Homework 5

Due 15 March

Handout 11
CSCI 334: Spring, 2012

Reading

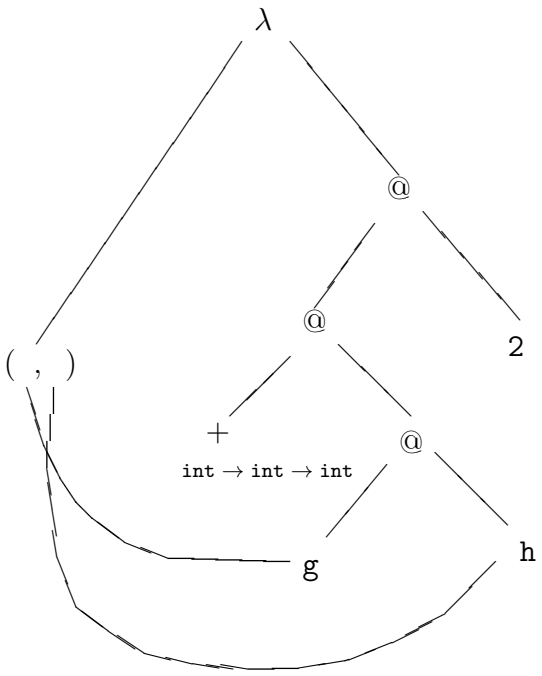
1. **(Required)** Read Mitchell, Chapters 6 and 7.

Problems

1. (*10 points*) ..... Parse Graph

Use the parse graph below to calculate the ML type for the function

```
fun f(g,h) = g(h) + 2;
```



2. (*15 points*) ..... Type Inference to Detect Race Conditions

The general techniques from our type inference algorithm can be used to examine other program properties as well. In this question, we look at a non-standard type inference algorithm to determine whether a concurrent program contains race conditions. Race conditions occur when two threads access the same variable at the same time. Such situations lead to non-deterministic behavior, and these bugs are very difficult to track down since they may not appear every time the program is executed. For example, consider the following program, which has two threads running in parallel:

```
Thread 1:
  t1 := !hits;
  hits := !t1 + 1;
```

```
Thread 2:
  t2 := !hits;
  hits := !t2 + 1;
```

Since the threads are running in parallel, the individual statements of Thread 1 and Thread 2 can be interleaved in many different ways, depending on exactly how quickly each thread is allowed to execute. For example, the two statements from Thread 1 could be executed before the two statements from Thread 2, giving us the following execution trace:

$$\texttt{hits} = 0 \xrightarrow{\texttt{t1:=!hits}} \texttt{hits} = 0 \xrightarrow{\texttt{hits:=!t1+1}} \texttt{hits} = 1 \xrightarrow{\texttt{t2:=!hits}} \texttt{hits} = 1 \xrightarrow{\texttt{hits:=!t2+1}} \texttt{hits} = 2$$

After all four statements execute, the `hits` counter is updated from zero to 2, as expected. Another possible interleaving is the following:

$$\texttt{hits} = 0 \xrightarrow{\texttt{t2:=!hits}} \texttt{hits} = 0 \xrightarrow{\texttt{hits:=!t2+1}} \texttt{hits} = 1 \xrightarrow{\texttt{t1:=!hits}} \texttt{hits} = 1 \xrightarrow{\texttt{hits:=!t1+1}} \texttt{hits} = 2$$

This again adds 2 to `hits` in the end. However, look at the following trace:

$$\texttt{hits} = 0 \xrightarrow{\texttt{t1:=!hits}} \texttt{hits} = 0 \xrightarrow{\texttt{t2:=!hits}} \texttt{hits} = 0 \xrightarrow{\texttt{hits:=!t1+1}} \texttt{hits} = 1 \xrightarrow{\texttt{hits:=!t2+1}} \texttt{hits} = 1$$

