Computer Science & Engineering 505 – Final

December 10, 1999
Open book & notes – 110 minutes – 10 points per question
90 points total

Name: _____

Please write any long answers on the back of the exam or on a separate piece of paper.

1. Suppose the following Miranda script has been loaded. (This script is type correct.)

```
fmap [] [] = []
fmap (f : fs) (x : xs) = f x : fmap fs xs

manyconsts = m 1
             where m i = my_const i : m (i+1)

my_const k x = k

double x = x+x
square x = x*x

my_abs x = x, if x>=0
         = -x, otherwise

mystery f x = f x : mystery f (f x)
```

   What is the result of evaluating the following Miranda expressions? (If the expression is followed by :: give its type.) If there is a compile or run time error, describe what the error is. (**Answers on line following expression.**)

   (a) `double ::`
   ```
   num->num
   ```

   (b) `my_const ::`
   ```
   *->**->*
   ```

   (c) `my_const double ::`
   ```
   *->num->num
   ```

   (d) `fmap ::`
   ```
   [*->**]->[*]->[**]
   ```

   (e) `manyconsts ::`
   ```
   [*->num]
   ```

   (f) `fmap manyconsts ::`
   ```
   [*]->[num]
   ```

   (g) `fmap double [1,2,3] ::`
   ```
   type error
   ```

(h)
```
fmap [double, square,abs] [1,2,3]
  [2,4,3]
```

(i)
```
mystery ::
  (*->*)->*->[*]
```

(j)
```
mystery double 1
   [2,4,8,16,32,64,128,256, ...
```

2. Suppose that we define a class `Animal` in Java, with a subclass `SeaCreature`. `SeaCreature` in turn has a subclass `Squid`. `Animal` and `SeaCreature` define a method `whenAttacked`, and `Animal` and `Squid` define a `startle` method.

```
public class Animal {
  public void whenAttacked() {
    this.startle();
    System.out.println("flee!!");
  }
  public void startle() {
    System.out.println("eek ...");
  }
}


public class SeaCreature extends Animal {
  public void whenAttacked() {
    System.out.println("swim away!");
    super.whenAttacked();
  }
}

public class Squid extends SeaCreature {
  public void startle() {
    System.out.println("how about a cloud of ink ...");
  }
}
```

What is printed when we invoke `whenAttacked` on an instance of Squid? On an instance of SeaCreature? On an instance of Animal?

Answer:

```
Squid whenAttacked:
  swim away!
  how about a cloud of ink ...
  flee!!

SeaCreature whenAttacked:
  swim away!
  eek ...
  flee!!

Animal whenAtacked:
  eek ...
  flee!!
```

3. Java does not allow two different methods with the same name and argument types but different return types. Consider changing the language to allow such overloading on the return type. Would there ever be ambiguous expressions? If so, give an example. Could the ambiguity be detected at compile time or only at run time? How could it be disambiguated?

   Also discuss the implications for progam maintainability. (Hint: are there Java programs that were correct but become incorrect after some modification?) Please write your answer on the back of this page.

   Answer: Yes, there would be ambigous expressions. Consider:

```
class Aquarium {
  Fish contents() {
    return new Fish();
  }
}

class SaltWalterAquarium {
  Octopus contents() {
    return new Octopus();
  }
}

class Biologist
{
    void observe (Fish f) {
      System.out.println("spotted a fish");
    }

    void observe (Octopus o) {
      System.out.println("spotted an octopus");
    }
}
```

   If we evaluate

```
Aquarium a;
Biologist b;
a = new SaltWaterAquarium();
b = new Biologist();
b.observe(a.contents());
```

   Java doesn't know which version of `observe` to use, since `a.contents` could be either a Fish or an Octopus. This possible ambiguity could be detected at compile time, at least if we didn't have dynamically loaded classes. We could require that the programmer add a cast to disambiguate:

```
b.observe( (Octopus) a.contents() );
```

   versus

```
b.observe( (Fish) a.contents() );
```

   This change could have an adverse affect on program maintainability, since adding a method to a class might make existing expressions ambiguous. In the absence of dynamically loaded classes we could still detect this at compile time, so it seems like a minor problem. The larger problem is the ambiguous meanings of expressions like that above.

