CIS 500 — Software Foundations

Final Exam

Answer key

May 6, 2009

Coq

1. (5 points) Briefly describe the difference between a proof object and a tactic script. How are the two related? (Your answer should not be longer than two or three sentences.)

   Answer: A proof object is a functional program that can be intepreted as evidence for a particular proposition (which is its type). A tactic script is a sequence of commands which constructs a proof object (and which, if well written, helps the user understand the structure of the proof).

   Grading scheme: 1 point for saying that a proof object is (or manipulates) evidence. 1 point for saying that proof objects are evidence for propositions/lemmas/theorems/etc. 1 point for saying that a tactic is not itself evidence. 2 points for saying that tactics generate proof objects.

2. (5 points) Coq's built-in programming language is restricted so that all `Fixpoint` definitions terminate.

   To make Coq more suitable for general-purpose programming, we might think of removing this restriction—i.e., removing the requirement that `Fixpoint` definitions be annoted with a `{struct ...}` declaration and the check that the designated argument is structurally decreasing on all recursive calls.

   Would there be any disadvantages to doing this? Briefly explain.

   Answer: Removing this restriction would make Coq's logic inconsistent: every proposition would become provable. Indeed, we could write a function

   ```
   Fixpoint bogus_proof_of (P : Prop) : P := fix P.
   ```

   such that `bogus_proof_of P` *has type* `P`*, for any* `P`*—i.e.,* `bogus_proof_of P` *would be evidence for proposition* `P`*, no matter what* `P` *is!*

   Grading scheme: 1 point for mentioning nontermination. (Up to) 2 points for saying that this makes programs harder to reason about or connecting the nontermination with difficulty in reasoning. (Up to) 2 points for explaining inconsistency. Points were also awarded if the answer demonstrated knowledge about the ramifications without mentioning inconsistency directly (e.g. if they said something like "all proofs involving fixpoints would need an additional proof of termination to be valid".

Logic

3. (12 points) Suppose we make the following inductive definition:

```
Inductive foo (X:Set) : Set :=
 | c1 : foo X
 | c2 : X → foo X → foo X
 | c3 : list (foo X) → foo X.
```

Fill in the blanks to complete the induction principle that will be generated by Coq.

```
foo_ind
     : forall (X : Set) (P : foo X → Prop),

       ____________________________________________________________ →

       (forall (x : X) (f : foo X), ____________________________) →

       (____________________________________________________________) →

       ____________________________________________________________
```

```
foo_ind
     : forall (X : Set) (P : foo X → Prop),
       P (c1 X) →
       (forall (x : X) (f : foo X), P f → P (c2 X x f)) →
       (forall l : list (foo X), P (c3 X l)) →
       forall f2 : foo X, P f2
```

Grading scheme: By line:

- *1 point each for* `P`, `c1`, `X`
- *1 point each for* `P f`, `P`, `c2`, *and* `X x f`
- *1 point each for* `P`, `c3`, *and* `X l`
- *1 point each for* `foo X` *and* `P f2`

While programs and Hoare Logic

4. (5 points) State the definition of *program equivalence* for while programs.

   Answer: Two while programs `c1` *and* `c2` *are said to be equivalent when* `c1` *maps* `st` *to* `st'` *iff* `c2` *does, for every pair of states* `st` *and* `st'`*.*

   Grading scheme: -2 points for a definition which doesn't handle non-termination properly. -1 or -2 points for unclear or imprecise answers.

5. (6 points) Which of the following pairs of programs are equivalent? Write "yes" or "no" for each one.

   (a)
   ```
   while (A1 <<= !X) do X ::= !X +++ A1
   ```
   and
   ```
   while (A2 <<= !X) do X ::= !X +++ A1
   ```

   Answer: No. (When started in a state where variable `X` *has value* `1`*, the first program diverges while the second one halts.)*

   (b)
   ```
   while BTrue do (while BFalse do X ::= !X +++ A1)
   ```
   and
   ```
   while BFalse do (while BTrue do X ::= !X +++ A1)
   ```

   Answer: No. (The first program always diverges; the second always halts.))

