Homework #5

CS231

Due by the end of the day on December 3. You should submit one file: `hw5.pdf`.

Remember that you are encouraged to work in pairs on homework assignments. See the course syllabus for details. Also remember the course's academic integrity policy. In particular, you must credit other people and other resources that you consulted. Again, see the syllabus for details.

1. In class we saw how to track effects as part of typechecking. Here are the type-and-effect rules for the simply-typed lambda caculus plus booleans. Note that function types are now annotated with an effect for the function body.

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool} \ ; \ \emptyset} \quad \text{(T-TRUE)}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool} \ ; \ \emptyset} \quad \text{(T-FALSE)}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \ ; \ \Phi_1 \qquad \Gamma \vdash t_2 : T \ ; \ \Phi_2 \qquad \Gamma \vdash t_3 : T \ ; \ \Phi_3}{\Gamma \vdash \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T \ ; \ \Phi_1 \cup \Phi_2 \cup \Phi_3} \quad \text{(T-IF)}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T \ ; \ \emptyset} \quad \text{(T-VAR)}$$

$$\frac{\Gamma, x{:}T_1 \vdash t : T_2 \ ; \ \Phi}{\Gamma \vdash \texttt{function } x{:}T_1 \texttt{ -> } t : T_1 \xrightarrow{\Phi} T_2 \ ; \ \emptyset} \quad \text{(T-FUN)}$$

$$\frac{\Gamma \vdash t_1 : T_2 \xrightarrow{\Phi} T \ ; \ \Phi_1 \qquad \Gamma \vdash t_2 : T_2 \ ; \ \Phi_2}{\Gamma \vdash t_1 \ t_2 : T; \ \Phi_1 \cup \Phi_2 \cup \Phi} \quad \text{(T-APP)}$$

We also saw that "checked exceptions" in Java use a form of type-and-effect system. In this problem you will define a variant of that system formally. We'll augment the simply-typed lambda calculus plus booleans with a form of exceptions and exception handling (see Chapter 14 of the text for something similar). For simplicity there will just be one kind of exception, so we don't need to give it a name. Instead we just use a new term `throw` to throw an exception. For example, the expression `function x:Bool -> if x then true else throw` is a function whose result is `true` if the parameter `x` has the value `true`; otherwise the function throws an exception. Note that `throw` is not a value.

A new expression of the form `try` t_1 `catch` t_2 can be used to catch an exception that occurs in the evaluation of t_1 and recover by executing t_2. Its operational semantics is as follows:

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{try t}_1\ \texttt{catch t}_2 \longrightarrow \texttt{try t}_1'\ \texttt{catch t}_2}$$

$$\frac{}{\texttt{try v}_1\ \texttt{catch t}_2 \longrightarrow \texttt{v}_1}$$

$$\frac{}{\texttt{try throw catch t}_2 \longrightarrow \texttt{t}_2}$$

Note that `try` $\texttt{t}_1$ `catch` $\texttt{t}_2$ catches any exception that is thrown anywhere during the evaluation of $\texttt{t}_1$, including in the bodies of functions called from $\texttt{t}_1$, etc. Formalizing this behavior requires additional rules to halt execution and "unwind the stack" whenever an exception is thrown. For example, one rule would say that `if throw then` $\texttt{t}_2$ `else` $\texttt{t}_3$ steps to `throw`. See Chapter 14 for details; these rules are irrelevant for our problem.

Here are the type rules for our new constructs:

$$\frac{}{\Gamma \vdash \texttt{throw : T}}$$

$$\frac{\Gamma \vdash \texttt{t}_1 \texttt{ : T} \qquad \Gamma \vdash \texttt{t}_2 \texttt{ : T}}{\Gamma \vdash \texttt{try t}_1\ \texttt{catch t}_2 \texttt{ : T}}$$

As things stand, we have broken Progress. Specifically, `throw` is well typed but is neither a value nor can it step. We can regain Progress with a type-and-effect system, similar to what we did in class to track mutable memory. Now our effect set Φ will either be the empty set, meaning that an exception cannot be thrown, or the singleton set $\{\texttt{exn}\}$, meaning that an exception may be thrown. Therefore, if done right, we should be able to prove Progress for any term whose effect set is empty (though we will not prove it).

