CIS 500: Software Foundations **Final Exam**

December 16-18, 2020

Solutions

1 (8 points)

Regular Expressions

Recall the definitions of regular expressions and regular expression matching in Coq. (For reference, we have included the relevant definitions on page 2 in the appendix.)

One interesting property of a regular expression is its *cardinality*—i.e., the number of strings that it matches. For example, here are some regular expressions along with their cardinalities:

```
EmptySet                          0
EmptyStr                          1
Char "x"                          1
Union (Char "x") (Char "y")       2
Star (Char "x")                   infinite
```

Choose the correct cardinality for each of the following regular expressions:

1.1
```
App (Union (Char "x") (Char "y"))
    (Union (Char "z") (Char "w"))
```

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☒ 4 ☐ infinite

1.2
```
App (Char "x") (Char "x")
```

☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ infinite

1.3
```
Union (Char "x") (Char "x")
```

☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ infinite

1.4
```
Star EmptyStr
```

☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ infinite

1.5
```
Star EmptySet
```

☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ infinite

1.6
```
Star (Union (Char "x") EmptySet)
```

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☒ infinite

1.7
```
Star (Star EmptyStr)
```

☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ infinite

1.8
```
Star (Star EmptySet)
```

☐ 0 ☒ 1 ☐ 2 ☐ 3 ☐ 4 ☐ infinite

2 (14 points)

Inductively Defined Relations

The following inductively defined predicate classifies regular expressions whose cardinality is nonzero (i.e., those that match at least one string).

```
Inductive nonempty {T : Type}: reg_exp T -> Prop :=
  | ne_char (c : T) :
      nonempty (Char c)
  | ne_app (e1 e2 : reg_exp T) :
      nonempty e1 -> nonempty e2 -> nonempty (App e1 e2)
  | ne_union_l (e1 e2 : reg_exp T) :
      nonempty e1 -> nonempty (Union e1 e2)
  | ne_union_r (e1 e2 : reg_exp T) :
      nonempty e2 -> nonempty (Union e1 e2)
  | ne_star (e : reg_exp T) :
      nonempty (Star e)
  | ne_emptystr : nonempty EmptyStr
.
```

For example:

```
nonempty (Char "x")
nonempty (App (Char "x") (Char "x"))
~ nonempty (App (Char "x") EmptySet)
```

2.1 The inductively defined predicate `contains_char` classifies regular expressions that contain some strings with at least one character in them. That is, expressions that are empty or only contain the empty string do *not* satisfy this predicate. For example:

```
contains_char (Char "x")
contains_char (App (Char "x") (Char "x") )
~ contains_char EmptyStr.
~ contains_char EmptySet
~ contains_char (App (Char "x") EmptySet)
```

Complete the definition of `contains_char`.

```
Inductive contains_char {T : Type}: reg_exp T -> Prop :=
```

Answer:

```
| cc_char (c : T) :
    contains_char (Char c)
| cc_app_l (e1 e2 : reg_exp T) :
    contains_char e1 -> nonempty e2 -> contains_char (App e1 e2)
| cc_app_r (e1 e2 : reg_exp T) :
    nonempty e1 -> contains_char e2 -> contains_char (App e1 e2)
| cc_union_l (e1 e2 : reg_exp T) :
    contains_char e1 -> contains_char (Union e1 e2)
| cc_union_r (e1 e2 : reg_exp T) :
    contains_char e2 -> contains_char (Union e1 e2)
| cc_star (e : reg_exp T) :
    contains_char e -> contains_char (Star e).
```

2.2 The inductively defined predicate `infinite` classifies regular expressions of infinite cardinality (i.e., ones that match infinitely many strings). For example:

```
infinite (Star (Char "x" ))
~ infinite (Star EmptyStr)
~ infinite (Star (App EmptySet (Char "x")) )
```

Complete the definition of `infinite`. Your solution may refer to `nonempty` and/or `contains_char`.

```
Inductive infinite {T : Type} : reg_exp T -> Prop :=
```

Answer:

```
| inf_cc_star e : contains_char e -> infinite (Star e)
| inf_app_l e1 e2 : infinite e1 -> nonempty e2 -> infinite (App e1 e2)
| inf_app_r e1 e2 : nonempty e1 -> infinite e2 -> infinite (App e1 e2)
| inf_union_l e1 e2 : infinite e1 -> infinite (Union e1 e2)
| inf_union_r e1 e2 : infinite e2 -> infinite (Union e1 e2).
```

3 (10 points)

Hoare Logic

Each of the following Hoare triples contain a variable `c`, representing an arbitrary Imp command. For each triple, check the appropriate box to indicate whether the triple is *always valid* (valid for all choices of `c`), *sometimes valid* (valid for some, but not all, choices of `c`), or *never valid* (invalid for all choices of `c`). If you choose sometimes valid, give an example of a command `c1` that makes the triple valid and a command `c2` that makes the triple invalid.

3.1
```
{{ X < Y }}
  X := Y;
  c
{{ X = Y }}
```

☐ Always valid ☒ Sometimes valid ☐ Never valid

```
c1 = skip
c2 = X := 40; Y := 2
```

3.2
```
{{ X = 2 }}
  while (0 < X) do
    c;
    X := X + 1
  end
{{ X = 2 }}
```

☒ Always valid ☐ Sometimes valid ☐ Never valid

```
c1 =
c2 =
```

3.3
```
{{ True }}
  if (X < 10)
    then while (X < 10) do c end
    else c
  end
{{ X > 10 }}
```

☐ Always valid ☒ Sometimes valid ☐ Never valid

```
c1 = X := 20
c2 = X := 3
```

3.4
```
{{ X = Y + 2 }}
  if (Y < X)
    then while (0 < Y) do c end
    else c
  end
{{ Y > X }}
```

☐ Always valid ☒ Sometimes valid ☐ Never valid

```
c1 =
c2 =
```

3.5
```
{{ X = Y }}
  Y := X + Y;
  while (X < Y) do c end
{{ X = Y }}
```

☐ Always valid ☒ Sometimes valid ☐ Never valid

```
c1 = X := Y
c2 = X := Y + 20
```

4 (14 points)

STLC with Nondeterminism

We've seen how nondterminism in the form of a `havoc` command can be added to the Imp language. In this problem, we'll consider adding a similar feature to the STLC.