This time, something bad happened. Although both threads updated `hits`, the final value is only 1. This is a race condition: the exact interleaving of statements from the two threads affected the final result. Clearly, race conditions should be prevented since it makes ensuring the correctness of programs very difficult. One way to avoid many race conditions is to protect shared variables with mutual exclusion locks. A lock is an entity that can be held by only one thread at a time. If a thread tries to acquire a lock while another thread is holding it, the thread will block and wait until the other thread has released the lock. The blocked thread may acquire it and continue at that point. The program above can be written to use lock `l` as follows:

```
Thread 1:                          Thread 2:
  synchronized(l) {                  synchronized(l) {
    t1 := !hits;                       t2 := !hits;
    hits := !t1 + 1;                   hits := !t2 + 1;
  }                                    }
```

The statement "`sychronized(l) { s }`" aquires lock `l`, executes `s`, and then releases lock `l`. There are only two possible interleavings for the program now:

$$\texttt{hits} = 0 \xrightarrow{\texttt{t1:=!hits}} \texttt{hits} = 0 \xrightarrow{\texttt{hits:=!t1+1}} \texttt{hits} = 1 \xrightarrow{\texttt{t2:=!hits}} \texttt{hits} = 1 \xrightarrow{\texttt{hits:=!t2+1}} \texttt{hits} = 2$$

and

$$\texttt{hits} = 0 \xrightarrow{\texttt{t2:=!hits}} \texttt{hits} = 0 \xrightarrow{\texttt{hits:=!t2+1}} \texttt{hits} = 1 \xrightarrow{\texttt{t1:=!hits}} \texttt{hits} = 1 \xrightarrow{\texttt{hits:=!t1+1}} \texttt{hits} = 2$$

All others are ruled out because only one thread can hold lock `l` at a time. Note that while we use assignable variables inside the synchronized blocks, the names we use for locks are constant. For example, the name `l` in the example program above always refers to the same lock.

Our analysis will check to make sure that locks are used to guard shared variables correctly. In particular, our analysis checks the following property for a program P:

> For any variable `y` used in P, there exists some lock `l` that is held by the current thread every time `y` is accessed.

In other words, our analysis will verify that every access to a variable `y` will occur inside the synchronized statement for some lock `l`. Checking this property usually uncovers many race conditions.

Let's start with a simple program containing only one thread:

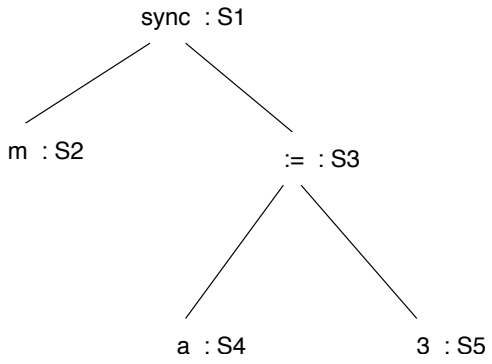
```
Thread 1:
  synchronized (m) {
    a := 3;
  }
```

For this program, our analysis should infer that lock `m` protects variable `a`.

As with standard type inference, we proceed by labeling nodes in the parse tree, generating constraints, and solving them.

Step 1: Label each node in the parse tree for the program with a variable. This variable represents the set of locks held by the thread every time execution reaches the statement represented by that node of the tree. Note that these variables keep track of sets of locks names, and NOT types, in this analysis.

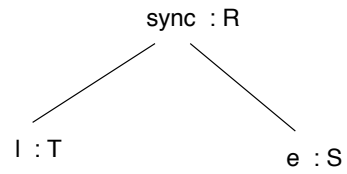
Here is the labeled parse tree for the example:



Step 2: Generate the constraints using the following four rules:

(a) If S is the variable on the root of the tree, then $S = \emptyset$.

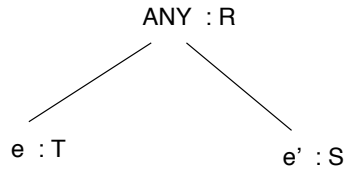
(b) For any subtree matching the form



we add two constraints:

$$\begin{aligned} T &= R \\ S &= R \cup \{\texttt{l}\} \end{aligned}$$

(c) For any subtree matching the form



where ANY matches any node other than a `sync` node, we add two constraints:

$$\begin{aligned} T &= R \\ S &= R \end{aligned}$$

(d) To determine $\texttt{lock}_\texttt{y}$, the lock guarding variable y, add the constraint

$$\texttt{lock}_\texttt{y} \quad \in \quad S$$

for each node $\texttt{y} : S$ or $!\texttt{y} : S$ in the tree. In other words, require that $\texttt{lock}_\texttt{y}$ be in the set of locks held at each location y is accessed.

