Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.820 Foundations of Program Analysis

Problem Set 2

Out: September 24, 2015
Due: October 7, 2015 at 5:00 PM

In this problem set you will have to finish implementing a Haskell program. If necessary, you may consult Problem Set 1 on how to use the Haskell interpreter `ghci` on Athena. We ask you to turn in working code that uses a standard main function so that we can automate testing; in the meantime, you are welcome to use any main function you like to test and debug your code. Note that you can only print values that are printable; in particular, functions cannot be printed, and types declared with a `data` declaration will only be printable if you include a `deriving Show` declaration at the end of the data declaration.

You may discover that you like to get at intermediate results in your code to ensure that they are correct. There is no officially defined way to do this; however, every Haskell implementation provides a `trace` construct, and `ghc` is no exception:

```
trace :: String -> a -> a
```

(In order to use this construct, you'll need a `import Debug.Trace` declaration at the top of your program.) When invoked, `trace` prints its first argument and returns its second argument. You can use `show` to convert printable objects into strings for `trace`. The dollar sign operator `$` represents right-associative function application (that's backwards from the usual application) and provides a handy way to insert traces unobtrusively:

```
fact 0 = 1
fact n = trace ("fact invoked with input " ++ show n) $
 n * fact (n - 1)
```

Another part of this problem set involves proving a theorem with Coq. See the Lecture 4 slides for pointers to getting started with Coq. Use of Coq is a new experiment for 6.820, so please don't be shy when it comes to letting us know about any crucial information that we've neglected to give out!

Please Remember: The problem set is to be handed in before 5 PM on the day it is due. We're changing the submission approach slightly compared to the last problem set: please turn in *one* `.tgz` or `.zip` file containing all of the files you produce for this problem set. Your Problem 3 answer should be in PDF or text format based on your preference (and it's possible we could be talked into permitting other formats). For the Haskell coding and Coq proving problems, please turn in *only the files that you've modified* within your archive file, without any subdirectory structure within that archive.

As always everyone is expected to do their own work. **You *may* discuss with other people Problems 1-2 and 4 only. However, everyone must turn in their own original work. You should type your program/Coq proof solutions yourself, without someone else looking over your shoulder guiding you.**

Problem 1 Coq Warm-Up (5 points)

We'll start with a simple warm-up proof in Coq, about recursive functions over an algebraic datatype.

Start your solution from the template file `ps2.v` which itself refers to another file `ps2-starter.v` containing Coq formalizations of a type of trees, two functions over it, and some lemmas that you may find helpful. Place both files in the same directory. `ps2.v` is the file you'll hand in for this problem and the next one.

The `ps2.v` template contains the theorem statement to prove. You may prove it by any means you like. There are ways to cheat and prove any theorem magically in Coq, like writing `Admitted` to end a "proof," but these aren't allowed here, of course.

The next problem description ends in a quick glossary of tactics that could be helpful. You'll probably also want to use the tactic form `induction x`, for induction over variable `x` from a theorem statement.

Problem 2 Operational Semantics for Exceptions, in Coq (35 points)

This problem will give you some practice with operational semantics and evaluation contexts, while also demonstrating how a popular language feature, exceptions, fits into that framework. We'll start out with a LaTeX definition of λ-calculus with exceptions, including big-step and small-step semantics. The final goal of this problem is to prove one direction of the equivalence between those two notions of semantics, following the basic approach from Lecture 4.

We extend λ-calculus syntax like so:

$$\begin{array}{rcl} e & ::= & x \mid e\ e \mid \lambda x.\ e \mid \mathsf{throw}\ e \mid \mathsf{try}\ e\ \mathsf{catch}\ x \Rightarrow e \\ v & ::= & \lambda x.\ e \end{array}$$

Keeping with the untyped theme of λ-calculus, we allow *any* value to be thrown or caught as an exception.