First, we add a new term, `havoc`, to the syntax of terms. This term has type `Nat` and can step to any constant natural number. Here is the fragment of the `step` relation that pertains to `havoc`.

```
Inductive step : tm -> tm -> Prop :=
  | ST_Havoc : forall (n : nat),
      <{ havoc }> --> <{ n }>
```

This version of the simply typed lambda calculus with `havoc` also has `let` bindings, the `fix` combinator, and basic arithmetic. These features all have the same semantics as they were given in lecture and homework assignments. The full definition can be found on page 3 of the appendix, for reference.

Since a single term can now step (and hence also multistep) to many different values, it is interesting to compare terms according to the sets of final values they can reduce to. We say that a term `t1` *refines* a term `t2` if the set of values that `t1` reduces to is a subset of the set of values that `t2` reduces to.

For example, the term

```
1*1
```

refines the term

```
havoc+1
```

because the first reduces (only) to the numeric value `1`, while the second term can reduce to any numeric value except `0`. Conversely, a counterexample for the claim "term `t1` refines the term `t2`" is a value `v` that `t1` reduces to but that `t2` cannot reduce to.

Here are several pairs of terms in the simply typed lambda calculus extended with `havoc`. For each pair, first answer whether the one on the left refines the one on the right, then answer whether the one on the right refines the one on the left.

4.1

```
havoc
```

```
2 * havoc
```

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: N.

```
1 (or any odd number)
```

4.2 Does the expression on the right refine the expression on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.3

```
havoc
```

```
havoc * havoc
```

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.4 Does the expression on the right refine the expression on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.5

```
let x := havoc in x * x
```

```
havoc * havoc
```

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.6 Does the expression on the right refine the expression on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: N.

```
2 (or any non square)
```

4.7

```
(\x:Nat,
     if x = havoc then 1 else 2)
   havoc
```

```
(\x:Nat,
    if x = havoc then 1 else 2)
  1
```

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.8 Does the expression on the right refine the expression on the left?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.9

```
fix (\f:Nat->Nat,
     \x:Nat, if x = 0
               then x
               else f havoc)
  1
```

```
0
```

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.10 Does the expression on the right refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.11

```
havoc
```

```
fix (\f:Nat -> Nat,
     \x:Nat, if x = havoc
               then x
               else f havoc)
  1
```

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.12 Does the expression on the right refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

4.13

```
fix (\f:Nat->Nat,
     \x:Nat, let y := havoc in
             if y = 0
               then y
               else f x) 0
```

```
fix (\f:Nat -> Nat,
     \x:Nat, f x)
 0
```

Does the expression on the left refine the expression on the right?
If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: N.

```
0
```

4.14 Does the expression on the right refine the expression on the right? If not, show a counterexample.

```
[] Yes
[] No (Give final value below)
      v =
```

Solution: Y.

5 (12 points)

Contextual Equivalence in the STLC

We saw in the Equiv chapter how to define and reason about "behavioral equivalence" of programs in Imp. Intuitively, two Imp programs are equivalent if they embody the same partial function from input states to output states. In the STLC, there are no "states," so, to define equivalence, we can just compare outputs. For example, the terms `(\x:Bool.x) true` and `(\x:Bool.true) false` are equivalent because they both reduce to the same value, `true`.

But what should we say about equivalence of terms with function types? For example, `(\x:Bool. x)` and `(\x:Bool. if x then true else false)` are "obviously" equivalent, even though they do *not* reduce to the same thing (both are already values). One popular answer to this puzzle is the notion of *contextual equivalence*—the idea that two functions should be considered equal if they behave the same when placed in any larger context.

Formally, we define a *term context* `P` to be an incomplete term containing one or more *holes*, written `[]`, that are waiting to be filled with a term.

```
Inductive tctx : Type :=
  | P_hole
  | P_var : string -> tctx
  | P_true : tctx
  | P_false : tctx
  | P_app : tctx -> tctx -> tctx
  | P_abs : string -> ty -> tctx -> tctx
  | P_if : tctx -> tctx -> tctx -> tctx.
```

We'll write term contexts using the same concrete notations as for ordinary terms, except that, in formal Coq definitions, we'll enclose them in `[{...}]` brackets to tell the parser that they may contain holes.

We next define a function, `fill`, that "plugs in" an arbitrary term into the holes in a term context.

```
Fixpoint fill (t:tm) (P:tctx) : tm :=
  match P with
  | P_var x => tm_var x
  | [{ [] }] => t
  | [{ true }] => <{ true }>
  | [{ false }] => <{ false }>
  | [{ P1 P2 }] => let t1' := fill t P1 in
                   let t2' := fill t P2 in
                  <{ t1' t2' }>
  | [{ \x:T, P }] => let t' := fill t P in
                  <{ \x : T, t' }>
  | [{ if P1 then P2 else P3 }] =>
                  let t1' := fill t P1 in
                  let t2' := fill t P2 in
                  let t3' := fill t P3 in
                  <{ if t1' then t2' else t3' }>
  end.
```