Here are the constraints generated for the example program:

$$\begin{array}{rcll} S1 & = & \emptyset & \text{(rule 2a)} \\ S2 & = & S1 & \text{(rule 2b)} \\ S3 & = & S1 \cup \{\texttt{m}\} & \text{(rule 2b)} \\ S4 & = & S3 & \text{(rule 2c)} \\ S5 & = & S3 & \text{(rule 2c)} \\ \texttt{lock}_\texttt{a} & \in & S4 & \text{(rule 2d)} \end{array}$$

Step 3: Solve the constraints to determine the set of locks held at each program point and which locks guard the variables:

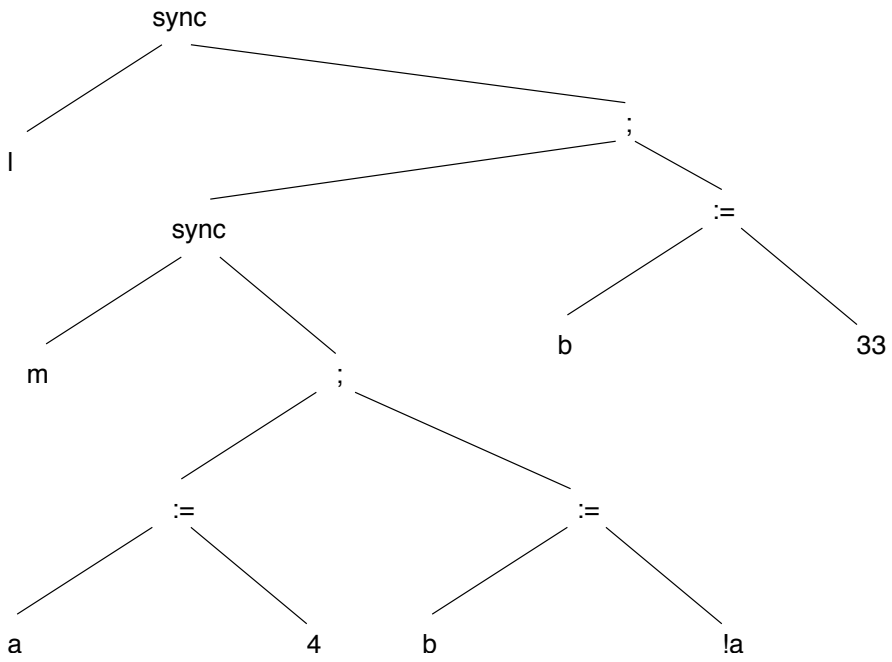
$$\begin{array}{rcl} S2 = S1 & = & \emptyset \\ S3 = S4 = S5 & = & \{\texttt{m}\} \\ \texttt{lock}_\texttt{a} & \in & \{\texttt{m}\} \end{array}$$

Clearly, $\texttt{lock}_\texttt{a}$ is m in this case, exactly as we expected.

You will now explore some aspects of this analysis:

(a) Here is another program and corresponding parse tree:

```
Thread 1:
  synchronized (l) {
    synchronized (m) {
      a := 4;
      b := !a;
    }
    b := 33;
  }
```



Compute $\texttt{lock}_a$ and $\texttt{lock}_b$ using the algorithm above. Explain why the result of your algorithm makes sense.

(b) Let's go back to the original example, but change Thread 2 to use a different lock:

```
Thread 1:                          Thread 2:
  synchronized(l) {                  synchronized(m) {
    t1 := !hits;                       t2 := !hits;
    hits := !t1 + 1;                   hits := !t2 + 1;
  }                                  }
```

Compute $\texttt{lock}_{t1}$, $\texttt{lock}_{t2}$, and $\texttt{lock}_{hits}$ using the algorithm above. Since there are two threads in the program, you should create two parse trees, one for each thread. Explain the result of your algorithm.