(a) Augment the above two typing rules to have the form $\Gamma \vdash \texttt{t : T ; } \Phi$.

$$\frac{}{\Gamma \vdash \texttt{throw : T};\{\texttt{exn}\}}$$

$$\frac{\Gamma \vdash \texttt{t}_1 \texttt{ : T};\Phi_1 \qquad \Gamma \vdash \texttt{t}_2 \texttt{ : T};\Phi_2}{\Gamma \vdash \texttt{try t}_1\ \texttt{catch t}_2 \texttt{ : T};\Phi_2}$$

(b) Show a typing derivation through your new rules, along with the type-and-effect rules shown above for booleans and for the simply-typed lambda calculus, for the following program, under the empty type environment:

```
try ((function x:Bool -> if x then true else throw) true) catch false
```

```
-------------------- T-Var  ----------------------- T-True  ---------------------------- T-Throw
{x:Bool}|- x:Bool;{}        {x:Bool}|- true:Bool;{}         {x:Bool}|- throw:Bool;{exn}
------------------------------------------------------------------------------------------------ T-If
{x:Bool} |- if x then true else throw : Bool;{exn}
------------------------------------------ T-Fun      --------------- T-True
|- fun x:Bool -> ... : Bool -{exn}-> Bool;{}            |- true:Bool;{}
------------------------------------------------------------------------ T-App  ---------------- T-False
|- (fun x:Bool -> if x then true else throw) true : Bool;{exn}                 |- false:Bool;{}
------------------------------------------------------------------------------------------------ T-Try
|- try ((fun x:Bool -> if x then true else throw) true) catch false : Bool; {}
```

2. For each of the following two types, indicate whether the first can safely be allowed to be a subtype of the second. If yes, just say so. If no, provide a term (in the simply typed lambda calculus plus integers, booleans, pairs, let, sequencing, and references) that would typecheck if we were to allow the two types to be in the subtype relation but would eventually get stuck at run time. You can assume that `Top` is a supertype of all types and that the type system has the following "subsumption" rule, which allows subtyping to be used anywhere:

$$\frac{\Gamma;\Sigma \vdash \texttt{t} : \texttt{T}' \qquad \texttt{T}' <: \texttt{T}}{\Gamma;\Sigma \vdash \texttt{t} : \texttt{T}} \qquad \text{(T-SUB)}$$

   (a) `(Top → Ref Top) ∧ Bool` and `(Top → Top) ∧ Top`
   Yes.

   (b) `(Ref Top) → Top` and `Top → Top`
   No. A counterexample is `(function c:Ref Top -> !c) 0`

   (c) `Ref Top` and `Ref (Top ∧ Top)`
   No. A counterexample is `(function r:Ref (Top ∧ Top) -> (fst !r)) (ref 0)`

   (d) `Ref (Top ∧ Top)` and `Ref Top`
   No. A counterexample is

```
let r1:(Ref (Top ∧ Top)) = ref (0,0) in
  let r2:(Ref Top) = r1 in
    r2:=0;(fst !r1)
```

   Here we use `let x:T = t1 in t2` as shorthand for `((function x:T -> t2) t1)`.

3. A standard compiler optimizations is *constant propagation* which substitutes variables with their constant values at compile time. It employs a dataflow analysis that tracks the values of variables. We can formalize this as a type qualifier system, as we saw in class. Specifically, we will use the following qualifiers for type `Int`:

   `q ::= n | any`

   That is, an `Int` can be annotated with either its constant value `n` or the qualifier `any`, denoting any possible value. So our lattice is simply that `n <: any` for all integers `n`.

   (a) We saw in class some special type rules for integer constants, addition, and unary negation that track `even` and `odd` qualifiers (see the lecture notes from 11/16). Now write new type rules for those constructs, to instead track our qualifiers above.

```
---------------- T-Num
G |- n : n Int
```

```
G |- t1 : q1 Int
G |- t2 : q2 Int
q1 <+> q2 = q
--------------------
G |- t1 + t2 : q Int

where:

n1 <+> n2 = n, where n = n1 [[+]] n2
any <+> _ = any
_ <+> any = any


G |- t : q Int
<->q = q'
-----------------
G |- -t : q' Int

where:

<->n = [[-]]n
<->any = any
```

