CIS 500 — Software Foundations

Final Exam

Answer key

May 3, 2010

Induction

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

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

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

Answer:

By induction on a derivation of `frob x y z`:

- Suppose the final rule used to show `frob x y z` is FrobBase. Then $\mathtt{x} = \mathtt{y} = \mathtt{z} = 0$. We must show $0 + 0 \leq 0$, which is immediate.
- Suppose the final rule was FrobInc. Then $\mathtt{z} = \mathtt{S}\ \mathtt{z}'$ for some $\mathtt{z}$ with $\mathtt{x} + \mathtt{y} \leq \mathtt{z}'$. We must show $\mathtt{x} + \mathtt{y} \leq \mathtt{S}\ \mathtt{z}'$, which is a simple arithmetic fact.
- Suppose the final rule was Frob1. Then $\mathtt{x} = \mathtt{S}\ \mathtt{x}'$ and $\mathtt{z} = \mathtt{S}\ \mathtt{z}'$ for some $\mathtt{x}'$ and $\mathtt{z}'$ with $\mathtt{x}' + \mathtt{y} \leq \mathtt{z}'$. We must show $\mathtt{S}\ \mathtt{x} + \mathtt{y} \leq \mathtt{S}\ \mathtt{z}'$, which again is a simple arithmetic fact.
- Suppose the final rule was Frob2. This case is similar to Frob1.

Logic in Coq

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

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

Answer:

```
conj : P → Q → (and P Q).
```

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

Answer:

```
| or_introl : P → or P Q
| or_intror : Q → or P Q.
```

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

Answer:

```
ex_intro : forall (witness:X), P witness → ex X P.
```

Hoare Logic

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

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

...and its imperative realization in IMP:

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

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

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

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

Answer: No. The correct invariant is `rev X ++ Y = rev l`*.*

Grading scheme: Binary grading; 3 pts for stating "no", 6 pts for stating correct invariant.

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

   (a)
```
    {{ ? }}
WHILE Y <= X DO
  X := X - 1
END
    {{ Y > X }}
```
   Answer:
```
True
```
   (b)
```
    {{ ? }}
IF X > 3 THEN Z := X ELSE Z := Y FI
    {{ Z = W }}
```
   Answer:
```
(X > 3 → X = W) ∧ (X <= 3 → Y = W)
```
   or equivalently
```
(X > 3 ∧ X = W) ∨ (X <= 3 ∧ Y = W)
```
   (c)
```
    {{ ? }}
WHILE IsCons X DO
   N := N + 1;
   X := Tail(X)
DONE
    {{ X = [] ∧ N = length l }}
```
   Answer:
```
N + length X = length l
```

Grading scheme: 3 points each. Partial credit given for preconditions that were correct but not the weakest possible. However, full credit was given on part (c) for the answer ($N = 0 \wedge$ length $X =$ length l).

Simply Typed Lambda Calculus

The next two problems concern the STLC extended with natural numbers, pairs, and fixpoints, defined formally on page 17 in the Appendix.

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

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

   Answer: `(A→A)*(Nat→A)`

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

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

   Answer: `Nat`

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

   Answer: No such `T`*.*

   Grading scheme: 2 points per type.

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

```
                Gamma |- t1 : T1→T1
                --------------------               (T_Fix)
                Gamma |- fix t1 : T1

                     t1 --> t1'
                 ------------------                (ST_Fix1)
                 fix t1 --> fix t1'

   -------------------------------------------     (ST_FixAbs)
   fix (\x:T1.t2) --> [(fix(\x:T1.t2)) / x] t2
```

Consider the following Coq `Fixpoint` definition of exponentiation:

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

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

Answer:

```
fix (\f:Nat → Nat → Nat.
  \b:Nat. \e:Nat.
    if0 e 1 (b * f b (pred e)))
```

Grading scheme: -1 for minor errors; -2 for getting typing of fix wrong; 0 to 2 points for answers with more serious problems.

References

The definition of the STLC extended with references can be found on page 20 of the Appendix, together with critical auxiliary definitions such as the `well_typed_store` and `extends` relations.

