced] Informal Proofs - Subtyping

Give a detailed proof of the following inversion lemma for typing abstractions in STLC with
subtyping (see the rules on page 24). You should prove this lemma for the with booleans and
functions only, not the extension with product and record types.

**Lemma (Typing Inversion for Abstractions):** If \( \Gamma \vdash \text{x}:S_1 . t_2 \in T \), then there is a type \( S_2 \) such
that \( \Gamma, x:S_1 \vdash t_2 \in S_2 \) and \( S_1 \to S_2 <: T \).

**Proof:** Let \( \Gamma, x, S_1, t_2 \) and \( T \) be given as described. Proceed by induction on the derivation
of \( \Gamma \vdash \text{x}:S_1 . t_2 \in T \). Cases \( T_{\text{Var}} \), \( T_{\text{App}} \), are vacuous as those rules cannot be used to give a type
to a syntactic abstraction.

- If the last step of the derivation is a use of \( T_{\text{Abs}} \) then there is a type \( T_{12} \) such that
  \( T = S_1 \to T_{12} \) and \( \Gamma, x:S_1 \vdash t_2 \in T_{12} \). Picking \( T_{12} \) for \( S_2 \) gives us what we need:
  \( S_1 \to T_{12} <: S_1 \to T_{12} \) follows from \( S_{\text{Refl}} \).

- If the last step of the derivation is a use of \( T_{\text{Sub}} \) then there is a type \( S \) such that \( S <: T \)
  and \( \Gamma \vdash \text{x}:S_1 . t_2 \in S \). The IH for the typing subderivation tell us that there is some type \( S_2 \)
  with \( S_1 \to S_2 <: S \) and \( \Gamma, x:S_1 \vdash t_2 \in S_2 \). Picking type \( S_2 \) gives us what we need, since
  \( S_1 \to S_2 <: T \) then follows by \( S_{\text{Trans}} \).

**Grading scheme:**

- 2 points for proof by induction
- 2 points for having a \( T_{\text{Abs}} \) case
- 2 points for having a \( T_{\text{Sub}} \) case
- 2 points for observing that other cases are trivial
- 3 points for saying what is known about typing in \( T_{\text{Abs}} \) case
- 2 points for saying what is known about typing in \( T_{\text{Sub}} \) case
- 3 points for using IH
- 2 points for using transitivity of subtyping

Proofs attempts by inversion on the subtyping relation received very little credit.
9. [Advanced] Informal Proofs — Progress with null

In this problem we will extend STLC with null. Your job will be to state and prove a progress lemma for this extension. Although we will work with informal notation for this problem, the base language is the one specified by the Coq formalization, starting on page 21. In particular, there is no subtyping in this language.

Informally, the only change we will make to the syntax of the language is to add null,

\[
t ::= x \\
| \lambda x : T . t \\
| t_1 \ t_2 \\
| \text{true} \\
| \text{false} \\
| \text{if } t_1 \ \text{then } t_2 \ \text{else } t_3 \\
| \text{null}
\]

which is a new form of value.

\[
\text{--------------- (v_null)}
\]

\text{value null}

We also add one new typing rule, shown below, that allows the null value to have any type.

\[
\text{--------------- (T_Null)}
\]

\[\Gamma \vdash \text{null} \in T\]

(The call-by-value operational semantics for this language is unchanged.)

The standard progress lemma doesn’t hold for STLC with this extension. The problem is that a term could get stuck when trying to use a null value. For example the term if null then t1 else t2 is well typed, but isn’t a value and doesn’t step. We say that stuck terms like this one throw null pointer exceptions. In this problem, we won’t model those exceptions directly. Instead, we define a relation, called npe, that describes where they should occur. We can use this relation to modify the Progress lemma (as shown on the next page) so that it is true.

The npe relation (shown on the next page) characterizes those terms that should throw null pointer exceptions. This relation includes terms that try to use null as another sort of value. It does not include any other stuck terms.

Your task for this problem is to complete the proof of this revised progress lemma.

Hint: This question is asking whether you understand the proof of the progress lemma for STLC. You will receive half credit for recreating the appropriate parts of the standard proof here, without the extension to null.

