CIS 500 — Software Foundations

Final Exam

May 9, 2011

Answer key

Hoare Logic

1. (7 points) What does it mean to say that the Hoare triple \{\{P\}\} c \{\{Q\}\} is valid?

*Answer:* \{\{P\}\} c \{\{Q\}\} means that, for any states \textit{st} and \textit{st}'', if \textit{st} satisfies \textit{P} and \textit{c} / \textit{st} \downarrow \textit{st}'', then \textit{st}'' satisfies \textit{Q}.

2. (18 points) Recall the Hoare rule for reasoning about sequences of commands:

\[
\begin{align*}
&\text{\{\{P\}\} c1 \{\{Q\}\}} & &\text{\{\{Q\}\} c2 \{\{R\}\}} \\
\hline
&\text{\{\{P\}\} c1;c2 \{\{R\}\}} & &\text{HOARE_SEQ}
\end{align*}
\]

Formally, this rule corresponds to a theorem:

\text{Theorem hoare_seq :} \forall \text{P Q R c1 c2},
\[
\begin{align*}
\text{\{\{P\}\} c1 \{\{Q\}\}} & \rightarrow \\
\text{\{\{Q\}\} c2 \{\{R\}\}} & \rightarrow \\
\text{\{\{P\}\} c1;c2 \{\{R\}\}}.
\end{align*}
\]

Give a careful informal proof (in English) of this theorem.

*Answer:* To show that the Hoare triple \{\{P\}\} c1;c2 \{\{R\}\} is valid, we must show that, for any states \textit{st} and \textit{st}'', if \textit{st} satisfies \textit{P} and \textit{c1;c2} / \textit{st} \downarrow \textit{st}'', then \textit{st}'' satisfies \textit{R}. So suppose \textit{st} satisfies \textit{P} and \textit{c1;c2} / \textit{st} \downarrow \textit{st}'.

By the definition of the evaluation relation, there must be some state \textit{st}'' such that \textit{c1} / \textit{st} \downarrow \textit{st} and \textit{c2} / \textit{st}'' \downarrow \textit{st}''. Since \{\{P\}\} c1 \{\{Q\}\} is a valid triple, we know that \textit{st}'' satisfies \textit{Q}. But then, since \{\{Q\}\} c2 \{\{R\}\} is a valid triple, we know that \textit{st}''' satisfies \textit{R}, as required.
3. (12 points) In the Imp program below, we have provided a precondition and postcondition. In the blank before the loop, fill in an invariant that would allow us to annotate the rest of the program.

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

Answer: Z + Y*n = n*n \land X = n

Grading scheme: Common mistakes included

- Using subtraction to state the invariant without including an assertion about the relative sizes of the things being subtracted (-2 points)
- Omitting X = n from the invariant (-3 points)

Otherwise, 4 points each for:

- invariant holds before the loop
- invariant is preserved by the loop
- invariant and negation of loop test implies final assertion
4. (16 points) Recall the definition of the substitution operation in the simply typed lambda-calculus (with no extensions, and omitting base types such as booleans for brevity):

```coq
Fixpoint subst (s:tm) (x:id) (t:tm) : tm :=
  match t with
  | tm_app t1 t2 => subst s x t1 (subst s x t2)
  | tm_var x' => if beq_id x x' then s else t
  | tm_abs x' T t1 => subst s x t1 (tm_abs x' T (if beq_id x x' then t1 else (subst s x t1))
  end.
```

This definition uses Coq’s Fixpoint facility to define substitution as a function. Suppose, instead, we wanted to define substitution as an inductive relation substi. We’ve begun the definition by providing the Inductive header and one of the constructors; your job is to fill in the rest of the constructors. (Your answer should be such that subst s x t = t' <-> substi s x t t', for all s, x, t, and t', but you do not need to prove it).