   Grading scheme: 3 points (binary) each. (Just writing "No" was enough. When there was a commentary, we corrected it if it was wrong, but still gave all the points if the "No" was clearly present at the beginning. Everything else received zero.)

6. (8 points) Give the weakest precondition for each of the following commands. (Use the informal notation for assertions rather than Coq notation—i.e., write `X = 5`, not `fun st => st X = 5`.)

(a)
```
{{ ? }}  X ::= A5  {{ X = 6 }}
```
Answer: `False`

(b)
```
{{ ? }}
  testif !X <<= !Y
    then skip
    else Z ::= !Y; Y ::= !X; X ::= !Z
{{ X <= Y }}
```
Answer: `True`

(c)
```
{{ ? }} while BNot (!Y === A5) do Y ::= !Y +++ A1 {{ Y = 5 }}
```
Answer: `True`

(d)
```
{{ ? }} while !X === A0 do Y ::= A1 {{ Y = 5 }}
```
Answer: `X=0` $\vee$ `Y=5`

Grading scheme: 2 points (binary) for a, b, c. 1 or 2 points for d. If the answer was logically equivalent to the correct one, then the whole points (2 points) were given. For d, we gave one point for some "close" answers: `X<>0` $\wedge$ `Y=5`, `X=0`, *or* `Y=5`.

7. (8 points) For each of the `while` programs below, we have provided a precondition and postcondition. In the blank before each loop, fill in an invariant that would allow us to annotate the rest of the program. Assume `X`, `Y` and `Z` are distinct program variables.

(a)

```
                                    { True }
    X ::= ANum n;
    Y ::= A1;
                                    { ____________________________________ }
    while (BNot (!X === A0)) do (
         Y ::= !Y *** A2;
         X ::= !X --- A1
    )
                                    { Y = 2^n }
```

Answer: `Y = 2^(n-X)`

(b)

```
                                    { True }
    X ::= ANum x;
    Y ::= ANum y;
    Z ::= A0;
                                    { ____________________________________ }
    while (BAnd (BNot (!X === A0)) (BNot (!Y === A0))) do (
         X ::= !X --- A1;
         Y ::= !Y --- A1;
         Z ::= !Z +++ A1
    )
                                    { Z = min(x,y) }
```

(where `min` is the mathematical "minimum" function)

Answer: `Z = min(x,y) - min(X,Y)`

Grading scheme: We used the same schema for both parts. We checked if the given "invariant" held in the following cases: before the while (1 point), for a step of the while, assuming the invariant held before (1 point), and for the implication of the postcondition, using the condition and invariant (2 points). In cases that seemed confused about the concept of invariant, we gave no points. (For example: writting the postcondition of the variables' initialisation. In some particular cases, the right answer was surrounded by some garbage, sometimes representing the postcondition of the variables' initialisation.) For example: (a) `X=n ∧ Y=1`*: 0 points. (a)* `X=n` *or* `Y=2^(n-X)`*: 4 points (b)* `x = X + Z ∧ y = Y + Z`*: 4 points. (b)* `Z = min(x-X, y-Y)`*: 2 points.*

Small-Step and Big-Step Operational Semantics

8. (6 points) Briefly explain the difference between big-step and small-step styles of operational semantics. What are the advantages of each style?

   Answer: The big-step style directly relates a term to the final result of its evaluation; the small-step style relates a term to a "slightly more reduced" term in which a single subphrase has taken a single step of computation. Big-step definitions tend to be shorter and are sometimes easier to read; their major disadvantage is that they conflate terms that have no result because they diverge and terms that have no result because their evaluation encounters an undefined state. Small-step definitions are sometimes preferred because they are closer to implementations; in particular, they explicitly expose order of evaluation.

   Grading scheme: 1 point for saying that big steps relate terms to final values. 1 point for an advantage (e.g. big step semantics are often more concise). 1 for saying that that small steps describe the intermediate states of computation. 1 point for advantage (e.g. that it doesn't conflate "nonterminating" with "stuck"). 2 "fudge points" for not saying anything bogus and for giving clear and precise explanations of the above.