(16 points)
The npe relation

\[
\begin{align*}
\text{NPE}_\text{If} & : \quad \text{npe} \begin{cases}
\text{null} & \text{then } t_1 \text{ else } t_2 \\
\text{npe} t_1 &
\end{cases}
\text{NPE}_\text{If1} & : \quad \text{npe} \begin{cases}
\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \\
\text{npe} t_1 &
\end{cases}
\text{NPE}_\text{App} & : \quad \text{npe} \begin{cases}
\text{null} & \text{t_1} \\
\text{npe} t_1 &
\end{cases}
\text{NPE}_\text{App1} & : \quad \text{npe} \begin{cases}
\text{t_1 t_2} \\
\text{npe} t_1 &
\end{cases}
\text{NPE}_\text{App2} & : \quad \text{npe} \begin{cases}
\text{value t_1} \quad \text{npe} t_2 \\
\text{npe} t_1 &
\end{cases}
\end{align*}
\]

Lemma (Progress): For all t and T, if \( \emptyset \vdash t \in T \) then either t is a value, throws a null pointer exception, or steps. (i.e. either value t or npe t holds or there exists some t’ such that \( t \Rightarrow t’ \)).

Proof: by induction on the derivation of \( \emptyset \vdash t \in T \).

- The last rule of the derivation cannot be T_Var, since a variable is never well typed in an empty context.

- The T_Null, T_True, T_False, and T_Abs cases are trivial, because in each of these cases we know immediately that t is a value.

- If the last rule of the derivation was T_If, then ... \( t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3 \), where \( t_1 \) has type Bool. By the IH, \( t_1 \) either is a value or takes a step or is a NPE.
  
  - Subcase: If \( t_1 \) is a value, then since it has type Bool it must be either true or false or null. If it is true, then \( t \) steps to \( t_2 \); if false, it steps to \( t_3 \), otherwise it is a NPE.
  
  - Subcase: Otherwise, if \( t_1 \) takes a step, and therefore so does \( t \) (by ST_If).
  
  - Subcase: Otherwise, if \( t_1 \) is a NPE, then the if expression is a NPE.

- If the last rule of the derivation was T_App, then ... Answer: \( t = t_1 t_2 \), and we know that \( t_1 \) and \( t_2 \) are also well typed in the empty context; in particular, there exists a type T2 such
that \(\vdash t_1 \in T_2 \rightarrow T\) and \(\vdash t_2 \in T_2\). By the induction hypothesis, either \(t_1\) is a value, or it can take an evaluation step, or \text{npe} \(t_1\).

- **Subcase:** If \(t_1\) is a value, we now consider \(t_2\), which by the other induction hypothesis must also either be a value, take an evaluation step, or be a NPE.
  * Suppose \(t_2\) is a value. Since \(t_1\) is a value with an arrow type, it must either be a lambda abstraction or null; therefore either \(t_1 \ t_2\) can take a step by \text{ST_AppAbs}, or we have a NPE \(\text{null} \ t_2\).
  * Otherwise, if \(t_2\) can take a step, then so can \(t_1 \ t_2\) by \text{ST_App2}.
  * Finally, if \(t_2\) is a NPE then we are done by \text{NPE_App2}.

- **Subcase:** If \(t_1\) can take a step, then so can \(t_1 \ t_2\) by \text{ST_App1}.

- **Subcase:** If \text{npe} \(t_1\) then we are done by \text{NPE_App1}.

*Grading scheme:*  
- 2 points for using IH for \(t_1\) to say there are 3 cases  
- 2 points for using IH for \(t_2\) to say there are 3 cases  
- 4 points for case where \(t_1\) is a value and \(t_2\) is a value (using inversion of typing to observe that \(t_1\) could be either lambda or null)  
- 2 points where \(t_1\) is a value and \(t_2\) steps  
- 2 points where \(t_1\) is a value and \text{npe} \(t_2\)  
- 2 points where \(t_1\) steps  
- 2 points where \(t_1\) is a npe
(T_{App} case of the revised progress proof).
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