We can explain how to execute these programs in a variety of ways. Here's a (call-by-value) big-step semantics, where results are either $\mathsf{Normal}(v)$, to indicate normal termination with value v; or $\mathsf{Exception}(v)$, to indicate abnormal termination with uncaught exception v.

$$\frac{}{\lambda x.\ e_1 \Downarrow \mathsf{Normal}(\lambda x.\ e_1)} \qquad \frac{e_1 \Downarrow \mathsf{Normal}(\lambda x.\ e_1') \quad e_2 \Downarrow \mathsf{Normal}(v) \quad e_1'[v/x] \Downarrow r}{e_1\ e_2 \Downarrow r}$$

$$\frac{e_1 \Downarrow \mathsf{Exception}(v)}{e_1\ e_2 \Downarrow \mathsf{Exception}(v)} \qquad \frac{e_1 \Downarrow \mathsf{Normal}(\lambda x.\ e_1') \quad e_2 \Downarrow \mathsf{Exception}(v)}{e_1\ e_2 \Downarrow \mathsf{Exception}(v)}$$

$$\frac{e_1 \Downarrow \mathsf{Normal}(v)}{\mathsf{throw}\ e_1 \Downarrow \mathsf{Exception}(v)} \qquad \frac{e_1 \Downarrow \mathsf{Exception}(v)}{\mathsf{throw}\ e_1 \Downarrow \mathsf{Exception}(v)}$$

$$\frac{e_1 \Downarrow \mathsf{Normal}(v)}{\mathsf{try}\ e_1\ \mathsf{catch}\ x \Rightarrow e_2 \Downarrow \mathsf{Normal}(v)} \quad \frac{e_1 \Downarrow \mathsf{Exception}(v) \quad e_2[v/x] \Downarrow r}{\mathsf{try}\ e_1\ \mathsf{catch}\ x \Rightarrow e_2 \Downarrow r}$$

We can also give a small-step semantics, starting with the definition of evaluation contexts.

$$C \quad ::= \quad \Box \mid C\ e \mid v\ C \mid \mathsf{throw}\ C \mid \mathsf{try}\ C\ \mathsf{catch}\ x \Rightarrow e$$

This sort of context focuses evaluation on the place where we hope to find a redex in an expression. To explain the semantics of throwing and catching exceptions, it will be useful to have a more restrictive context form, which does not allow us to descend inside of a try..catch. We use these contexts to be sure that any thrown exception is caught by the *innermost* enclosing handler.

$$C' \quad ::= \quad \Box \mid C'\ e \mid v\ C' \mid \mathsf{throw}\ C'$$

Presuming the natural definitions of $C[e]$ and $C'[e]$, for plugging an expression into a context, we can proceed to give the four rules defining the small-step operational semantics.

$$\frac{}{C[(\lambda x.\ e)\ v] \longrightarrow C[e[v/x]]} \quad \frac{}{C[\mathsf{try}\ v\ \mathsf{catch}\ x \Rightarrow e] \longrightarrow C[v]}$$

$$\frac{}{C[\mathsf{try}\ (\mathsf{throw}\ v)\ \mathsf{catch}\ x \Rightarrow e] \longrightarrow C[e[v/x]]} \quad \frac{}{C[C'[\mathsf{throw}\ v]] \longrightarrow C[\mathsf{throw}\ v]}$$

The last of these rules may be undesirable, because it supports the effectless small-step judgment $\mathsf{throw}\ v \longrightarrow \mathsf{throw}\ v$, but avoiding such pathological cases requires us to make the rules fancier in ways that are orthogonal to what interests us in this problem.