Answer:

```coq
Inductive substi (s:tm) (x:id) : tm -> tm -> Prop :=
  | s_app : forall t1 t2 t1' t2', substi s x t1 t1' -> substi s x t2 t2' -> substi s x (tm_app t1 t2) (tm_app t1' t2')
  | s_var1 : substi s x (tm_var x) s
  | s_var2 : forall x', beq_id x x' = false -> substi s x (tm_var x') (tm_var x')
  | s_abs1 : forall T t1, substi s x (tm_abs x T t1) (tm_abs x T t1)
  | s_abs2 : forall x' T t1 t1', beq_id x x' = false -> substi s x t1 t1' -> substi s x (tm_abs x' T t1) (tm_abs x' T t1').
```
5. (12 points) The next few problems concern the STLC extended with natural numbers and references (reproduced on page 13, with the same informal notations as we’re using here).

(a) In this system, is there a type \( T \) that makes

\[
x: T; \emptyset \vdash (\lambda x: \text{Nat}. \text{Nat} \cdot 2 \cdot x) (x \; x) : \text{Nat}
\]

provable? If so, what is it?

*Answer: No.*

(b) Is there a type \( T \) that makes

\[
\text{empty}; \emptyset \vdash (\lambda x: \text{Ref Nat}. ((\_	ext{Unit} \cdot !x), (\lambda y: \text{Nat}. x := y))) (\text{ref 0}) : T
\]

provable? If so, what is it?

*Answer: \((\text{Unit} \rightarrow \text{Nat}) \times (\text{Nat} \rightarrow \text{Unit})\)*

(c) Is there a type \( T \) that makes

\[
x: T; \emptyset \vdash (!(!x)) : \text{Nat}
\]

provable? If so, what is it?

*Answer: \( \text{Ref} \; (\text{Ref} \; \text{Ref Nat}) \)*

(d) Is there a type \( T \) that makes

\[
x: T; \emptyset \vdash (\lambda y: \text{Nat} \times \text{Nat}. \text{pred} (y.\text{fst})) \; (x.\text{snd} \\ x.\text{fst}) : \text{Nat}
\]

provable? If so, what is it?

*Answer: \( S \; \times \; (S \rightarrow \text{Nat} \times \text{Nat}), \text{for any type} \; S.\)*
6. (8 points) Briefly explain the term aliasing. Give one reason why it is a good thing and one reason why it is bad.

Answer: Aliasing can happen in any language with a pointers and a heap. It occurs when two different variables both refer to the same heap cell — or, more generally, when a single heap cell is accessible to a program via multiple “paths” of pointers. It is both a good thing and a bad thing: good, because it is the basis of shared state and so fundamental to mainstream OO programming; but also bad, because it makes reasoning about programs more difficult. For example, executing the assignments \( x := 5; \) \( y := 6 \) leaves \( x \) set to 5 unless \( x \) and \( y \) are aliases for the same cell.
7. (24 points) Recall the preservation theorem for the STLC with references. In formal Coq notation it looks like this:

Theorem preservation : forall ST t t’ T st st’,
  has_type empty ST t T ->
  store_well_typed empty ST st ->
  t / st ==> t’ / st’ ->
  exists ST’,
    (extends ST’ ST /
     has_type empty ST’ t’ T /
     store_well_typed empty ST’ st’).

Informally, it looks like this:

Theorem (Preservation): If empty; ST |- t : T with ST |- st, and t in store st takes a step to t’ in store st’, then there exists some store typing ST’ that extends ST and for which empty; ST’ |- t’ : T and ST’ |- st’.

(a) Briefly explain why the extra (compared to preservation for the pure STLC) refinement “exists ST’...” is needed here.

Answer: Because reducing a term of the form ref v allocates a new location l and yields l as the result of reduction, but l is not in the domain of ST and hence not typable under ST.
The proof of this theorem relies on some subsidiary lemmas:

Lemma store_weakening : forall Gamma ST ST’ t T,  
extends ST’ ST ->  
has_type Gamma ST t T ->  
has_type Gamma ST’ t T.