Notice that the term `t` being plugged into the hole is not required to be closed: it can have free variables, which can then be *captured* by variable binders above the hole in the term context `P`. That is, hole filling can be thought of as a sort of "non-capture-avoiding substitution." For instance,

```
fill <{ x (\y:Bool,y) }> [{ \x:(Bool->Bool)->Bool, [] }]
```

yields the term:

```
\x:(Bool->Bool)->Bool, x (\y:Bool,y)
```

Finally, we say that terms `x` and `y` are *contextually equivalent* if they can be plugged into *any* term context `P` and produce the same results. Formally:

```
Definition contextually_equivalent (t1 t2 : tm) : Prop :=
  forall (P : tctx) (v : tm),
  value v ->
  ((fill t1 P) -->* v <-> (fill t2 P) -->* v).
```

Note that this definition does not require that the terms being compared be either well typed (we are ignoring typing completely) or closed. In particular, understanding whether two terms are contextually equivalent requires considering how free variables in each might be bound when they are plugged into a term context.

For instance, these two terms are contextually equivalent

```
if true then x else y
```

```
if false then y else x
```

because, whenever they are plugged into a term context that binds variables `x` and `y`, both of them will reduce to the same value. For example, plugging the first into the term context

```
(\x:Bool, \y:Bool, []) true false
```

yields the term

```
(\x:Bool, \y:Bool, if true then x else y) true false
```

which reduces to `true`, while plugging the second term into the same context yields

```
(\x:Bool, \y:Bool, if false then y else x) true false
```

which again reduces to `true`.

By contrast, these terms are *not* contextually equivalent

```
\y:Bool, y
```

```
\y:Bool, x
```

because we can exhibit a *distinguishing context* for them. A distinguishing context for terms `t1` and `t2` is a term context `P` such that `fill t1 P -->* v1` and `fill t2 P -->* v2`, where `v1` and `v2` are distinct boolean values. Here is a distinguishing context `P` for these terms.

```
(\x:Bool, [] false) true
```

Filling the hole in `P` with the first term yields

```
(\x:Bool, (\y:Bool,y) false) true
```

which reduces to `false`. On the other hand, filling `P` with the second term yields

```
(\x:Bool, (\y:Bool,x) false) true
```

which reduces to `true`.

For each of the following, determine if the two given terms are contextually equivalent (CE). If they are not, give an example of a distinguishing context `P`.

5.1

```
if x then true else false
```

```
if (if x then false else true)
  then false
  else true
```

```
[] CE
[] Not CE (give a distinguishing context P below)
      P =
```

Solution: CE

5.2

```
(\y:Bool, (\x:Bool, y))
```

```
(\y:Bool, (\x:Bool, x))
```

```
[] CE
[] Not CE (give a distinguishing context P below)
      P =
```

Solution: Not CE.

```
[] true false
```

5.3

```
if x then true else false
```

```
if x then true else true
```

```
[] CE
[] Not CE (give a distinguishing context P below)
      P =
```

Solution: Not CE

```
P = (\x:Bool, []) false
```

5.4

```
(\f:Bool->Bool,
    (\a:Bool, f a)) x y
```

```
x y
```

```
[] CE
[] Not CE (give a distinguishing context P below)
      P =
```

Solution: CE.

5.5

```
if false then x else true
```

```
if true then x else false
```

```
[] CE
[] Not CE (give a distinguishing context P below)
      P =
```

Solution: not CE

```
P = (\x:Bool, []) false
```

5.6

```
    (\y:Bool, y)
```

```
  (if x then (\y:Bool, x)
      else (\y:Bool, y))
```

```
[] CE
[] Not CE (give a distinguishing context P below)
      P =
```

Solution: Not CE

```
(\x:Bool, [] false) true
```

6 (12 points)

STLC with I/O

Suppose we want to add input and output to the simply typed lambda calculus (with base type `Nat`). A straightforward way to do this is to add three new forms of term:

- `print n`, which adds `n` to a list of outputs from the program;
- `read`, which returns the next number from a list that is provided to the program when it begins executing; and
- `t1; t2`, which evaluates `t1`, ignores its result, and returns the result of evaluating `t2`, thus forcing all of the side effects (reads and writes) of `t1` to occur before those of `t2`.

Here are a few examples showing how these new forms of terms reduce. Note that the input list is the first list element of the tuple, and the output list is the second.

```
      (<{ (\x : Nat, \y : Nat, x) read read }>, [5;6;7], [1;2])
-->* (<{ 5 }>, [7], [1;2])
```

(That is, if the remaining input list contains `5`, `6`, and `7` and if `2` and `1` have been output, then this expression reads `5` and `6`, yields `5` as its result, and leaves the input list two elements shorter and the output list unchanged.)

```
      (<{ print 1; print 2; print 3 }>, [], [])
-->* (<{ 3 }> [], [3;2;1])
```

(That is, printing `1`, then `2`, then `3`, starting from empty input and output lists, yields the result `3` and the output list `[3;2;1]`, in that order.)

```
      (<{ print read }>, [], [] )
-->* (<{ 0 }> [], [0])
```

(Reading from an empty input list yields `0`.)

Here is the definition of the `has_type` relation for this language.

```
Inductive has_type : context -> tm -> ty -> Prop :=
  (* pure STLC *)
  | T_Var : forall Gamma x T1,
      Gamma x = Some T1 ->
      Gamma |- x \in T1
  | T_Abs : forall Gamma x T1 T2 t1,
    (x |-> T2 ; Gamma) |- t1 \in T1 ->
      Gamma |- \x:T2, t1 \in (T2 -> T1)
  | T_App : forall T1 T2 Gamma t1 t2,
      Gamma |- t1 \in (T2 -> T1) ->
      Gamma |- t2 \in T2 ->
      Gamma |- t1 t2 \in T1
  (* numbers *)
  | T_Nat : forall Gamma (n : nat),
      Gamma |- n \in Nat
  (* I/O *)
  | T_Read : forall Gamma,
      Gamma |- read \in Nat
  | T_Print : forall Gamma t,
      Gamma |- t \in Nat ->
      Gamma |- print t \in Nat
  | T_Seq : forall Gamma t1 t2 T1 T2,
      Gamma |- t1 \in  T1->
      Gamma |- t2 \in T2 ->
      Gamma |- t1 ; t2 \in T2

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

And here are the (unsurprising) definitions of values and substitution.