We would like to know that the two semantics are equivalent. A useful ingredient there is an operator $\lceil r \rceil$, for translating a *result* from the big-step semantics into an *expression* suitable for the small-step semantics. In particular:

$$\begin{aligned} \lceil \mathsf{Normal}(v) \rceil &= v \\ \lceil \mathsf{Exception}(v) \rceil &= \mathsf{throw}\ v \end{aligned}$$

Your task is to prove in Coq the formal equivalent of the following theorem, which states one direction of the semantics equivalence, using $\longrightarrow^*$ to refer to the transitive-reflexive closure of $\longrightarrow$:

$$\forall e, r.\ \text{if } e \Downarrow r \text{ then } e \longrightarrow^* \lceil r \rceil$$

The instructions from Problem 1 should already have set up your working `.v` file, where you'll fill in this proof. As before, `ps2-starter.v` both defines the semantics and proves some lemmas that should be useful as you complete this problem. The same rules about cheap-shot "proofs" like `Admitted` apply here, but otherwise you're free to prove the theorem however you like. Here is a quick glossary of the tactics that we used in our solution, which is about 70 lines of tactic code.

- `apply THM`: apply a theorem or lemma `THM`, replacing the goal with one subgoal for each premise of `THM`.

- `apply (THM t1 ... tN)`: like `apply`, but uses expressions `t1` through `tN` as the values of the first `N` quantified variables in the statement of `THM`.

- `apply THM in H`: where `H` matches the *hypothesis* of theorem `THM`, replace `H` with the *conclusion* of `THM`.

- `assumption`: prove a conclusion ("below the line") that matches one of the known hypotheses ("above the line").

- `eapply THM`: like `apply`, but works when not all quantified variables of `THM` can be deduced immediately, leaving placeholders to figure out later.

- `eassumption`: like `assumption`, but figures out values of placeholders introduced by `eapply`.

- `induction N`: induction on the `N`th hypothesis appearing in the conclusion, numbering from left to right starting with 1.

- `inversion_clear H`: case analysis on which rule was used to prove the judgment in hypothesis `H`.

  - If you're using an older version of Coq that doesn't include this tactic, you can instead write the more wordy `inversion H; clear H; subst`.

- `simpl`: perform computational simplifications in the conclusion part of the goal only.

- `simpl in *`: perform computational simplifications throughout the goal.

Problem 3 Basic Hindley-Milner typechecking (25 points)

This problem focuses on typechecking in both our simple type system as well as the basic Hindley-Milner type systems. In this problem we will ignore overloading. Remember $\rightarrow$ is right associative; that means you read $a \rightarrow b \rightarrow c$ as $a \rightarrow (b \rightarrow c)$.

Part a: (7 points) Simple Types

Give the simple Hindley-Milner type (i.e. no polymorphism) for the last named, top-level function defined in each of the following code snippets. Each snippet is worth 1 point. Assume that all arithmetic operations take arguments of type Int and that all comparisons return results of type $Bool$. If a type variable is not fully constrained then just assign it type `Free`.

1.
```
det a b c = (b * b) - 4 * a * c
```

2.
```
step (a,b) = (b,b+1)
```

3.
```
sum f n =
    if (n<0) then  0
             else  sum f (n-1) + f n

  sumSum f n = sum (sum f) n
```

4.
```
repeat n f x =
    if (n==0) then  x
              else  repeat (n-1) f (f x)

  decrement n =
    let (result, n_again) = repeat n step (-1,0)
    in  result
```

5.
```
loopy x = loopy x
```

Part b: (10 points) Simple Types and Hindley-Milner

Here are some more Haskell snippets. For each one:

- Determine if it types in our simple type system.
- State whether it types in the HM type-system (i.e. with polymorphism).
- If it types in our simple system, give the type of the last named, top-level function. Otherwise, if it types in the HM type system, give the principal type of the last named, top-level function. Otherwise, state why it was not typeable.

1.
```
f x = if x then x+3 else x*2
```

2. ```
   foo f =
     let func x = f x x
     in func
   ```

3. ```
   r g x y = if (g x) then  g y
                      else  1+(g y)
   ```

4. ```
   s g =
     let h x = g (g x)
     in  h (h True, h False)
   ```

5. ```
   id x = x

   herbert y =
      if (id y) then (id 0)
                else (id 1)
   ```

Part c: (8 points) Theorems for Free

Given a function type, there are often only a few functions we can define that have that type, if we restrict our attention only to functions that *always terminate normally*, which rules out such cop-outs as raising exceptions. For example, we can only write one function that has the type $a \rightarrow a$:

```
ident x = x
```

This phenomenon is called "theorems for free" in the functional programming world, because we are able to deduce certain theorems about behavior of functions just by considering their types.