Lemma store_well_typed_snoc : forall ST st t1 T1,  
store_well_typed ST st ->  
has_type empty ST t1 T1 ->  
store_well_typed (snoc ST T1) (snoc st t1).

Lemma assign_pres_store_typing : forall ST st l t,  
l < length st ->  
store_well_typed ST st ->  
has_type empty ST t (store_ty_lookup l ST) ->  
store_well_typed ST (replace l t st).

Lemma substitution_preserves_typing : forall Gamma ST x s S t T,  
has_type empty ST s S ->  
has_type (extend Gamma x S) ST t T ->  
has_type Gamma ST (subst x s t) T.

Suppose we carry out a proof of preservation by induction on the given typing derivation. In which cases of the proof are the above lemmas used? 
Match names of lemmas to proof cases by drawing a line from from each lemma to each proof case that uses it.

\[
\begin{align*}
&\text{T.Abs} & &\text{store_weakening} \\
&\text{T.App} & &\text{store_well_typed_snoc} \\
&\text{T_Ref} & &\text{assign_pres_store_typing} \\
&\text{T.Deref} & &\text{substitution_preserves_typing}
\end{align*}
\]

Answer:

- \text{store_weakening} is used in the T.App and T.Assign cases.
- \text{store_well_typed_snoc} is used in the T_Ref case.
- \text{assign_pres_store_typing} is used in the T.Assign case.
- \text{substitution_preserves_typing} is used in the T_App case.

Grading scheme: 2 points for each correct line drawn, -1 point for each incorrect line.
(c) Here is the beginning of the T_Ref case of the proof. Complete the case.

Theorem (Preservation): If empty; ST |- t : T with ST |- st, and t in store st takes a step to t’ in store st’, then there exists some store typing ST’ that extends ST and for which empty; ST’ |- t’ : T and ST’ |- st’.

Proof: By induction on the given derivation of empty; ST |- t : T.

• ...cases for other rules...

• If the last rule in the derivation is T_Ref, then t = ref t1 for some t1 and, moreover, empty; ST |- t1 : T1 for some T1, with T = Ref T1.

Answer: There are two cases to consider, one for each rule that could have been used to show that ref t1 takes a step to t’.

– Suppose ref t1 takes a step by ST_Ref, with t1 stepping to t1’ and new store st’. Then by the IH there is some store typing ST’ such that ST’ extends ST, st’ is well typed with respect to ST’, and empty; ST’ |- t1’ : T1. Hence, empty; ST’ |- ref t1’ : Ref T1 by T_Ref.

– Suppose ref t1 takes a step by ST_RefValue. Then t1 is a value, st’ = snoc st t1, and t’ is the length of st – i.e., the location of t1 in st’. If we choose ST’ = snoc ST T1, we obtain

  * ST’ extends ST by construction
  * st’ is well typed with respect to ST’ by the store_well_typed_snoc lemma, and
  * empty; ST’ |- l : Ref T1 by T_Loc, as required.

Grading scheme:

• Correct case analysis of step: 2 points

• ST_Ref case: 4 points

• ST_RefValue case: 4 points
Subtyping

8. (8 points) Recall the simply-typed lambda calculus extended with products and subtyping (reproduced on page 15).

The subtyping rule for products

\[
\begin{array}{c}
S_1 <: T_1 \\
\hline
S_2 <: T_2 \\
\hline
\hline
S_1 \times S_2 <: T_1 \times T_2
\end{array}
\]  

(S_Prod)

intuitively corresponds to the “depth” subtyping rule for records. Extending the analogy, we might consider adding a “permutation” rule

\[
\begin{array}{c}
T_1 \times T_2 :: T_2 \times T_1
\end{array}
\]  

(S_ProdP)

for products.

Is this a good idea? Briefly explain why or why not.

Answer: No, since it will break preservation: \((\mathsf{true}, \mathsf{unit}).1\) has type \(\mathsf{Unit}\) according to this rule, but reduces to \(\mathsf{true}\), which does not have type \(\mathsf{Unit}\).