Simply Typed Lambda-Calculus

In this section of the exam, we consider the simply typed lambda-calculus with natural numbers, product types, and the fixpoint operator **fix** (but *not* records). The formal definition of this system is given in the accompanying handout.

9. (4 points)

   (a) Is there a type `T` that makes

   ```
   x:T |- if0 ((\x:nat. pred x) x.fst) then x.snd else (x.fst, x.fst) : (nat * nat)
   ```

   provable? If so, what is it?
   Answer: Yes: `T = nat * (nat * nat)`.

   (b) Are there types `S` and `T` that make

   ```
   empty |- (\x:T. \y:T. x y) : S
   ```

   provable? If so, what are they?
   Answer: No; it would have to be the case that `T = T → S`*, but there can be no such (finite) type* `T`.

10. (9 points)

    (a) Suppose we add a term `foo` with the following evaluation rules:

    ```
    (\x:A. x) ~~> foo                (ST_Foo1)

    foo ~~> 0                        (ST_Foo2)
    ```

    Do progress and preservation continue to hold after this change, or does one (or do both) fail? Why?

    Answer: Preservation fails, since we have no typing rules for `foo` *but* `\x:A. x` *has type* `A→A`*. Progress still holds: we are only adding to the step relation, and this can never damage progress.*

    (b) Suppose we add a term `zap`, with the following evaluation rule

    ```
    t ~~> zap                        (ST_Zap)
    ```

    and the following typing rule:

    ```
    Gamma |- zap : T                 (T_Zap)
    ```

    Do progress and preservation continue to hold after this change, or does one (or do both) fail? Why?

    Answer: Both properties continue to hold. Progress holds trivially: every term can take a step to `zap`*! Preservation holds because* `zap` *can have any type.*

    (c) Suppose we change `ST_AppAbs` to the following rule:

    ```
    (\x:T. t12) t2 ~~> (subst x t2 t12)          (ST_AppAbs')
    ```

    Do progress and preservation continue to hold after this change, or does one (or do both) fail? Why?

    Answer: Both properties continue to hold. (Substitution preserves typing irrespective of whether the term being substituted into another term is a value or not.)

    Grading scheme: There were 3 subproblems. Each asked about whether (a) preservation holds (b) why? (c) whether progress holds, and (d) why? Each part (a)-(d) was worth 0.75 points; we rounded to get an integral score.

11. (6 points) Give a term in the simply typed lambda-calculus with `succ`, `pred`, `0`, `if0`, and `fix` that adds numbers — that is, give a term `add` of type `nat→nat→nat` such that `add m n` reduces to the sum of `m` and `n`.

    Please use informal notation (`\x:T.t`, not `tm_abs x T t`).

    Answer:

    ```
    add =
      fix (\a:nat→nat→nat.
             \m:nat. \n:nat.
                if0 m then n else a (pred m) (succ n))
    ```

    Grading scheme: 1 point for `a`*'s type, 2 points for properly using* `fix`*, and* `3` *poitns for the right recursive logic*

12. (20 points) Here is the statement of the preservation theorem for the simply typed lambda-calculus with products and `fix`, along with the statement of the substitution lemma, on which it immediately depends. Fill in the blanks in the proof of preservation. (You do not have to prove the substitution lemma.)

```
Lemma substitution_preserves_typing : forall Gamma x U v t S,
     has_type (extend Gamma x U) t S
  → has_type empty v U
  → has_type Gamma (subst x v t) S.

Theorem preservation : forall t t' T,
     has_type empty t T
  → t ~~> t'
  → has_type empty t' T.
```

Proof: By induction on the given typing derivation. The cases involving numbers (`T_Nat`, `T_Succ`, `T_Pred`, `T_Mult`, and `T_If0`) are omitted.