```
Inductive value : tm -> Prop :=
  | value_const : forall (n : nat), value <{ n }>
  | value_abs : forall x T e, value <{ \x : T, e }>.

Fixpoint subst (x : string) (s : tm) (t : tm) :=
  match t with
  | tm_var y => if (eqb_string x y) then s else t
  | tm_const n => tm_const n
  | <{ t1 t2}> => <{ ({subst x s t1} ) ({subst x s t2})  }>
  | <{ \y: T, t1 }> => <{ \y : T, ({ if eqb_string x y then t1 else subst x s t1})  }>
  | <{ read }> => <{ read }>
  | <{ print t1 }> => <{ print ({subst x s t1}) }>
  | <{ t1 ; t2 }> =>  <{ ({subst x s t1} ); ({subst x s t2})  }>
   end
where "'[' x ':=' s ']' t" := (subst x s t) (in custom stlc).
```

Your task is to complete the definition of the `step` relation below. A correct definition will satisfy the following `progress` and `preservation` theorems.

```
Theorem progress : forall (t : tm)
                          (input output : list nat)
                          (T : ty),
    empty |- t \in T ->
    (value t \/ exists t' input' output',
                  (t, input, output) --> (t', input', output')).

Theorem preservation : forall (t t' : tm)
                              (input output input' output' : list nat)
                              (T : ty),
    empty |- t \in T ->
    (t,input,output) --> (t',input',output') ->
    empty |- t' \in T.
```

Pay close attention to the provided header.

```
Inductive step : (tm * list nat * list nat) -> (tm * list nat * list nat) -> Prop :=
```

Answer:

```
  | ST_Print_const : forall output input (n : nat),
      ( <{ print n }>, input, output) --> ( tm_const n, input, (n :: output))
  | ST_value_seq : forall input output t2 v1,
      value v1 ->
      (<{ v1; t2 }>, input, output) --> (t2, input, output)
  | ST_seq1 : forall input input' output output' t1 t1' t2,
      (t1, input, output) --> (t1', input', output') ->
      (<{ t1 ; t2 }>, input, output ) --> (<{ t1'; t2 }>, input', output')
  | ST_Read : forall input output (n : nat),
      (<{ read }>, (n :: input), output ) --> ( tm_const n, input, output )
  | ST_Abs : forall input input' output output' t1 t1' t2,
      (t1, input, output) --> (t1', input', output') ->
      (<{ t1 t2 }>, input, output ) --> (<{ t1' t2 }>, input', output' )
  | ST_subst : forall input output x t1 T t2,
      value t1 ->
      (<{ (\x : T, t1) t2 }>, input, output) --> (<{ [x := t2] t1 }>, input, output)

where "t '-->' t'" := (step t t').
```

7 (6 points)

Subtyping

For each of the following pairs of types, `S` and `T`, select the appropriate description of how they are ordered in the subtype relation.

7.1
```
S = (Top -> Top) -> Top
T = Top -> Top -> Top
```
☐ S:<T ☒ T<:S ☐ equivalent ☐ unrelated

7.2
```
S = (Top * Top) -> Top
T = Top -> (Top -> Top)
```
☐ S:<T ☒ T<:S ☐ equivalent ☐ unrelated

7.3
```
S = {x : Bool, y : Top, z : Bool} -> Top
T = {x : Bool, y : Bool} -> Top
```
☐ S:<T ☐ T<:S ☐ equivalent ☒ unrelated

7.4
```
S = Bool * Bool -> Bool
T = Bool -> Bool -> Bool
```
☐ S:<T ☐ T<:S ☐ equivalent ☒ unrelated

7.5
```
S = (Top -> Bool) -> Top
T = (Bool -> Top) -> Bool
```
☐ S:<T ☒ T<:S ☐ equivalent ☐ unrelated

7.6
```
S = {x : Bool, y : Nat} -> {y : Nat, x : Bool}
T = {y : Nat, x : Bool} -> {x : Bool, y : Nat}
```
☐ S:<T ☐ T<:S ☒ equivalent ☐ unrelated

8 (10 points)

Designing a Typed Stack Machine

In this problem your task will be to design a typing relation for the world's simplest stack machine.

The machine itself consists of a straight-line program—a list of instructions—plus a stack that can hold both numeric and boolean values.

```
Inductive stack_element :=
| elt_bool : bool -> stack_element
| elt_nat  : nat -> stack_element.

Definition stack := list stack_element.

Inductive instr :=
| instr_push : nat -> instr
| instr_add : instr
| instr_and : instr
| instr_eq : instr.

Definition prog := list instr.
```

A single step of the machine executes a single instruction, taking a starting stack to an ending stack.

```
Reserved Notation "i '/' s '-->' s'" (at level 40).

Inductive step : instr -> stack -> stack -> Prop :=
| ST_push : forall s n,
    instr_push n / s --> (elt_nat n :: s)
| ST_add : forall s n1 n2,
    instr_add / (elt_nat n1 :: elt_nat n2 :: s) --> (elt_nat (n1+n2) :: s)
| ST_and : forall s b1 b2,
    instr_and / (elt_bool b1 :: elt_bool b2 :: s) --> (elt_bool (b1&&b2) :: s)
| ST_eq : forall s n1 n2,
    instr_eq / (elt_nat n1 :: elt_nat n2 :: s) --> (elt_bool (beq_nat n1 n2) :: s)

where "i / s '-->' s'" := (step i s s').
```

This is lifted to a multi-step reduction relation that executes one or more instructions and threads the ending stack from each into the starting stack for the next.