\[
\text{Inductive ceval : com} \to \text{state} \to \text{state} \to \text{Prop} :=
\]
\[
| \text{E\_Skip} : \forall \text{st}, \\
\quad \text{SKIP} / \text{st} \mid\mid \text{st} \\
| \text{E\_Ass} : \forall \text{st} a1 n X, \\
\quad \text{aeval st a1 = n ->} \\
\quad (X ::= a1) / \text{st} \mid\mid (\text{update st} X n) \\
| \text{E\_Seq} : \forall \text{c1 c2 st st'} st'', \\
\quad \text{c1} / \text{st} \mid\mid \text{st'} -> \\
\quad \text{c2} / \text{st'} \mid\mid \text{st''} -> \\
\quad (\text{c1} ; \text{c2}) / \text{st} \mid\mid \text{st''} \\
| \text{E\_IfTrue} : \forall \text{st st'} b1 c1 c2, \\
\quad \text{beval st b1 = true ->} \\
\quad \text{c1} / \text{st} \mid\mid \text{st'} -> \\
\quad (\text{IFB b1 THEN c1 ELSE c2 FI}) / \text{st} \mid\mid \text{st'} \\
| \text{E\_IfFalse} : \forall \text{st st'} b1 c1 c2, \\
\quad \text{beval st b1 = false ->} \\
\quad \text{c2} / \text{st} \mid\mid \text{st'} -> \\
\quad (\text{IFB b1 THEN c1 ELSE c2 FI}) / \text{st} \mid\mid \text{st'} \\
| \text{E\_WhileEnd} : \forall b1 \text{st c1}, \\
\quad \text{beval st b1 = false ->} \\
\quad (\text{WHILE b1 DO c1 END}) / \text{st} \mid\mid \text{st} \\
| \text{E\_WhileLoop} : \forall \text{st st'} st'' b1 c1, \\
\quad \text{beval st b1 = true ->} \\
\quad \text{c1} / \text{st} \mid\mid \text{st'} -> \\
\quad (\text{WHILE b1 DO c1 END}) / \text{st'} \mid\mid \text{st''} -> \\
\quad (\text{WHILE b1 DO c1 END}) / \text{st} \mid\mid \text{st''} \\
\]

where "c1 '/' st '||' st'" := (ceval c1 st st').

Program equivalence

\[
\text{Definition bequiv (b1 b2 : bexp) : Prop :=} \\
\quad \forall \text{st:state}, \text{beval st b1 = beval st b2}. \\
\]

\[
\text{Definition cequiv (c1 c2 : com) : Prop :=} \\
\quad \forall \text{st st'} : \text{state}, \\
\quad (\text{c1} / \text{st} \mid\mid \text{st'}) <-> (\text{c2} / \text{st} \mid\mid \text{st'}). \\
\]

Hoare triples

\[
\text{Definition hoare\_triple (P:Assertion) (c:com) (Q:Assertion) : Prop :=} \\
\quad \forall \text{st st', c / st} \mid\mid \text{st'} -> \text{P st} -> \text{Q st'}. \\
\]

\[
\text{Notation "\{\{ P \}\} c \{\{ Q \}\}" := (hoare\_triple P c Q).} \\
\]
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

\[ \{\text{assn\_sub } X a \ Q \} X := a \{Q\} \]  \hspace{1cm} (hoare\_asgn)

\[ \{P\} \text{SKIP} \{P\} \]  \hspace{1cm} (hoare\_skip)

\[ \{P\} c_1 \{Q\} \]
\[ \{Q\} c_2 \{R\} \]
\[ \{P\} c_1; c_2 \{R\} \]  \hspace{1cm} (hoare\_seq)

\[ \{P \land b\} c_1 \{Q\} \]
\[ \{P \land \neg b\} c_2 \{Q\} \]
\[ \{P\} \text{IFB } b \text{ THEN } c_1 \text{ ELSE } c_2 \text{ FI } \{Q\} \]  \hspace{1cm} (hoare\_if)

\[ \{P \land b\} c \{P\} \]
\[ \{P\} \text{WHILE } b \text{ DO } c \text{ END } \{P \land \neg b\} \]  \hspace{1cm} (hoare\_while)

\[ \{P'\} c \{Q'\} \]
\[ P \rightarrow P' \]
\[ Q' \rightarrow Q \]
\[ \{P\} c \{Q\} \]  \hspace{1cm} (hoare\_consequence)

\[ \{P'\} c \{Q\} \]
\[ P \rightarrow P' \]
\[ \{P\} c \{Q\} \]  \hspace{1cm} (hoare\_consequence\_pre)

\[ \{P\} c \{Q'\} \]
\[ Q' \rightarrow Q \]
\[ \{P\} c \{Q\} \]  \hspace{1cm} (hoare\_consequence\_post)
Decorated programs

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

\[
\begin{align*}
\{\{ \ P \ \}\} \\
\text{SKIP} \\
\{\{ \ P \ \}\}
\end{align*}
\]