If we allow dynamically loaded classes, then we may not be able to detect such ambiguities at compile time, since the compiler won't have complete knowledge of the classes. This seems really unpleasant. (We didn't insist that your answer include information about the dynamic loading case.)

(As an aside — not expected as part of your answer — Ada does allow overloading on return types, and has rules regarding inserting qualifiers to disambiguate in such cases. So this is sometimes done in programming languages.)

4. Suppose we have a type `Fish` with a subtype `Salmon`. We also have a type `Fisher` that has a `catch` method with an argument of type `Fish`, and another type `SalmonFisher` that defines a `catch` method with an argument of type `Salmon`.

   We might sketch these classes in a Java-like language as follows:

```
class Fish
{ ...
}

class Salmon extends Fish
{ ...
}

class Fisher
{
    void catch(Fish f)
    { ...
    }
}

class SalmonFisher
{
    void catch(Salmon s)
    { ...
    }
}
```

   However, we aren't considering Java here, but rather languages that implement the contravariant or the covariant subtyping rules.

   What is the subtyping relation if any between `Fisher` and `SalmonFisher` under the contravariant typing rule?

   Answer: `Fisher` is a subtype of `SalmonFisher` (somewhat counter-intuitively, due to the inversion of the typing relation for the argument to `catch`).

   What is the subtyping relation if any between `Fisher` and `SalmonFisher` under the covariant typing rule?

   Answer: `SalmonFisher` is a subtype of `Fisher`. This seems more reasonable, but programs that type check correctly can have runtime type errors. For example:

```
Fisher f;
f = new SalmonFisher();
f.catch(new Fish());
```

   The problem is that `f` is declared as a `Fisher`, and so `catch` should be happy with a fish. But really `f` a `SalmonFisher` who doesn't want just any fish.

If one of the rules gives an incorrect answer, give an example of a program that is statically correct but that has a runtime type error; or one that will always execute without type errors, even though the rule says it is incorrect.

Answer: See above.

5. In Pizza we might define an interface `Sortable` that specifies a method `min` for finding the minimum of two values. The `min` method is defined for a sortable object, takes an object of the same type, and returns an object (yet again) of that type. Given this interface, we could then define a class `MyInteger` that implements it, and another class `MyString` that implements it as well. So we should be able to find the min of two MyInteger objects, and the min of two MyString objects; but not the min of a MyInteger and a MyString.

   This code should type check correctly:

```
MyInteger i,j,k;
MyString r,s,t;
...
k = i.min(j);
t = r.min(s);
```

   We might try defining this interface as follows:

```
interface Sortable {
  Sortable min(Sortable x)
}
```

   However, this isn't right. Give an example of a statement that type-checks correctly but that shouldn't. Now give a correct definition for `Sortable`.

   Answer: Here is a statement that type checks correctly but that shouldn't (since we're trying to find the min of an integer and a string):

```
MyInteger i;
MyString s;
Sortable x;
...
x = i.min(s);
```

   In addition, the earlier code doesn't type check, even though it should, since we can't assign a `Sortable` (returned from `min`) to either a `MyInteger` or a `MyString`.

   Here is a correct definition:

```
interface Sortable<elem> {
  elem min(elem x)
}
```

6. Smalltalk has a pure object model, while Java uses a hybrid model. What are the advantages and disadvantages of each approach?

   Answer: In Smalltalk everything is an object: windows, points, and also simple types such as integers and booleans. In Java there are 8 primitive types (int, bool, etc) which are *not* objects; everything else *is* an object. Wrapper classes (`Integer`, `Boolean`, etc.) can be used to wrap primitive types so that they can be used as real objects.

   The advantage of the pure object model is uniformity. There are fewer concepts for students to learn, and fewer exceptions. As an example, in Smalltalk one can insert integers and booleans into a collection

just like any other object; in Java collections hold objects but not primitive types, and so primitives must be wrapped.

The advatage of the hybrid model is efficiency. Primitive types can be (easily) stored more efficiently, and we know they don't need to be garbage collected. The compiler knows that an expression like `i+j` can be compiled using integer addition if `i` and `j` are declared as ints. (Clever compiler technology can address many of these efficiency issues — and hopefully all, in the long term.)