Try to come up with functions that have the following types and always terminate normally. Be careful not to give functions whose types are too general! Use Haskell syntax. A few questions may have several possible answers; you only need to give one. (1 point for each question)

1. $a \rightarrow Int$
2. $(a, b) \rightarrow (b, a)$
3. $a \rightarrow b \rightarrow a$
4. $a \rightarrow b \rightarrow b$
5. $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$
6. $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$
7. $(a \rightarrow b \rightarrow c) \rightarrow (a, b) \rightarrow c$
8. $(a \rightarrow c) \rightarrow (b \rightarrow d) \rightarrow (a, b) \rightarrow (c, d)$

Problem 4 Implementing a Simple Typechecker with Inference (35 points)

In this problem, we will be writing a simple program that performs simple type inference on a small λ-calculus language. In our small language, we have the following definitions for a simple λ-calculus language with the following expressions and statements.

```
data Exp =
    EVar Ident
  | ELambda Ident Exp
  | EApp Exp Exp
  | ECond Exp Exp Exp
  | EPrim Prim
  | ELet Stmt Exp

data Stmt =
    SEmpty
  | SAssign Ident Exp
  | SSeq Stmt Stmt
```

We don't require you to handle the case of `ELet` with multiple bindings of the same variable. We consider such definitions ill-formed, but we won't require you to detect them (though you're free to do so if you like).

We also have the following simple types.

```
data BaseType = BTInt | BTBool
  deriving (Eq, Show)
data Type =
    TBase BaseType
  | TVar TVar
  | TArrow Type Type
  deriving (Eq, Show)
```

You will be writing a program that infers these types given an expression. We will only test your code on examples that either lead to type errors or produce fully determined types with no `TVar`s, so those type variables are intended for use in intermediate stages of inference.

Part a: (2 points) Download `ps2Code.zip` from the course website. Make sure you can compile the assignment skeleton using the following additional options:

```
ghc -XDeriveDataTypeable --make Main.hs
```

Make sure you understand the data types and the utility functions we have provided you.

How are we handling exceptions in this program? How could we handle exceptions *without* using `Control.Exception` and `catch`?

[For this and other parts of the problem that ask for prose answers, please include your answers in the PDF or text file that you hand in.]

Part b: (3 points) Provide an implementation for `freeVarsInEnv`, which returns the free variables in the environment. You may want to write an additional function,

```
freeVars :: Type -> [TVar]
```

Part c: (7 points) In our type-checking framework, we represent substitutions as a mapping of type variables to types:

```
type Subst = [(TVar, Type)]
idSubst = []
```

Provide an implementation for unification, which produces a substitution that may bind additional type variables:

```
unify :: Constraints -> Subst
```

Note that this function can either unify a pair of types or a list of constraints on types, depending on how you choose to implement `inferTypes`.

Part d: (10 points)

Provide an implementation for `inferTypes`, which takes a constraint environment and an expression and returns a (possibly modified) constraint environment with the resulting type. Depending on how you decide to implement your type checker, this function may call `unify`.

```
inferTypes :: ConstraintEnv -> Exp -> (ConstraintEnv, Type)
```

Part e: (3 points) Now you are ready to write the type-checking function:

```
typeCheck :: TEnv -> Exp -> Type
```

Part f: (7 points) Write tests for your type checker in `Tests.hs` to test the different paths of the type checker. Make sure at least some of these tests cause your type-checker to fail.

Part g: (2 points) How would we extend this implementation to support HM-style polymorphism?

Part h: (1 points) What state did we store while writing this program? How could effects (namely, mutation) have made this implementation "easier"?

6.820 Fundamentals of Program Analysis

Fall 2015