(b) The sequential composition of \(c_1\) 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 \(R\)):

\[
\begin{align*}
\{\{ \ P \ \}\} \\
c_1; \\
\{\{ \ Q \ \}\} \\
c_2 \\
\{\{ \ R \ \}\}
\end{align*}
\]

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

\[
\begin{align*}
\{\{ \ P \ [X |-> a] \ \}\} \\
X ::= a \\
\{\{ \ P \ \}\}
\end{align*}
\]

(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 \(P \land b\) and \(P \land \neg b\) and if its "then" branch is locally consistent (with respect to \(P \land b\) and \(Q\)) and its "else" branch is locally consistent (with respect to \(P \land \neg b\) and \(Q\)):

\[
\begin{align*}
\{\{ \ P \ \}\} \\
\text{IFB } b \ \text{THEN} \\
\{\{ \ P \land b \ \}\} \\
c_1 \\
\{\{ \ Q \ \}\} \\
\text{ELSE} \\
\{\{ \ P \land \neg b \ \}\} \\
c_2 \\
\{\{ \ Q \ \}\} \\
\text{FI} \\
\{\{ \ Q \ \}\}
\end{align*}
\]
(e) A while loop with precondition $P$ is locally consistent if its postcondition is $P \land \neg b$ and if the pre- and postconditions of its body are exactly $P \land b$ and $P$:

\[
\{\{ P \}\}\ \\
\text{WHILE } b \text{ DO} \\
\{\{ P \land b \}\} \\
c1 \\
\{\{ P \}\} \\
\text{END} \\
\{\{ P \land \neg b \}\}
\]

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

\[
\{\{ P \}\} \Rightarrow \ \\
\{\{ P' \}\}
\]
Coq formalization of STLC with booleans

Identifiers

Inductive id : Type :=
  Id : nat -> id.

Theorem eq_id_dec : forall id1 id2 : id, {id1 = id2} + {id1 <> id2}.

Definition x := (Id 0).
Definition y := (Id 1).
Definition z := (Id 2).

Types

Inductive ty : Type :=
  | TBool : ty
  | TArrow : ty -> ty -> ty.

Terms

Inductive tm : Type :=
  | tvar : id -> tm
  | tapp : tm -> tm -> tm
  | tabs : id -> ty -> tm -> tm
  | ttrue : tm
  | tfalse : tm
  | tif : tm -> tm -> tm -> tm.

Inductive value : tm -> Prop :=
  | v_abs : forall x T t, value (tabs x T t)
  | v_true :
    value ttrue
  | v_false :
    value tfalse.

Substitution

Reserved Notation "'[ x :=' s '']' t" (at level 20).

Fixpoint subst (x:id) (s:tm) (t:tm) : tm :=
  match t with
  | tvar x' =>
    if eq_id_dec x x' then s else t
  | tabs x' T t1 =>
    tabs x' T (if eq_id_dec x x' then t1 else ([x:=s] t1))
| tapp \ t1 \ t2 \Rightarrow 
  \ tapp ([x:=s] \ t1) ([x:=s] \ t2) \\
| ttrue \Rightarrow 
  \ ttrue \\
| tfalse \Rightarrow 
  \ tfalse \\
| tif \ t1 \ t2 \ t3 \Rightarrow 
  \ tif ([x:=s] \ t1) ([x:=s] \ t2) ([x:=s] \ t3) 
end 

where "'[ ' x ':=' s ' ] ' t" := (subst x s t).

Reduction

Reserved Notation "t1 '==>' t2" (at level 40).

Inductive step : tm -> tm -> Prop :=
  | ST_AppAbs : forall x T t12 v2, 
    value v2 -> 
    (tapp (tabs x T t12) v2) ==> [x:=v2]t12 \\
  | ST_App1 : forall t1 t1' t2, 
    t1 ==> t1' -> 
    tapp t1 t2 ==> tapp t1' t2 \\
  | ST_App2 : forall v1 t2 t2', 
    value v1 -> 
    t2 ==> t2' -> 
    tapp v1 t2 ==> tapp v1 t2' \\
  | ST_IfTrue : forall t1 t2, 
    (tif ttrue t1 t2) ==> t1 \\
  | ST_IfFalse : forall t1 t2, 
    (tif tfalse t1 t2) ==> t2 \\
  | ST_If : forall t1 t1' t2 t3, 
    t1 ==> t1' -> 
    (tif t1 t2 t3) ==> (tif t1' t2 t3) 
where "t1 '==>' t2" := (step t1 t2).