Grading scheme: 1 point for “no,” remaining points for clarity of explanation why.
9. (15 points) The preservation and progress theorems about the STLC with subtyping (page 15) depend on a number of technical lemmas, including the following one, which describes the possible “shapes” of types that are subtypes of an arrow type:

**Lemma:** For all types $U$, $V_1$, and $V_2$, if $U <: V_1 \rightarrow V_2$, then there exist types $U_1$ and $U_2$ such that

(a) $U = U_1 \rightarrow U_2$,
(b) $V_1 <: U_1$, and
(c) $U_2 <: V_2$.

The following purported proof of this lemma contains two significant mistakes. Explain what is wrong and how the proof should be corrected.

**Proof:** By induction on a derivation of $U <: V_1 \rightarrow V_2$.

- The last rule in the derivation cannot be $S_{\text{Prod}}$ or $S_{\text{Top}}$ since $V_1 \rightarrow V_2$ is not a product type or Top.
- If the last rule in the derivation is $S_{\text{Arrow}}$, all the desired facts follow directly from the form of the rule.
- Suppose the last rule in the derivation is $S_{\text{Trans}}$. Then, from the form of the rule, there is some type $U'$ with $U <: U'$ and $U' <: V_1 \rightarrow V_2$. We must show that $U' = U_1' \rightarrow U_2'$, with $V_1 <: U_1'$ and $U_2' <: V_2$; this follows from the induction hypothesis.

**Answer:**

- The case for $S_{\text{Refl}}$ is omitted. In that case $U = V_1 \rightarrow V_2$ and we have $V_1 <: V_1$ and $V_2 <: V_2$ by $S_{\text{Refl}}$.
- The $S_{\text{Trans}}$ case is incomplete: it does not show anything about $U$. Once we know that $U <: U_1' \rightarrow U_2'$, we must apply the IH again to conclude that $U = U_1 \rightarrow U_2$ where $U_1' <: U_1$ and $U_2' <: U_2'$. Since we also know $V_1 <: U_1'$ and $U_2' <: V_2$, by two applications of $S_{\text{Trans}}$ we conclude that $V_1 <: U_1$ and $U_2 <: V_2$. This finally gives us what we wanted to show.

**Grading scheme:** 5 points for the Refl part, 10 for the Trans part.
For Reference...

IMP programs

Here are the key definitions for the syntax and big-step semantics of IMP programs:

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).
------ (E_Skip) ------

SKIP / st ≽ st

------ (E_Ass) ------
aeval st a1 = n

------ (E_Seq) ------
l := a1 / st ≽ (update st l n)
c1 / st ≽ st'
c2 / st’ ≽ st’’
c1;c2 / st ≽ st’’

------ (E_IfTrue) ------
IF b1 THEN c1 ELSE c2 FI / st ≽ st’
beval st b1 = true
c1 / st ≽ st’

------ (E_IfFalse) ------
IF b1 THEN c1 ELSE c2 FI / st ≽ st’
beval st b1 = false
c2 / st ≽ st’

------ (E_WhileEnd) ------
WHILE b1 DO c1 END / st ≽ st
beval st b1 = true
c1 / st ≽ st’
WHILE b1 DO c1 END / st’ ≽ st’’

------ (E_WhileLoop) ------
WHILE b1 DO c1 END / st ≽ st’’
Hoare logic rules

\[
\begin{align*}
\text{Hoare\_Asgn} & & \begin{array}{c}
\text{Hoare\_Consequence} \\
\text{Hoare\_Pre} & & \text{Hoare\_Post}
\end{array} \\
\text{Hoare\_Skip} & & \text{Hoare\_Seq} \\
\text{Hoare\_If} & & \text{Hoare\_While}
\end{align*}
\]

