What To Turn In

Please hand in work in two pieces, one for the Problems and one for the Programming (Partner Optional):

Problems: Turn in handwritten or typed answers by the due date. Be sure your work is stapled and that your answers are clearly marked and in the correct order.

Programming (Partner Optional): Turn in a printout of your code separately from your problem answers. If you worked with a partner, only one person needs to hand in a printout, but be sure both IDs are at the top of the file. Turn in an electronic copy as well using the instructions at the end of the programming questions.

Reading

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

Problems

Q1. (10 points) Parse Graph

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

fun f(g,h) = g(h) + 2;
Q2. (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:
\[
t1 := \neg hits; \\
hits := \neg t1 + 1;
\]

Thread 2:
\[
t2 := \neg hits; \\
hits := \neg t2 + 1;
\]

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:

\[
hits = 0 \quad t1:=\neg hits \quad \rightarrow \quad hits = 0 \quad \neg hits:=\neg t1+1 \quad \rightarrow \quad hits = 1 \quad t2:=\neg hits \quad \rightarrow \quad hits = 1 \quad \neg hits:=\neg t2+1 \quad \rightarrow \quad hits = 2
\]

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

\[
hits = 0 \quad t2:=\neg hits \quad \rightarrow \quad hits = 0 \quad \neg hits:=\neg t2+1 \quad \rightarrow \quad hits = 1 \quad t1:=\neg hits \quad \rightarrow \quad hits = 1 \quad \neg hits:=\neg t1+1 \quad \rightarrow \quad hits = 2
\]

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

\[
hits = 0 \quad t1:=\neg hits \quad \rightarrow \quad hits = 0 \quad t2:=\neg hits \quad \rightarrow \quad hits = 0 \quad \neg hits:=\neg t1+1 \quad \rightarrow \quad hits = 1 \quad \neg hits:=\neg t2+1 \quad \rightarrow \quad hits = 1
\]

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 1 as follows:

Thread 1:
\[
synchronized(1) \{ \\
\quad t1 := \neg hits; \\
\quad hits := \neg t1 + 1;
\}
\]

Thread 2:
\[
synchronized(1) \{ \\
\quad t2 := \neg hits; \\
\quad hits := \neg t2 + 1;
\}
\]

The statement “synchronized(1) \{ s \}” acquires lock 1, executes s, and then releases lock 1. There are only two possible interleavings for the program now:

\[
hits = 0 \quad t1:=\neg hits \quad \rightarrow \quad hits = 0 \quad \neg hits:=\neg t1+1 \quad \rightarrow \quad hits = 1 \quad t2:=\neg hits \quad \rightarrow \quad hits = 1 \quad \neg hits:=\neg t2+1 \quad \rightarrow \quad hits = 2
\]

and

\[
hits = 0 \quad t2:=\neg hits \quad \rightarrow \quad hits = 0 \quad \neg hits:=\neg t2+1 \quad \rightarrow \quad hits = 1 \quad t1:=\neg hits \quad \rightarrow \quad hits = 1 \quad \neg hits:=\neg t1+1 \quad \rightarrow \quad hits = 2
\]

All others are ruled out because only one thread can hold lock 1 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 1 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:

```java
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:

```
sync : S1

  m : S2

  := : S3

    a : S4

    3 : S5
```

**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

```
sync : R

  l : T

  e : S
```

we add two constraints:

\[
T = R \\
S = R \cup \{l\}
\]

(c) For any subtree matching the form
where ANY matches any node other than a sync node, we add two constraints:

\[
\begin{align*}
T &= R \\
S &= R 
\end{align*}
\]

(d) To determine \(\text{lock}_y\), the lock guarding variable \(y\), add the constraint

\[\text{lock}_y \in S\]

for each node \(y : S\) or \(\bar{y} : S\) in the tree. In other words, require that \(\text{lock}_y\) be in the set of locks held at each location \(y\) is accessed.

Here are the constraints generated for the example program:

\[
\begin{align*}
S_1 &= \emptyset & \text{(rule 2a)} \\
S_2 &= S_1 & \text{(rule 2b)} \\
S_3 &= S_1 \cup \{m\} & \text{(rule 2b)} \\
S_4 &= S_3 & \text{(rule 2c)} \\
S_5 &= S_3 & \text{(rule 2c)} \\
\text{lock}_a &\in S_4 & \text{(rule 2d)}
\end{align*}
\]

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

\[
\begin{align*}
S_2 = S_1 &= \emptyset \\
S_3 = S_4 = S_5 &= \{m\} \\
\text{lock}_a &\in \{m\}
\end{align*}
\]

Clearly, \(\text{lock}_a\) 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:
```java
synchronized (l) {
    synchronized (m) {
        a := 4;
        b := !a;
    }
    b := 33;
}
```
Compute \( \text{lock}_a \) and \( \text{lock}_b \) 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:
```
synchronized(l) {
    t1 := \!\text{hits};
    \text{hits} := \!\text{t}1 + 1;
}
```

Thread 2:
```
synchronized(m) {
    t2 := \!\text{hits};
    \text{hits} := \!\text{t}2 + 1;
}
```

Compute \( \text{lock}_t_1 \), \( \text{lock}_t_2 \), and \( \text{lock}_\text{hits} \) 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:
```
synchronized(\!l) {
    a := \!a + 1;
    m := \!l;
}
```

Thread 2:
```
synchronized(\!m) {
    x := \!b + 3;
    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 \( \text{lock}_a \), \( \text{lock}_b \), and \( \text{lock}_x \) may be useful, however.
Q3. (10 points) Parameter Passing

Mitchell, Problem 7.4

Q4. (20 points) Swift Parameter Modes

This question explores parameter modes in the context of Apple's Swift language. We'll look at three modes: in, in-out, and out. Here is an example of each in Swift:

```swift
func test1(x: Int) -> Int { ... } /* default is "in" mode */
func test2(x: inout Int) -> Int { ... } /* "inout" mode */
func test3(x: out Int) -> Int { ... } /* "out" mode */
```

(Swift doesn't actually support the “out” mode, but other languages do, most notably Ada.) The three modes, have the following meaning:

**in**: The value of the parameter `x` cannot be changed inside the function. If we call `test1(y)`, the value of `y` is the same before and after the call.