- If the last rule used was `T_App`, then `t = t1 t2`, with `empty |- t1 : T1→T` and `empty |- t2 : T1`.
  - If the final rule in the derivation of `t ~~> t'` is `ST_App1`, then `t1 ~~> t1'` and `t' = t1' t2`. By the IH, `empty |- t1' : T1→T`. By `T_App`, `empty |- t1' t2 : T`, as required.
  - If the final rule in the derivation of `t ~~> t'` is `ST_App2`, then `t1` is a value, `t2 ~~> t2'`, and `t' = t1 t2'`. By the IH, `empty |- t2' : T1`. By `T_App`, `empty |- t1 t2' : T`, as required.
  - If the final rule in the derivation of `t ~~> t'` is `ST_AppAbs`, then `t2` is a value, `t1 = tm_abs x T11 t12`, and `t' = subst x t2 t12`. We know by assumption that `empty |- tm_abs x T11 t12 : T1→T` and since (by inspection of the typing rules) the only way this could have been proved is by using rule `T_Abs`, it follows that `x:T1 |- t12 : T`. By Lemma `substitution_preserves_typing`, `empty |- subst x t2 t12 : T`, as required.
- If the last rule used was `T_Pair`... *then* `t = (t1,t2)` *and* `T = T1*T2`, *with* `empty |- t1 : T1` *and* `empty |- t2 : T2`.
  - *If the final rule in the derivation of* `t ~~> t'` *is* `ST_Pair1`, *then* `t1 ~~> t1'` *and* `t' = (t1',t2)`. *By the IH,* `empty |- t1' : T1`, *and the result follows by* `T_Pair`.
  - *If the final rule in the derivation of* `t ~~> t'` *is* `ST_Pair2`, *then* `t2 ~~> t2'` *and* `t' = (t1,t2')`. *By the IH,* `empty |- t2' : T1`, *and the result follows by* `T_Pair`.
- If the last rule used was `T_Fst`... *then* `t = t1.fst`, *with* `empty |- t1 : T*T2` *for some* `T2`.
  - *If the final rule in the derivation of* `t ~~> t'` *is* `ST_Fst1`, *then* `t1 ~~> t1'` *and* `t' = t1'.fst`. *By the IH,* `empty |- t1' : T*T2` *and the desired result follows by* `T_Fst`.
  - *If the final rule in the derivation of* `t ~~> t'` *is* `ST_FstPair`, *then* `t1 = (v1,v2)` *for some* `v1` *and* `v2`, *and* `t' = v1`. *But the only way* `empty |- (v1,v2) : T*T2` *could have been proved is by rule* `T_Pair`, *so it must be that* `empty |- v1 : T` *and* `empty |- v2 : T2`. *The first of these is the desired result.*
- The case where the last rule used was `T_Snd` is similar to the `T_Fst` case.
- If the last rule was `T_Fix`... *then* `t = fix t1`, *with* `empty |- t1 : T→T`. *There are two possibilities for the final rule in the derivation of* `t ~~> t'`: `ST_Fix1` *and* `ST_FixAbs`.
  - *If the final rule in the derivation of* `t ~~> t'` *is* `ST_Fix1`, *then* `t1 ~~> t1'`. *But then, by the IH,* `empty |- t1' : T→T`, *from which the result follows by* `T_Fix`.
  - *If the final rule in the derivation of* `t ~~> t'` *is* `ST_FixAbs`, *then* `t1 = \x:T1.t12` *and* `t' = subst x t t12`. *The only way to prove* `empty |- t1 : T→T` *is if* `T1 = T` *and* `x:T |- t12 : T`. *Since* `empty |- t : T`, *Lemma* `substitution_preserves_typing` *gives us* `empty |- subst x t t12 : T`, *as required.* □

Grading scheme: -2 for not indicating which typing rules were being applied. -2 for not indicating inversions. -1 for not showing how `T` *is constrained in each case. -4 for assuming (in the* `T_Fst` *case) that* `t` *is a pair. -4 for missing the* `FixAbs` *subcase; more for larger omissions. Variable deductions for confused reasoning.*

Subtyping

Throughout this section, we consider the simply typed lambda-calculus with subtyping, product types, and records. All of the appropriate definitions are at the end of the exam.

13. (6 points) In this problem, we examine possible variations of the simply-typed lambda calculus with subtyping.

    (a) Suppose we remove the `S_Arrow` rule from the subtyping relation. Do progress and preservation continue to hold after this change, or does one (or do both) fail? If either fails, give a counterexample.