1. \[\{assn\_sub V a Q\} V := a \{Q\}\]
2. \[\{P'\} c \{Q'\} \quad P \rightarrow P' \quad Q' \rightarrow Q \quad \{P\} c \{Q\}\]
3. \[\{P'\} c \{Q\} \quad P \rightarrow P' \quad \{P\} c \{Q\}\]
4. \[\{P\} \quad \text{SKIP} \quad \{P\}\]
5. \[\{P \land b\} \quad c1 \quad \{Q\} \quad \{P \land \neg b\} \quad c2 \quad \{Q\} \quad \{P\} \quad \text{IFB} b \quad \text{THEN} \quad c1 \quad \text{ELSE} \quad c2 \quad \text{FI} \quad \{Q\}\]
6. \[\{P \land b\} \quad c1 \quad \{P\} \quad \text{WHILE} \quad b \quad \text{DO} \quad c \quad \text{END} \quad \{P \land \neg b\}\]
STLC with references

(Some of the questions concerning STLC with references also use natural numbers and arithmetic operations; the syntax and semantics of these constants and operators is standard.)

Syntax

| T ::= | Unit |
| T -> T |
| Ref T |
| \x:T. t |
| unit |
| ref t |
| !t |
| t := t |
| loc n |

| t ::= | x |
| \x:T. t |
| unit |
| loc n |

| v ::= |

Operational semantics

\[ \text{value } v_2 \]

------------------------------------- (ST_AppAbs)
\[ (\lambda a:T.t_12) v_2 / st \rightarrow [v_2/a]t_12 / st \]

\[ t_1 / st \rightarrow t_1' / st' \] (ST_App1)

\[ t_1 t_2 / st \rightarrow t_1' t_2 / st' \]

\[ \text{value } v_1 \]

---------------------------------- (ST_App2)
\[ v_1 t_2 / st \rightarrow v_1 t_2' / st' \]

--------------------------------- (ST_RefValue)
\[ \text{ref } v_1 / st \rightarrow \text{loc } |st| / st, v_1 \]

---------------------------------- (ST_Ref)
\[ \text{ref } t_1 / st \rightarrow \text{ref } t_1' / st' \]

\[ l < |st| \]

--------------------------------- (ST_DerefLoc)
\[ !(\text{loc }1) / st \rightarrow \text{lookup } 1 \text{ st} / st \]

---------------------------------- (ST_Deref)
\[ t_1 / st \rightarrow t_1' / st' \]

\[ !t_1 / st \rightarrow !t_1' / st' \]
\[
\begin{align*}
l < |st| & \quad \text{(ST_Assign)} \\
l \gets v2 \ / \ st \Rightarrow \text{unit} / (\text{replace } l \ v2 \ st) \\
\text{t1} \ / \ st \Rightarrow \text{t1'} \ / \ st' & \quad \text{(ST_Assign1)} \\
t1 \ = \ t2 \ / \ st \Rightarrow t1' \ = \ t2 \ / \ st' & \quad \text{(ST_Assign2)} \\
t2 \ / \ st \Rightarrow t2' \ / \ st' \\
v1 \ = \ t2 \ / \ st \Rightarrow v1 \ = \ t2' \ / \ st'
\end{align*}
\]

\textbf{Typing}

\[
\begin{align*}
\Gamma \ x = T & \quad \text{(T_Var)} \\
\Gamma; ST \vdash x : T \\
\Gamma, x:T11; ST \vdash t12 : T12 & \quad \text{(T_Abs)} \\
\Gamma; ST \vdash \lambda x:T11. t12 : T11 \rightarrow T12 \\
\Gamma; ST \vdash t1 : T11 \rightarrow T12 \\
\Gamma; ST \vdash t2 : T11 & \quad \text{(T_App)} \\
\Gamma; ST \vdash t1 \ t2 : T12 \\
\Gamma; ST \vdash \text{unit} : \text{Unit} & \quad \text{(T_Unit)} \\
l < |ST| & \quad \text{(T_Loc)} \\
\Gamma; ST \vdash \text{loc} \ l : \text{Ref} (\text{lookup} \ l \ ST) \\
\Gamma; ST \vdash t1 : T1 & \quad \text{(T_Ref)} \\
\Gamma; ST \vdash \text{ref} \ t1 : \text{Ref} T1 \\
\Gamma; ST \vdash t1 : \text{Ref} T11 & \quad \text{(T_Deref)} \\
\Gamma; ST \vdash !t1 : T11 \\
\Gamma; ST \vdash t1 : \text{Ref} T11 \\
\Gamma; ST \vdash t2 : T11 & \quad \text{(T_Assign)} \\
\Gamma; ST \vdash t1 := t2 : \text{Unit}
\end{align*}
\]
STLC with products and subtyping