**in-out**: The parameter `x` can be both read and written, and the value of `y` after a call to `test2(&y)` is the last value written to `x` in the function. The “&” in the call to `text2` is required by Swift to indicate that `y` is being passed as an in-out parameter.

**out**: The parameter `x` can be written to, but it cannot be read. If we call `test3(&y)`, the value of `y` after the call is the last value written to `x` in the function.

(a) Why do you think the designers of Swift require the “&” for in-out parameters at call sites?

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

(c) In general, what is the advantage of permitting the compiler flexibility in how it implements parameter modes for such functions?

(d) Now consider the following function that takes two parameters. Would the function `main` print the same value for all strategies you outlined for `test2` above? If it doesn’t, why might that be problematic?

```swift
func incTwo(a: inout Int, b: inout Int) {
    a += 1;
    b += 1;
}

func main() {
    var w : Int = 3;
    incTwo(&w, &w);
    print(w);
}
```

(e) Swift disallows passing the same variable as two different in-out parameters, meaning the call to `incTwo(&w, &w)` would be an error. Is this sufficient to avoid any problematic behavior identified in the previous part? There are alternative ways to achieve the same effect. Identify at least one. Why did you think the designers of Swift choose the option they did?
Q5. (10 points) .................................................. Static and Dynamic Scope

Mitchell, Problem 7.8

Q6. (15 points) ......................... Function Calls and Memory Management

Mitchell, Problem 7.12

Q7. (15 points) ......................... Function Returns and Memory Management

Mitchell, Problem 7.13

(Note: \( g \) in the diagram in the book is a pointer and should have a dot next to it like the other pointers.)

Programming (Partner Optional)

P1. (30 points) ................................................................. Folding Fun

Your GitLab account will have a project for your to use for this question. You can follow the same instructions as on HW 1 for cloning it and adding a (optional) partner.

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:

\[
\begin{align*}
\text{fun foldr } f \ v \ \text{nil} &= \ v \\
&| \text{foldr } f \ v \ (x::xs) = f \ (x, \ \text{foldr } f \ v \ xs) \\
\text{fun foldl } f \ v \ \text{nil} &= \ v \\
&| \text{foldl } f \ v \ (x::xs) = \text{foldl } f \ (f(x, \ v)) \ xs;
\end{align*}
\]

Thus, \( \text{foldr} \ g \ b \ [a_0, \ldots, a_n] \) computes
\[
g(a_0, g(a_1, g(a_2, \ldots g(a_n, b) \ldots )))
\]
and \( \text{foldl} \ g \ b \ [a_0, \ldots, a_n] \) computes
\[
g(a_n, g(a_{n-1}, g(a_{n-2}, \ldots, g(a_0, b) \ldots )))
\]

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 \( \text{sum} \) that adds together the numbers in a list:

\[
\begin{align*}
- \text{fun add}(x,y) &= x+y; \\
- \text{fun sum elem} &= \text{foldr} \ \text{add} \ 0 \ \text{elem} \\
- \text{sum} \ [2,3,4]; \\
\text{val it} &= 9: \ \text{int}
\end{align*}
\]

In effect, \( \text{sum} \ [2,3,4] \) computes
\[
\text{add}(2, \ \text{add}(3, \ \text{add}(4, 0)))
\]
Writing that function recursively would give us:

```ml
fun sum_rec nil = 0
  | sum_rec (x::xs) = x + sum_rec(xs);
```

which computes the exact same value: `sum_rec [2,3,4]` computes `2 + (3 + (4 + 0))`. Many computations involve traversing a list and computing a “summary” value for it. We explore other examples below, and our folding operations enable us to write them in a succinct, elegant way.

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

```ml
- 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`:

```ml
- 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.

(o) Using a fold operation, write a function `concatWords: string list -> string`. This function should return a string with all strings in the list concatenated:

```ml
- concatWords nil;
  val it = "" : string
- concatWords ["Three", "Short", "Words"];
  val it = "ThreeShortWords" : string
```

(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:

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

(c) 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.

(d) 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:

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

(e) Using a fold, write a function `partition: int -> int list -> int list * int list` that takes an integer `p` and a list of integers `l`, and that returns a pair of lists containing the elements of `l` smaller than `p` and those greater than or equal to `p`. The ordering of the original list should be preserved in the returned lists. (We wrote a recursive form during lecture as part of quicksort.)
Using a fold, write a function \( \text{poly}: \text{real list} \to (\text{real} \to \text{real}) \) 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 \( g = \text{poly} [1.0, 2.0] \);
  val it = fn: real \to real
  - \( g(3.0) \);
  val it = 7.0: real
- val \( g = \text{poly} [1.0, 2.0, 3.0] \);
  val it = fn: real \to real
  - \( g(2.0) \);
  val it = 17.0: real

(Hint: \( a_0 + a_1 x + a_2 x^2 + a_3 x^3 = a_0 + x(a_1 + x(a_2 + xa_3)) \). 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.)

What To Turn In.

- Your code for these questions should be documented — comments have the form “(* comment *)” — and include your ID (and that of your partner) at the top.
- Turn in a printout of your code file separate from the answers to the written problems. If you worked with a partner, please turn in only one copy.
- Also, commit and push all of your code to our Gitlab server.