Definition normal_form {X:Type} (R:relation X) (t:X) : Prop := 
  ~ exists t', R t t'.

Contexts

Definition partial_map (A:Type) := id -> option A.

Definition empty {A:Type} : partial_map A := (fun _ => None).

22
Definition extend \{A:Type\} (Gamma : partial_map A) (x:id) (T : A) :=
  fun x' => if eq_id_dec x x' then Some T else Gamma x'.

Typing Relation

Reserved Notation "Gamma '|-' t \"in\' T" (at level 40).

Inductive has_type : context -> tm -> ty -> Prop :=
  | T_Var : forall Gamma x T,
    Gamma x = Some T ->
    Gamma |- tvar x \in T
  | T_Abs : forall Gamma x T11 T12 t12,
    extend Gamma x T11 |- t12 \in T12 ->
    Gamma |- tabs x T11 t12 \in TArrow T11 T12
  | T_App : forall T11 T12 Gamma t1 t2,
    Gamma |- t1 \in TArrow T11 T12 ->
    Gamma |- t2 \in T11 ->
    Gamma |- tapp t1 t2 \in T12
  | T_True : forall Gamma,
    Gamma |- ttrue \in TBool
  | T_False : forall Gamma,
    Gamma |- tfalse \in TBool
  | T_If : forall t1 t2 t3 T Gamma,
    Gamma |- t1 \in TBool ->
    Gamma |- t2 \in T ->
    Gamma |- t3 \in T ->
    Gamma |- tif t1 t2 t3 \in T

where "Gamma '|-' t \"in\' T" := (has_type Gamma t T).
STLC with subtyping

Typing relation

\[ \Gamma x = T \]
\[ \Gamma \vdash x \in T \quad (T_{\text{Var}}) \]
\[ \Gamma, x : T_{11} \vdash t_{12} \in T_{12} \]
\[ \Gamma \vdash \lambda x : T_{11}. t_{12} \in T_{11} \rightarrow T_{12} \quad (T_{\text{Abs}}) \]
\[ \Gamma \vdash t_{1} \in T_{11} \rightarrow T_{12} \]
\[ \Gamma \vdash t_{2} \in T_{11} \]
\[ \Gamma \vdash t_{1} t_{2} \in T_{12} \quad (T_{\text{App}}) \]
\[ \Gamma \vdash \text{true} \in \text{Bool} \quad (T_{\text{True}}) \]
\[ \Gamma \vdash \text{false} \in \text{Bool} \quad (T_{\text{False}}) \]
\[ \Gamma \vdash t_{1} \in \text{Bool} \quad \Gamma \vdash t_{2} \in T \quad \Gamma \vdash t_{3} \in T \]
\[ \Gamma \vdash \text{if } t_{1} \text{ then } t_{2} \text{ else } t_{3} \in T \quad (T_{\text{If}}) \]
\[ \Gamma \vdash t \in S \quad S <: T \]
\[ \Gamma \vdash t \in T \quad (T_{\text{Sub}}) \]

Subtyping relation

\[ S <: U \quad U <: T \]
\[ S <: T \quad (S_{\text{Trans}}) \]
\[ S <: T \]
\[ T <: T \quad (S_{\text{Ref1}}) \]
\[ T <: T \]
\[ S <: \text{Top} \quad (S_{\text{Top}}) \]
\[ T_{1} <: S_{1} \quad S_{2} <: T_{2} \]
\[ S_{1} \rightarrow S_{2} <: T_{1} \rightarrow T_{2} \quad (S_{\text{Arrow}}) \]
STLC subtyping, extended with pairs and records

Subtyping relation

\[
S_1 <: T_1 \quad S_2 <: T_2
----------------------------- (S_{Prod})
S_1 * S_2 <: T_1 * T_2
\]

\[
n > m
----------------------------- (S_{RcdWidth})
\{i_1:T_1...i_n:T_n\} <: \{i_1:T_1...i_m:T_m\}
\]

\[
S_1 <: T_1 \quad ... \quad S_n <: T_n
----------------------------- (S_{RcdDepth})
\{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\}
\]

\{i_1:S_1...i_n:S_n\} is a permutation of \{i_1:T_1...i_n:T_n\}
----------------------------- (S_{RcdPerm})
\{i_1:S_1...i_n:S_n\} <: \{i_1:T_1...i_n:T_n\}