(c) Suppose that we allow assignments to lock variables. For example, in the following program, `l` and `m` are references to locks, and we can change the locks to which those names refer with an assignment statement:

```
Thread 1:                          Thread 2:
  synchronized(!l) {                  synchronized(!m) {
    a := !a + 1;                        x := !b + 3
  }                                     b := 11 + x;
  m := !l;                            }
  synchronized(!m) {
    a := !b + 1;
    b := !a;
  }
```

Describe any problems that arise due to assignments to lock variables, and what the implications for the analysis are. You do not have to show the constraints from this example or change the analysis to handle mutable lock variables. A coherent discussion of the issues is sufficient. Thinking about what the algorithm would compute for $\texttt{lock}_a$, $\texttt{lock}_b$, and $\texttt{lock}_x$ may be useful, however.

3. (15 points) Folding Fun

The "fold-left" and "fold-right" functions appear in many languages (as `reduceRight/Left` in Javascript, as `accumulate` in C++, as `foldl/foldr` in ML, and so on.)

Here are their definitions in ML:

```
fun foldr f v nil =     v
  | foldr f v (x::xs) = f (x, foldr f v xs);

fun foldl f v nil =     v
  | foldl f v (x::xs) = foldl f f(x, v) xs;
```

Thus, `foldr g b [`a_0`, ..., `a_n`]` computes

`g(`a_0`, g(`a_1`, g(`a_2`, ... g(`a_n`, b) ... )))`

and `foldl g b [`a_0`, ..., `a_n`]` computes

`g(`a_n`, g(`a_{n-1}`, g(`a_{n-2}`, ..., g(`a_0`, b) ... )))`

The "fold-right" function reduces the elements in a list to a single value by repeated application of g, starting at the right of the list and working to the left. The "fold-left" function starts from the left and works to the right.

Here is an example usage, which defines a function `sum` that adds together the numbers in a list:

```
- fun add(x,y) = x+y;
- fun sum elems = foldr add 0 elems;
- sum [2,3,4];
val it = 9: int
```

In effect, `sum [2,3,4]` computes

```
add(2, add(3, add(4, 0)))
```

We could also define `sum` using `foldl`:

```
- fun sum2 elems = foldl add 0 elems;
```

in which case `sum2 [2,3,4]` computes

```
add(4, add(3, add(2, 0)))
```

Of course, we typically combine folding with anonymous functions, as in the following definition of `sum`:

```
- fun sum elems = foldr (fn (x,result) => (x+result)) 0 elems;
```

The type of both `foldr` and `foldl` is

```
('c * 'd -> 'd) -> 'd -> 'c list -> 'd
```

That is, it takes as parameters a reducing function, an initial value, and a list. It produces a single summary value.

(a) Can we always use `foldl` in place of `foldr`? If yes, explain. If no, give an example function `f`, list `l`, and initial value `v` such that `foldr f v l` and `foldl f v l` behave differently.

(b) Using a fold operation, write a function `words_length: string list -> int`. This function should return the total length of all words appearing in a list of strings. For example:

```
- words_length nil;
val it = 0 : int
- words_length ["Three", "Short", "Words"];
val it = 15 : int
```

(c) Using a fold, write a function `count: ''a -> ''a list -> int`. It computes the number of times a value appears in a list. For example:

```
- count "sheep" ["cow", "sheep", "sheep", "goat"];
val it = 2 : int
- count 4 [1,2,3,4,1,2,3,4,1,2,3,4];
val it = 3 : int
```

(d) Using a fold, write a function `poly: real list -> (real -> real)` that takes a list of reals $[a_0, a_1, \ldots, a_{n-1}]$ and returns a function that takes an argument `b` and evaluates the polynomial

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$$

at $x = b$; that is, it computes $\sum_{i=0}^{n-1} a_i b_i$. For example,

```
- val g = poly [1.0, 2.0];
val it = fn: real -> real
- g(3.0);
val it = 7.0: real
- val g = poly [1.0, 2.0, 3.0];
val it = fn: real -> real
- g(2.0);
val it = 17.0: real
```

(Hint: $a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + x(a_1 + x(a_2 + x a_3))$. This is an example of Horner's Rule. Horner's Rule demonstrates that we can evaluate a degree n polynomial with only $O(n)$ multiplies.)