```
Reserved Notation "p '/' s -->* s'" (at level 40).

Inductive multistep : prog -> stack -> stack -> Prop :=
  | multi_refl : forall (s : stack), [] / s -->* s
  | multi_step : forall i p (s s_mid s' : stack),
      i / s --> s_mid ->
      p / s_mid -->* s' ->
      (i::p) / s -->* s'

where "p '/' s -->* s'" := (multistep p s s').
```

Notice that this machine can get stuck in various situations, when there are not enough operands on the stack, or the top stack elements do not have the right types. E.g.

```
Example stepeg0 :
  ~ exists s',
      instr_add / [elt_nat 4] --> s'.

Example stepeg1 :
  ~ exists s',
      instr_add / [elt_nat 4; elt_bool false] --> s'.
```

To typecheck programs for this machine, we first begin by assigning types to individual stack elements and to whole stacks.

```
Inductive ty :=
| ty_Bool
| ty_Nat.

Definition stack_ty := list ty.

Inductive elt_has_type : stack_element -> ty -> Prop :=
| ET_nat : forall n, elt_has_type (elt_nat n) ty_Nat
| ET_bool : forall b, elt_has_type (elt_bool b) ty_Bool.

Reserved Notation "'|-' s '\in*' sty" (at level 40).

Inductive stack_has_type : stack -> stack_ty -> Prop :=
| SHT_nil : |- [] \in* []
| SHT_cons : forall s sty e T,
    |- s \in* sty ->
    elt_has_type e T ->
    |- (e::s) \in* (T::sty)

where "'|-' s '\in*' sty" := (stack_has_type s sty).
```

Your task is to fill in the details of the following definitions.

8.1 First, complete the following definition of the typing relation for individual instructions. Notice that it relates an instruction and *two* types, one describing the stack before the instruction executes and one for the state after. For example:

```
Example eg0 :
  |- instr_push 4 \in [ty_Nat] --> [ty_Nat; ty_Nat].

Reserved Notation "'|-' i '\in' st --> st'" (at level 40).

Inductive instr_has_type : instr -> stack_ty -> stack_ty -> Prop :=
```

Answer:

```
| T_push : forall sty n,
    |- instr_push n \in sty --> (ty_Nat :: sty)
```

```
| T_add : forall sty,
    |- instr_add \in (ty_Nat :: ty_Nat :: sty) --> (ty_Nat :: sty)
| T_and : forall sty,
    |- instr_and \in (ty_Bool :: ty_Bool :: sty) --> (ty_Bool :: sty)
| T_eq: forall sty,
    |- instr_eq \in (ty_Nat :: ty_Nat :: sty) --> (ty_Bool :: sty)

  where "|- i \in sty '-->' sty'" := (instr_has_type i sty sty').
```

8.2 Next, use the instruction typing relation to define a similar relation describing how executing a whole program changes the shape of the stack.

```
Example eg1 :
  |- [instr_push 4; instr_push 6]
     \in [] -->* [ty_Nat; ty_Nat].

Reserved Notation "'|-' p '\in' st '-->*' st'" (at level 40).

Inductive prog_has_type : prog -> stack_ty -> stack_ty -> Prop :=
```

Answer:

```
| prog_nil : forall sty, |- [] \in sty -->* sty
| prog_cons : forall i p sty sty_mid sty',
    |- i \in sty --> sty_mid ->
    |- p \in sty_mid -->* sty' ->
    |- (i :: p) \in sty -->* sty'
```

9 [Advanced Track Only] (16 points)

Progress and Preservation for the Typed Stack Machine

Next (for those on the advanced track or taking the exam as WPE-I), we will prove correctness of the stack machine typing relation from your solution to the previous problem.

Feel free to state (and prove!) auxiliary lemmas if you need them.

Write your proofs in good, clear English, stating any induction hypotheses *explicitly*.

Since the `and` and `eq` instructions are very similar to `add`, you can elide the cases for these instructions and just give the cases for `push` and `add`.

9.1 First, show that well-typed machine states (program plus stack) are not stuck.

```
Theorem progress : forall i s sty sty',
    |- i \in sty --> sty' ->
    |- s \in* sty ->
    exists s', i / s --> s'.
Proof.
```

Answer:

```
We are given:
    HI: |- i \in sty --> sty'
    HS: |- s \in* sty
Proceed by cases on the rule used to prove HI.
- Case T_push:
    i = instr_push n
  Then
    i / s --> (elt_nat n)::s
  by rule ST_push.
- Case T_add:
    i = instr_add
    sty = ty_Nat :: ty_Nat :: sty0
  By the form of the stack typing relation, from HS we
  immediately have
    s = elt_nat n :: elt_nat n0 :: s0
  for some n, n0, and s0.
  So rule ST_add applies, yielding
    i / s --> (elt_nat (n+n0))::s0.
- The T_and and T_eq cases follow exactly the same pattern.
```

9.2 Next, prove that stepping a single instruction "preserves typing," in the sense that the type you assign to the instruction correctly describes the way the instruction transforms the shape of the stack.

```
Lemma step_preserves_typing : forall i s sty s' sty',
    |- i \in sty --> sty' ->
    |- s \in* sty ->
    i / s --> s' ->
    |- s' \in* sty'.
Proof.
```

Answer:

```
We are given:
    HI:   |- i \in sty --> sty'
    HS:   |- s \in* sty
    HST:  i / s --> s'
Proceed by cases on the rule used to prove HI.