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

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

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

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

```
[6, 7]
```

one possible program to build this store would be:

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

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

(a) `[5, loc 0]`

Answer:

```
ref (ref 5)

[Nat, Ref Nat]
```

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

Answer:

```
let r = ref (ref 5) in
r := ref 4

[Nat, Ref Nat, Nat]
```

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

Answer:

```
let r = (ref (\x : Unit . x)) in
r := (\x : Unit . (!r) unit)

[Unit → Unit]
```

Grading scheme: Each subpart is 1pt for store typing, 3pts for program. -2 for writing "loc" in program (which is not typable in the empty store typing); -1 for other minor errors. In part (ii), -2 for not using assignment to create a circular store. In part (iii), -1 for not initializing the location with a dummy function; -2 for getting assignment of the real function wrong.

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

Answer:

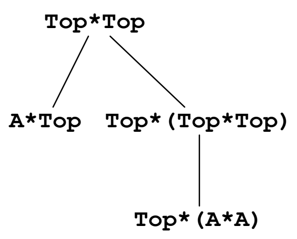
```
Theorem preservation : forall ST t t' T st st',
  has_type empty ST t T →
  store_well_typed empty ST st →
  t / st --> t' / st' →
  exists ST',
    (extends ST' ST ∧
     has_type empty ST' t' T ∧
     store_well_typed empty ST' st').

Theorem progress : forall ST t T st,
  has_type empty ST t T →
  store_well_typed empty ST st →
  (value t ∨ exists t', exists st', t / st --> t' / st').
```

Grading scheme: 4 points per theorem. Partial credit given for statements that had the right idea but missing various premises or conclusions.

Subtyping

The remaining questions on the exam concern the simply typed lambda calculus extended with products and subtyping, described formally on page 23 in the Appendix.

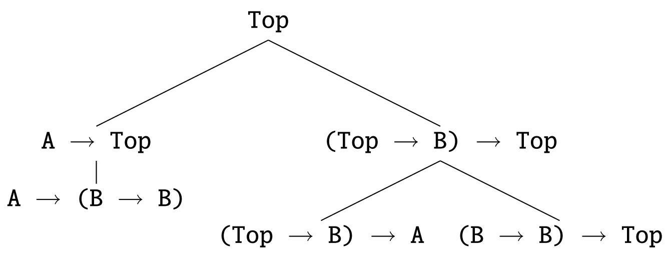
9. (6 points) The subtyping relations among a collection of types can be visuaized compactly in tree form: we draw the tree so that `S <: T` iff `S` is below `T` (either directly or indirectly). For example, a tree for the types `Top*Top`, `A*Top`, `Top*(Top*Top)`, and `Top*(A*A)` would look like this:



Draw a tree for the following six types.

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

Answer:



Grading scheme: Generally, 2 points off for each misplaced type.

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

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

    (a)
```
       T2 <: T1
   ----------------           (S_Ref_Wrong1)
   Ref T1 <: Ref T2
```
    Answer: If a context is expecting a `Ref T2`*, this rule says that it can be safely given a* `Ref T1`*. it may then dereference the pointer and obtain a* `T1` *when it was expecting a* `T2`*; but we only know that it is safe to use a* `T2` *when a* `T1` *is expected, not the other way around.*

    The following term would be well-typed, since `{x:Nat} <: {}` *and thus* `Ref {} <: Ref {x:Nat}`*, so by the subsumption rule we may give* `ref {}` *the type* `Ref {x:Nat}` *and pass it as an argument to the lambda. However, executing this program will result in trying to project the* `x` *field from an empty record.*
```
(\r:Ref {x:Nat}. (!r).x) (ref {})
```
    (b)
```
       T1 <: T2
   ----------------           (S_Ref_Wrong2)
   Ref T1 <: Ref T2
```
    Answer: If a context expecting a `Ref T2` *is given a* `Ref T1`*, it can then store a value of type* `T2` *into the reference cell. Then any other code dereferencing the* `Ref T1` *later will get a* `T2` *when it expected a* `T1`*—but we only know that it is safe to use a* `T1` *when a* `T2` *is expected, not the other way around.*

    The following term would be well-typed: since `{x:Nat} <: {}`*, we have* `Ref {x:Nat} <: Ref {}`*, so we may give the reference* `r` *the type* `Ref {}` *and assigns the empty record into its cell. Later trying to project the* `x` *field will now fail.*
```
(\r:Ref {x:Nat}. r := {}; (!r).x) (ref {x=5})
```

11. (12 points) This problem asks you to consider the possible consequences if we add to this language (STLC with products and subtyping) a reduction rule of the form
```
----------                (ST_Unit)
unit --> ?
```
    where `?` is some term.