(b) Use your type system along with the rules for dataflow analysis of imperative statements that we saw in class (see those same lecture notes) to analyze the following piece of code. Show the Γ that would be produced after each program point, starting from an empty initial type environment.

```
x := 8;
if(x > 0)
  y := x + x;
  x := x + 1;
else
  y := 0;
  x := x + 1;
while(x > 0) do
  x := x - 1;
  y := 5;
```

```
                          {}
x := 8;
                          {x : 8 Int}
if(x > 0)
                          {x : 8 Int}
  y := x + x;
                          {x : 8 Int, y : 16 Int}
  x := x + 1;
                          {x : 9 Int, y : 16 Int}
else
                          {x : 8 Int}
  y := 0;
                          {x : 8 Int, y : 0 Int}
  x := x + 1;
                          {x : 9 Int, y : 0 Int}
<merge point>
<iteration 1 of the loop>
                          {x : 9 Int, y : any Int}
while(x > 0) do
  x := x - 1;
                          {x : 8 Int, y : any Int}
  y := 5;
                          {x : 8 Int, y : 5 Int}
<iteration 2 of the loop>
                          {x : any Int, y : any Int}
while(x > 0) do
  x := x - 1;
                          {x : any Int, y : any Int}
  y := 5;
                          {x : any Int, y : 5 Int}
<join>
                          {x : any Int, y : any Int}
<fixpoint reached>
```

4. Recall the definition of the weakest precondition from class (see the notes from 11/18).

   (a) Produce the weakest precondition that ensures that the following Java-like code will not incur a `NullPointerException` or `ArrayOutOfBoundsException`. You may assume the code has already been successfully typechecked. Show the WP computation at every program point.

   ```
   if (x != null) then
     n := x.f;
   else
     n := z-1;
   res := a[n];
   ```

```
                          {(x != null => x != null && a != null && x.f >= 0 && x.f < a.length) &&
                           (!(x != null) => a != null && z-1 >= 0 && z-1 < a.length)}
if (x != null) then
                          {x != null && a != null && x.f >= 0 && x.f < a.length}
  n := x.f;
else
                          {a != null && z-1 >= 0 && z-1 < a.length}
  n := z-1;
                          {a != null && n >= 0 && n < a.length}
res := a[n];
                          {true}
```

(b) Suppose we want to prove that $\{n > 0\}\texttt{code}\{r = n^2\}$, where `code` is the following:

```
k := 0;
r := 0;
s := 1;
while (k != n) {
  r := r + s;
  s := s + 2;
  k := k + 1;
}
```

Come up with a loop invariant for the loop in the above code such that $(n > 0) \Rightarrow \text{WP}(\texttt{code}, r = n^2)$, where the weakest precondition computation uses your loop invariant.
$(r = k^2) \wedge (s = 2k + 1)$

5. This problem will give you some practice with Linear Temporal Logic (LTL).

(a) You own a movie theatre and, like everywhere else, your customers stand in a FIFO queue to get their tickets. You notice that the use of cellphones is on the rise in movie theaters. To make a point, you make a rule saying that folks have to get out of the ticket queue when on the phone and must go to the end of the queue once their call is done. Your customers complain that this is unfair. Particularly, on busy movie days, those who repeatedly answer phone calls are not guaranteed to ever get their tickets. You respond by saying that this is acceptable as you are giving them an opportunity to answer one or more calls without disturbing other patrons — they are guaranteed to get their tickets provided they only spend a finite amount of time on the phone.

Write the liveness property of your queueing system using the predicates:

- $P(x)$, which is true exactly when customer x is on a phone call.
- $T(x)$, which is true exactly when customer x receives a ticket.

Assume that there are an unbounded number of tickets and movie shows. Also, you will issue a ticket whenever the queue is not empty. If you need to make other assumptions, explicitly state them.
$\forall x. FG(\neg P(x)) \rightarrow F(T(x))$

(b) After persistent complaints, you give in and come up with a new system. Every person who comes in to get a ticket gets a token with a monotonically increasing number. You will still not issue tickets to people who are on phone calls. But at every instant, of those people not on the phone, you will issue a ticket to the one with the minimum token number. Express the liveness property of this new system using the predicates $P(x)$ and $T(x)$ above. You can additionally assume that tickets are issued instantaneously.
$\forall x. GF(\neg P(x)) \rightarrow F(T(x))$