- Case T_push:
    i = instr_push n
    sty' = ty_Nat :: sty
  Then the rule used to prove HST must be ST_push, with
    s' = (elt_nat n)::s
  Since (elt_nat n) has type ty_Nat, the required conclusion
    |- s \in* sty
  follows immediately by rule SHT_cons, using HS.

- Case T_add:
    i = instr_add
    sty  = ty_Nat :: ty_Nat :: sty0
    sty' = ty_Nat :: sty0
  The rule used to prove HST must be ST_add, with
    s = (elt_nat n1)::(elt_nat n2)::s0
    s' = (elt_nat (n1+n2))::s
  Since (elt_nat (n1+n2) has type ty_Nat, the result follows by SHT_cons.

- The T_and and T_eq cases follow the same pattern.
```

9.3 Finally, show that multistep reduction preserves typing in the same sense.

```
Theorem multistep_preserves_typing : forall p sty sty',
    |- p \in sty -->* sty' ->
    forall s s',
    |- s \in* sty ->
    p / s -->* s' ->
    |- s' \in* sty'.
Proof.
```

Answer:

```
We prove, by induction on a derivation of
    |- p \in sty -->* sty'
that
    forall s s',
    |- s \in* sty ->
    p / s -->* s' ->
    |- s' \in* sty'.
- Case prog_nil:
    p = []
    sty' = sty
  Suppose we are given s and s' with:
    HS:  |- s \in* sty
```

```
    HP:  p / s -->* s'
  By the form of p, the last rule used in HP must be multi_refl,
  with s' = s.  The result is then immediate from HS.
- Case prog_cons:
    p = i :: p0
    IH:  forall s_mid s' : stack,
           |- s_mid \in* sty_mid ->
           p0 / s_mid -->* s' ->
           |- s' \in* sty'
  Suppose we are given s and s' with:
    HS:  |- s \in* sty
    HP:  p / s -->* s'
  From
 the form of p, the last rule used in HP must be multi_step,
  with
      HI:  i / s --> s_mid
      HP0: p0 / s_mid -->* s'
  for some s_mid.
  From HI and HS, Lemma step_preserves_typing yields:
     |- s_mid \in* sty_mid
  From this, HP0, and the IH, we obtain
    IH0: |- s' \in* sty'
  as required.
```

10 **[Advanced Track Only]** (15 points)

Correctness of factorial in STLC

For this problem, we will be working in a variant of the simply typed lambda-calculus enriched with numbers and fixed points, summarized on page 6 in the appendix.

Recall the definition of the factorial function as a term in this system:

```
Definition stlcfact :=
  <{fix
      (\f:Nat->Nat,
        \a:Nat,
         if0 a then 1 else (a * (f (pred a))))}>.
```

To simplify the reasoning in this problem, we make one technical change to the way the system is defined, replacing the two reduction rules for the "fix" construct with the following "macro rule"

```
    | ST_FixAbsApp : forall f x T1 T1' t1 v2,
        value v2 ->
        <{ fix (\f:T1->T1', \x:T1, t1) v2 }> -->
        <{ [x := v2] ([f := fix (\f:T1->T1', \x:T1, t1)] t1)}>
```

and adding a clause to the definition of values

```
    | v_fix : forall v1,
        value v1 ->
        value <{fix v1}>.
```

so that a term like

```
fact (2+1)
```

will reduce to

```
fact 3
```

before any reduction steps involving the body of `stlcfact` take place.

The `ST_FixAbsApp` rule is not quite as powerful as the two rules it replaces

```
    | ST_Fix1 : forall t1 t1',
         t1 --> t1' ->
         <{ fix t1 }> --> <{ fix t1' }>
    | ST_FixAbs : forall x T1 t1,
        <{ fix (\ x : T1, t1) }> -->
        <{ [x := fix \x : T1, t1 ] t1 }>
```

(it does not support mutually recursive functions, for example), but it is a little simpler to reason about because it takes larger steps.

Now, it is intuitively clear that the term `stlcfact` represents the "real" factorial function that we defined earlier in the semester in Gallina:

```
Fixpoint realfact (x:nat) :=
  match x with
    0 => 1
  | S x' => x * (realfact x')
  end.
```

To state this claim rigorously, we can say that an STLC term `tf` of type `Nat->Nat` *represents* a Coq function `f` of type `nat->nat` if applying them to the same input yields the same result.

```
Definition represents (tf: tm) (f: nat->nat) : Prop :=
  forall n,
    (tm_app tf (tm_const n)) -->* tm_const (f n).
```

In particular:

```
Theorem fact_correct: represents stlcfact realfact.
```

Write a careful *informal* proof of this theorem in English. If your proof uses induction, make sure to state the induction hypothesis explicitly. Feel free to state (and prove!) auxiliary lemmas if this is useful.

Since this is an informal proof, the `<{...}>` braces around STLC terms are not needed here (they are used in the textbook just to avoid confusing the Coq parser).

Proof:

```
We first prove a congruence lemma for multiplication:

Lemma multistep_congr_mult : forall t1 t2 t2',
     t2 -->* t2' ->
     value t1 ->
     <{t1 * t2}> -->* <{t1 * t2'}>.

Proof: Straightforward induction on a derivation of
t2 -->* t2'.  If the final rule in this derivation is multi_refl,
then t1 * t2 -->* t1 * t2' also follows by multi_refl.  If the
final rule is multi_step, we have t2 --> t2a and t2a -->* t2 and
(the IH) t1 * t2a -->* t1 * t2.  By ST_Mult2, t1 * t2 --> t1 * t2a; from
this and the IH, the desired result follows by multi_step.