(e) Folds over lists are one specific example of what's called a "catamorphism," which generalizes transformations between datatypes. (You may also encounter catamorphisms in an abstract algebra class...) Here is a polymorphic tree definition and `foldTree` operation that perfroms a "fold-like" transformation on a tree:

```
datatype ('a, 'c) Tree = Leaf of 'a
                       | Node of 'c * ('a, 'c) Tree * ('a, 'c) Tree;

fun foldTree (leafFn, nodeFn) (Leaf v) = leafFn(v)
  | foldTree (leafFn, nodeFn) (Node (c, left,right)) =
      let val leftValue =  foldTree(leafFn, nodeFn) left;
          val rightValue = foldTree(leafFn, nodeFn) right
      in
          nodeFn(c, leftValue, rightValue)
      end;
```

The tree stores `'a` values on the leaves and `'c` values on the internal nodes. The `foldTree` operation applies the function `leafFn` to each leaf and the `nodeFn` to each internal node and the result of folding each subtree. Thus, to apply fold to a tree, you provide two functions describing: 1) what to do to a leaf, and 2) what to do to an interior node. As a specific example, given

```
val e = Node(w1, Node(w2, Leaf v1, Leaf v2), Leaf v3);
```

the expression `foldTree (f,g) e` computes:

`g(w1, g (w2, f(v1), f(v2)), f(v3))`.

Using `foldTree`:

- Write a function `treeDepth: ('a,'b) Tree -> int`. It should have the following form:

```
fun treeDepth tree = foldTree (..., ...) tree;
```

  where each `...` is replaced by a lambda function or helper function that you define.
- Write a function `treeEval: ExprTree -> int` that evaluates an expression tree where the leaves store `int`s and the nodes are `Op`s, defined below:

```
datatype Op = PLUS | MULT;
type ExprTree = (int, Op) Tree;

fun treeEval (tree : ExprTree) = foldTree(..., ...) tree;
```

  With these definitions, we have:

```
- treeEval Node(PLUS, Leaf 3, Node(MULT, Leaf 4, Leaf 5));
val it = 23 : int
```

4. (*15 points*) ..... Ada Parameter Modes

The Ada programming language permits parameters to be labeled as `in`, `out`, or `in out`, as in the following procedure definitions, where `T` is some type:

```
procedure test1(in x: T) is begin ... end
procedure test2(out x: T) is begin ... end
procedure test3(in out x: T) is begin ... end
```

The modifiers, or modes, have the following meaning:

- `in`: The value of the parameter `x` cannot be changed inside the procedure. If we call `test1(y)`, the value of `y` is the same before and after the call..
- `out`: The parameter `x` can be written to, but it cannot be read. If we call `test2(y)`, the value of `y` after the call is the last value written to `x` in the procedure.
- `in out`: The parameter `x` can be both read and written, and the value of `y` after a call to `test3(y)` is the last value written to `x` in the procedure.

The language definition does not specify how each mode should be implemented, and the compiler may use any appropriate parameter passing mechanism to implement them.

(a) Which parameter passing mechanism could be used to implement `test1`, `test2`, and `test3`? The choices are pass-by-reference, pass-by-value, and pass-by-value-result (as described in problem 7.6). If more than one is possible, describe the advantages/disadvantages of each.

(b) Consider the following procedure that takes two parameters. Does the following program print the same value for all strategies you outlined for `in out` parameters above?

```
procedure incTwo(in out x:integer, in out y:integer) is
  begin
    x := x + 1;
    y := y + 1;
  end

procedure main() is
```

```
w : integer = 3;
begin
  incTwo(w,w);
  print w;
end
```

(c) Discuss the advantages and disadvantages of permitting the compiler such flexibility in how it implements parameter modes.

5. (*10 points*) ..... Static and Dynamic Scope

Mitchell, Problem 7.8

6. (*15 points*) ..... Function Calls and Memory Management

Mitchell, Problem 7.12

7. (*15 points*) ..... Function Returns and Memory Management

Mitchell, Problem 7.13