Syntax

\[ T ::= \text{Top} \]
\[ t ::= x \]
\[ v ::= \x:T. \ t \]
\[ | T \rightarrow T \]
\[ | \m{T} \]
\[ | T \ast T \]
\[ | \x:T. \ t \]
\[ | (t,t) \]
\[ | t.fst \]
\[ | t.snd \]

Operational semantics

------------------------- (ST_AppAbs)
(\a:T.t12) v2 => \[v2/a\]t12

\[ t1 => t1' \]
------------------------- (ST_App1)
\[ t1 t2 => t1' t2 \]

\[ t2 => t2' \]
------------------------- (ST_App2)
\[ v1 t2 => v1 t2' \]

\[ t1 => t1' \]
------------------------- (ST_Pair1)
\[ (t1,t2) => (t1',t2) \]

\[ t2 => t2' \]
------------------------- (ST_Pair2)
\[ (v1,t2) => (v1,t2') \]

\[ t1 => t1' \]
------------------------- (ST_Fst1)
\[ t1.fst => t1'.fst \]

------------------------- (ST_FstPair)
\[ (v1,v2).fst => v1 \]

\[ t1 => t1' \]
------------------------- (ST_Snd1)
\[ t1.snd => t1'.snd \]

------------------------- (ST_SndPair)
\[ (v1,v2).snd => v2 \]
Subtyping

-----
T <: T

S <: U  U <: T
----------
S <: T

-----
S <: Top

T1 <: S1  S2 <: T2
-----------
S1->S2 <: T1->T2

S1 <: T1  S2 <: T2
-----------
S1*S2 <: T1*T2

16
Typing

\begin{align*}
\text{Gamma } x &= T \\
\text{-------------} & \quad \text{(T_Var)} \\
\text{Gamma } \vdash x : T \\
\hline
\text{Gamma, } x : T_{11} & \vdash t_{12} : T_{12} \\
\text{------------------------} & \quad \text{(T_Abs)} \\
\text{Gamma } \vdash \lambda x: T_{11}. t_{12} : T_{11} \to T_{12} \\
\hline
\text{Gamma } \vdash t_{1} : T_{11} \to T_{12} \\
\text{Gamma } \vdash t_{2} : T_{11} \\
\text{------------------------} & \quad \text{(T_App)} \\
\text{Gamma } \vdash t_{1} \ t_{2} : T_{12} \\
\hline
\text{Gamma } \vdash t_{1} : T_{1} \\
\text{Gamma } \vdash t_{2} : T_{2} \\
\text{-----------------------------------------} & \quad \text{(T_Pair)} \\
\text{Gamma } \vdash (t_{1}, t_{2}) : T_{1} \times T_{2} \\
\hline
\text{Gamma } \vdash t_{1} : T_{11} \times T_{12} \\
\text{------------------------} & \quad \text{(T_Fst)} \\
\text{Gamma } \vdash t_{1}. \text{fst} : T_{11} \\
\hline
\text{Gamma } \vdash t_{1} : T_{11} \times T_{12} \\
\text{------------------------} & \quad \text{(T_Snd)} \\
\text{Gamma } \vdash t_{1}. \text{snd} : T_{12} \\
\hline
\text{Gamma } \vdash t : S \\
\text{S <: T} \\
\text{------------------------} & \quad \text{(T_Sub)} \\
\text{Gamma } \vdash t : T
\end{align*}