Main theorem: stlcfact represents realfact.

Proof: Unfolding the definition of "represents", we get

     forall n, <{ stlcfact n }> -->* realfact n

Proceed by induction on n.

Case n=0.
  In this case, <{ stlcfact n }> steps to <{0}> and
  realfact n = 0, as required.
```

```
Case n = S n', with
     IH: <{ stlcfact n' }> -->* realfact n'.

  In this case, <{ stlcfact (S n) }> reduces in a few steps to

    <{ (S n') * (stlcfact n') }>

  By the congruence lemma for times and the IH,

    <{ (S n') * (stlcfact n') }> -->* <{ (S n') * (realfact n') }>

  Also, by ST_Mulconsts

      <{ (S n') * (realfact n') }> --> (S n') * (realfact n')

  hence

      <{ (S n') * (realfact n') }> -->* (S n') * (realfact n')

  By the transitivity of -->*  (twice),

      <{ stlcfact (S n') }> -->* (S n') * (realfact n')

  as required.
```

For Reference

Regular Expressions

```
Inductive reg_exp (T : Type) : Type :=
  | EmptySet
  | EmptyStr
  | Char (t : T)
  | App (r1 r2 : reg_exp T)
  | Union (r1 r2 : reg_exp T)
  | Star (r : reg_exp T).

Reserved Notation "s =~ re" (at level 80).

Inductive exp_match {T} : list T -> reg_exp T -> Prop :=
  | MEmpty : [] =~ EmptyStr
  | MChar x : [x] =~ (Char x)
  | MApp s1 re1 s2 re2
             (H1 : s1 =~ re1)
             (H2 : s2 =~ re2)
           : (s1 ++ s2) =~ (App re1 re2)
  | MUnionL s1 re1 re2
               (H1 : s1 =~ re1)
             : s1 =~ (Union re1 re2)
  | MUnionR re1 s2 re2
               (H2 : s2 =~ re2)
             : s2 =~ (Union re1 re2)
  | MStar0 re : [] =~ (Star re)
  | MStarApp s1 s2 re
                (H1 : s1 =~ re)
                (H2 : s2 =~ (Star re))
              : (s1 ++ s2) =~ (Star re)
  where "s =~ re" := (exp_match s re).
```

STLC with let, booleans, `fix`, and `havoc`

```
Inductive ty : Type :=
  | Ty_Arrow : ty -> ty -> ty
  | Ty_Nat  : ty
  | Ty_Bool : ty.

Inductive tm : Type :=
  (* pure STLC *)
  | tm_var : string -> tm
  | tm_app : tm -> tm -> tm
  | tm_abs : string -> ty -> tm -> tm
  (* numbers *)
  | tm_const: nat -> tm
  | tm_succ : tm -> tm
  | tm_pred : tm -> tm
  | tm_mult : tm -> tm -> tm
  (* let *)
  | tm_let : string -> tm -> tm -> tm
  (* fix *)
  | tm_fix  : tm -> tm
  (* havoc *)
  | tm_havoc : tm
  (* bools *)
  | tm_tru : tm
  | tm_fls : tm
  | tm_eq : tm -> tm -> tm
  | tm_if : tm -> tm -> tm -> tm.

Inductive value : tm -> Prop :=
  | v_abs : forall x T2 t1,
      value <{\x:T2, t1}>
  | v_nat : forall n : nat,
      value <{n}>
  | v_tru : value <{ true }>
  | v_fls : value <{ false }>

Reserved Notation "'[' x ':=' s ']' t" (in custom stlc at level 20, x constr).

Fixpoint subst (x : string) (s : tm) (t : tm) : tm :=
  match t with
  (* pure STLC *)
  | tm_var y =>
      if eqb_string x y then s else t
  | <{\y:T, t1}> =>
      if eqb_string x y then t else <{\y:T, [x:=s] t1}>
  | <{t1 t2}> =>
      <{([x:=s] t1) ([x:=s] t2)}>
  (* numbers *)
  | tm_const _ =>
      t
  | <{succ t1}> =>
```

```
      <{succ [x := s] t1}>
  | <{pred t1}> =>
      <{pred [x := s] t1}>
  | <{t1 * t2}> =>
      <{ ([x := s] t1) * ([x := s] t2)}>
  | <{ t1 = t2 }> => <{ ( [x := s]t1 ) = ( [x := s]t2 )   }>
  (* booleans *)
  | <{if t1 then t2 else t3}> =>
      <{if [x := s] t1 then [x := s] t2 else [x := s] t3}>
  | <{ true }> => <{ true }>
  | <{ false }> => <{ false }>
  (* let *)
  | <{let y := t1 in t2}> =>
      <{let y := [x:=s] t1
        in ({ if eqb_string x y then t2 else <{ [x:=s] t2 }> }) }>
  (* fix *)
  | <{ fix t1 }> =>
      <{ fix ([x:=s] t1) }>
  (* havoc *)
  | <{ havoc }> => <{ havoc }>
end

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

Inductive step : tm -> tm -> Prop :=
  (* pure STLC *)
  | ST_AppAbs : forall x T2 t1 v2,
        value v2 ->
        <{(\x:T2, t1) v2}> --> <{ [x:=v2]t1 }>
  | ST_App1 : forall t1 t1' t2,
        t1 --> t1' ->
        <{t1 t2}> --> <{t1' t2}>
  | ST_App2 : forall v1 t2 t2',
        value v1 ->
        t2 --> t2' ->
        <{v1 t2}> --> <{v1  t2'}>
  (* numbers *)
  | ST_Succ : forall t1 t1',
        t1 --> t1' ->
        <{succ t1}> --> <{succ t1'}>
  | ST_SuccNat : forall n : nat,
        <{succ n}> --> <{ {S n} }>
  | ST_Pred : forall t1 t1',
        t1 --> t1' ->
        <{pred t1}> --> <{pred t1'}>
  | ST_PredNat : forall n:nat,
        <{pred n}> --> <{ {n - 1} }>
  | ST_Mulconsts : forall n1 n2 : nat,
        <{n1 * n2}> --> <{ {n1 * n2} }>
  | ST_Mult1 : forall t1 t1' t2,
        t1 --> t1' ->
        <{t1 * t2}> --> <{t1' * t2}>
```