    Answer: Neither breaks. Intuitively, we added subtyping to a language that was already sound in order to allow more terms to have types. Reducing some of that freedom maintains soundness.

    Grading scheme: One point if only one is identified as still holding; 0 if both are given as failing.

    (b) Suppose we change the `S_Arrow` rule to:

    ```
    T1 <: S1      T2 <: S2
    ----------------------                    (S_Arrow_Odd)
     S1 → S2 <: T1 → T2
    ```

    Do progress and preservation continue to hold after this change, or does one (or do both) fail? If either fails, give a counterexample.

    Answer: Preservation breaks. Consider `empty |- ((\x:{}. {}) {}).i : A→A`*. We can show* `{i:A→A} <: {}`*, so* `{} → {} <: {} → {i:A→A}`*, and so the inner application has type* `{i:A→A}`*. But when we take a step, we must show that* `empty |- {}.i : A→A`*, but we cannot — `{}` can only be typed as `{}` or `Top`.*

    Grading scheme: 1 point for identifying preservation as failing, another for a counterexample, and 1 for saying progress holds.

14. (6 points) The subtyping rule for products in the final homework assignment

    ```
    S1 <: T1     S2 <: T2
    ---------------------                     (S_Prod)
      S1 * S2 <: T1 * T2
    ```

    intuitively corresponds to the "depth" subtyping rule for records. Extending the analogy, we might consider adding a "width" rule

    ```
    -------------                             (S_ProdW)
    S1 * S2 <: S1
    ```

    for products.

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

    Answer: No, since it will break progress: `({i=\y:A.y},{}).i` *cannot take a step even though it is well typed using this rule.*

    Grading scheme: 1 point for "bad idea", 1 point for noting preservation holds, 2 points for noting progress fails, and 2 points for a counterexample. Due to its open-ended nature, some answeres to this question were given points despite not discussing these topics, e.g., if they offered a set of changes that would make a system with this rule sound.

15. (14 points) Write a careful informal proof of the following theorem.

Theorem: If `S1*S2 <: T`, then either `T = Top` or else `T = T1*T2`, with `S1 <: T1` and `S2 <: T2`.

You may use the following result in your proof (you do not need to prove it):

Lemma [sub-inversion-Top]: For any type `S`, if `Top <: S` then `S = Top`.

Answer:

Proof: *By induction on the given derivation.*

- *If the last rule is* `S_Refl`*, then* `T = S1*S2` *and the result follows by two uses of* `S_Refl`*.*
- *If the last rule is* `S_Trans`*, then* `S <: U` *and* `U <: T` *for some* `U`*. By the IH, either* `U = Top` *or else* `U = U1*U2` *with* `S1 <: U1` *and* `S2 <: U2`*. In the first case, a straightforward inner induction on the derivation of* `Top <: T` *shows that* `T = Top`*. In the second case, applying the IH to* `U1*U2 <: T` *tells us that either* `T = Top` *or else* `T = T1*T2` *with* `U1 <: T1` *and* `U2 <: T2`*. If* `T = Top`*, we are finished. If* `T = T1*T2`*, then by* `S_Trans` *we have* `S1 <: T1` *and* `S2 <: T2`*, as required.*
- *If the last rule is* `S_Top`*, then* `T = Top` *immediately.*
- *If the last rule is* `S_Pair`*, then* `T = T1*T2` *with* `S1 <: T1` *and* `S2 <: T2`*, and the result is immediate.*
- *None of the other rules (*`S_Arrow`*,* `S_RcdWidth`*,* `S_RcdDepth`*, or* `S_RcdPerm`*) are possible.* □

Grading scheme: 1 point for induction on the correct derivation, 1 point per case (refl, trans, pair, and top), 1 point for mentioning that the other cases do not adhere, 1 point each for the logic of the easy cases (refl, top, and pair), and 5 points for the `S_Trans` *case: 1 for the first application of the IH, 1 for the* `U = Top` *case, 1 for the second IH application, 1 for the* `T = Top` *case, and 1 for the* `T = T1*T2` *case.*