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

    (a) Preservation

    Answer: We can break preservation by replacing `?` *by a value of a different type, such as* `\x.x`*.*

    (b) Progress

    Answer: We can never break progress with additional reduction rules.

    (c) Normalization of well-typed terms

    Answer: We can break normalization by, for example, replacing `?` *by* `unit`*.*

    Grading scheme: 4 points each.

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

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

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

> **Lemma [Canonical forms of arrow types]:** If `Gamma |- s : T1→T2` and `s` is a value, then `s = tm_abs x S1 s2` for some `x`, `S1`, and `s2`.
>
> **Lemma [Canonical forms of product types]:** If `Gamma |- s : T1*T2` and `s` is a value, then `s = tm_pair s1 s2` for some `s1` and `s2`.

Answer:

Let `t` and `T` be given such that `empty |- t : T`. Proceed by induction on the typing derivation. Cases `T_Abs` and `T_Unit` are immediate because abstractions and `unit` are always values. Case `T_Var` is vacuous because variables cannot be typed in the empty context. The remaining cases are as follows:

- If the last step in the typing derivation is by `T_App`, then there are terms `t1`, `t2` and types `T1`, `T2` such that `t = t1 t2`, `T = T2`, `empty |- t1 : T1 → T2` and `empty |- t2 : T1`.
  The induction hypotheses for these typing derivations yield that `t1` is a value or steps, and that `t2` is a value or steps. We consider each case:
  - Suppose `t1 -> t1'` for some term `t1'`. Then `t1 t2 -> t1' t2` by `ST_App1`.
  - Otherwise `t1` is a value.
    * Suppose `t2 -> t2'` for some term `t2'`. Then `t1 t2 -> t1 t2'` by rule `ST_App2` because `t1` is a value.
    * Otherwise, `t2` is a value. By the canonical forms of arrow types lemma, `t1 = \x:S1.s2` for some `x`, `S1`, and `s2`. And `(\x:S1.s2) t2 -> [t2/x]s2` by `ST_AppAbs`, since `t2` is a value.
- If the last step in the typing derivation is by `T_Pair`, then there are terms `t1`, `t2` and types `T1`, `T2` such that `t = (t1,t2)`, `T = T1 * T2`, `empty |- t1 : T1`, and `empty |- t2 : T2`.
  The induction hypotheses for these typing derivations yield that `t1` is a value or steps, and that `t2` is a value or steps. We consider each case:
  - Suppose `t1 -> t1'` for some term `t1'`. Then `(t1,t2) -> (t1',t2)` by `ST_Pair1`.
  - Otherwise `t1` is a value.
    * Suppose `t2 -> t2'` for some term `t2'`. Then `(t1,t2) -> (t1,t2')` by rule `ST_Pair2` because `t1` is a value.
    * Otherwise, `t1` and `t2` are both values; then `(t1,t2)` is also a value by `v_pair`.
- If the last step in the typing derivation is by `T_Fst`, then there is a term `tp` and types `T1`, `T2` such that `t = tp.fst`, `T = T1`, and `empty |- tp : T1 * T2`.
  The induction hypothesis for this typing derivation yields that `tp` is either a value or steps.
  - Suppose `tp -> tp'` for some term `tp'`. Then `tp.fst -> tp'.fst` by `ST_Fst1`.
  - Otherwise, `tp` is a value. Then by the canonical form of product types lemma, `tp = tm_pair t1 t2` for some `t1` and `t2`. Since `tp` is a value, by inversion of the `value` judgment `t1` and `t2` must be values as well; therefore `(t1,t2).fst -> t1` by `ST_FstPair`.
- The case for `T_Snd` is exactly analogous to the case for `T_Fst`.
- If the final step of the derivation is by `T_Sub`, then there is a type `S` such that `S <: T` and `empty |- t : S`. The desired result is exactly the induction hypothesis for the typing subderivation.

Grading scheme: 0-6 points for missing or very garbled proofs. 7-13 points for proofs that included most of the important ideas but weren't put together completely correctly. 14-18 points for mostly correct proofs, with small deductions for local problems.