```
| ST_Mult2 : forall v1 t2 t2',
       value v1 ->
       t2 --> t2' ->
       <{v1 * t2}> --> <{v1 * t2'}>
| ST_Eq1 : forall t1 t1' t2,
    t1 --> t1' ->
    <{ t1 = t2 }> --> <{ t1' = t2 }>
| ST_Eq2 : forall (n : nat) t2 t2',
    t2 --> t2' ->
    <{ n = t2 }> --> <{ n = t2' }>
| ST_Eq_true : forall (n: nat),
    <{ n = n }> --> <{ true }>
| ST_Eq_false : forall (n m : nat),
    n <> m ->
    <{ n = m }> --> <{ false }>
(* booleans *)
| ST_If : forall t1 t1' t2 t3,
       t1 --> t1' ->
       <{if t1 then t2 else t3}> --> <{if t1' then t2 else t3}>
| ST_If_true : forall t2 t3,
       <{if true then t2 else t3}> --> t2
| ST_If_false : forall t2 t3,
       <{if false then t2 else t3}> --> t3
(* let *)
| ST_Let1 : forall x t1 t1' t2,
     t1 --> t1' ->
     <{ let x := t1 in t2}> --> <{ let x := t1' in t2 }>
| ST_LetValue : forall x v1 t2,
     value v1 ->
     <{ let x := v1 in t2 }> --> <{ [x:=v1]t2 }>
(* fix *)
| ST_Fix1 : forall t1 t1',
     t1 --> t1' ->
     <{ fix t1 }> --> <{ fix t1' }>
| ST_FixAbs : forall x T1 t1,
    <{ fix (\ x : T1, t1) }> -->
    <{ [x := fix \x : T1, t1 ] t1 }>
(* havoc *)
| ST_Havoc : forall (n : nat),
    <{ havoc }> --> <{ n }>

where "t '-->' t'" := (step t t').
```

STLC with let, products, sums, and alternate rules for `fix`

```
Inductive ty : Type :=
  | Ty_Arrow : ty -> ty -> ty
  | Ty_Nat   : ty.

Inductive tm : Type :=
  (* pure STLC *)
  | tm_var : string -> tm
  | tm_app : tm -> tm -> tm
  | tm_abs : string -> ty -> tm -> tm
  (* numbers *)
  | tm_const: nat -> tm
  | tm_succ : tm -> tm
  | tm_pred : tm -> tm
  | tm_mult : tm -> tm -> tm
  | tm_if0  : tm -> tm -> tm -> tm
  (* fix *)
  | tm_fix  : tm -> tm.

Inductive value : tm -> Prop :=
  | v_abs : forall x T2 t1,
      value <{\x:T2, t1}>
  | v_nat : forall n : nat,
      value <{n}>
  (* NEW: fix applied to something is a value if its argument is *)
  | v_fix : forall v1,
      value v1 ->
      value <{fix v1}>.

Inductive step : tm -> tm -> Prop :=
  (* pure STLC *)
  | ST_AppAbs : forall x T2 t1 v2,
         value v2 ->
         <{(\x:T2, t1) v2}> --> <{ [x:=v2]t1 }>
  | ST_App1 : forall t1 t1' t2,
         t1 --> t1' ->
         <{t1 t2}> --> <{t1' t2}>
  | ST_App2 : forall v1 t2 t2',
         value v1 ->
         t2 --> t2' ->
         <{v1 t2}> --> <{v1  t2'}>
  (* numbers *)
  | ST_Succ : forall t1 t1',
         t1 --> t1' ->
         <{succ t1}> --> <{succ t1'}>
  | ST_SuccNat : forall n : nat,
         <{succ n}> --> <{ {S n} }>
  | ST_Pred : forall t1 t1',
         t1 --> t1' ->
         <{pred t1}> --> <{pred t1'}>
  | ST_PredNat : forall n:nat,
```

```
       <{pred n}> --> <{ {n - 1} }>
  | ST_Mulconsts : forall n1 n2 : nat,
       <{n1 * n2}> --> <{ {n1 * n2} }>
  | ST_Mult1 : forall t1 t1' t2,
       t1 --> t1' ->
       <{t1 * t2}> --> <{t1' * t2}>
  | ST_Mult2 : forall v1 t2 t2',
       value v1 ->
       t2 --> t2' ->
       <{v1 * t2}> --> <{v1 * t2'}>
  | ST_If0 : forall t1 t1' t2 t3,
       t1 --> t1' ->
       <{if0 t1 then t2 else t3}> --> <{if0 t1' then t2 else t3}>
  | ST_If0_Zero : forall t2 t3,
       <{if0 0 then t2 else t3}> --> t2
  | ST_If0_Nonzero : forall n t2 t3,
       <{if0 {S n} then t2 else t3}> --> t3
  (* NEW: macro rule for fix *)
  | ST_FixAbsApp : forall f x T1 T1' t1 v2,
      value v2 ->
      <{ fix (\f:T1->T1', \x:T1, t1) v2 }> -->
      <{ [x := v2] ([f := fix (\f:T1->T1', \x:T1, t1)] t1)}>
```

(No change to typing or substitution rules.)