7. Consider the following Smalltalk class definition.

```
Object subclass: #Octopus
   instanceVariableNames: 'myblock'
   classVariableNames: ''
   poolDictionaries: ''

setValue: x
   myblock := [x].

setBlock: y
   myblock := y.

getValue
  ^ myblock value
```

What is the result of evaluating the following code? (The value in each case will be the value of the last `o getValue` expression.)

(a)
```
| o a |
a := 3.
o := Octopus new.
o setValue: a.
a := a+1.
o getValue
```

Answer: 3

(b)
```
| o a b |
a := 3.
b := [a].
o := Octopus new.
o setBlock: b.
a := a+1.
o getValue
```

Answer: 4

8. Suppose we modify CLP($\mathcal{R}$) by adding static type declarations. Are there programs that used to work correctly before adding type checking that now fail during the static type checking phase? Are there programs that used to fail at runtime (perhaps due to a type error) that now fail static type checking? Give examples.

Answer: CLP($\mathcal{R}$) includes lists and similar structures, so we need to accomodate lists of floats, lists of atoms, etc. Suppose that we use Miranda-style universal polymorphism. In that case a program that searched a list containing mixed floats and atoms:

```
?- member(3,[3,4,5,fred])
```

would no longer make it past the type checker, even though this works fine in CLP($\mathcal{R}$) currently.

There are also programs that used to fail at runtime due to type errors that will now have the problem detected at compile time (probably a good thing). For example:

```
?- X=a, Y=X+3.
```

would formerly fail at runtime due to a type error, which should now be detected at compile time.

9. Suppose that in some application of Cassowary it is important to be able to change the weight and/or strength of a constraint, and this operation should be fast and incremental. For example, we might want to change a `stay` constraint from weak to strong. In Cassowary as currently implemented this would need to be done by deleting the weak stay constraint and adding a strong stay constraint. Could this be done more efficiently? If so, how?

   Hints: Consider the case both of changing to or from a required to a non-required constraint, and also the case of changing the strength or weight of a constraint that is non-required both before and after the change. My answer to this question is quite short — if you find yourself describing an elaborate algorithm extension something has probably gone amiss.

   Answer: This was a hard question, and we gave partial credit for all reasonable answers.

   First consider equality constraints.

   Consider changing the strength of a non-required equality constraint to another non-required strength. A non-required constraint C is represented in the tableau by an equation that includes error variables δ^+ and δ^-. Suppose C has symbolic weight w. Then the objective function includes $w\delta^+ + w\delta^-$. If the new symbolic weight for C is w', then subtract $w\delta^+ + w\delta^-$ from the objective function, and add $w'\delta^+ + w'\delta^-$. Re-optimize if necessary. (Note that in adding a term $w\delta$ to the objective function, δ should be a parametric variable. If it is basic, find the right hand side r of its equation in the tableau, and add wr to the objective function instead.)

   For ther other cases of changing a constraint to or from required, it isn't clear that we can improve that much over deleting and re-adding the constraints (and if you just said that, that was enough). But here are techniques for doing this.

   To change a non-required constraint C to a required one, try to make the values of its error variables be 0 (pivoting if necessary) and drop them from the tableau. To make the error variables take on value 0, first check if they already are 0 (meaning C was already satisfied). Otherwise, increase the strength of C to a new non-required strength that is stronger than any standard strength. Reoptimize. If the error variables are now 0, drop them. Otherwise signal an error that the required constraints cannot be satisfied.

   To change a required constraint C to a non-required one with weight w, add error variables δ^+ and δ^- for C to the tableau and to the objective function. Reoptimize if necessary. To add the error variables to the objective function, just add $w\delta^+ + w\delta^-$ to the objective function. To add the error variables to the rest of the tableau, we need to find which rows include multiples of C. To do this, look up the marker variable m for C. If m is basic, just add $\delta^+ - \delta^-$ to that row. If m is parameteric, find each row in which it occurs. Add $a\delta^+ - a\delta^-$ to each such row, where a is the coefficient of m in that row.

   Finally, if the original constraint C is an inequality, it will just have one error variable δ^+ or δ^- rather than two. Proceed as above, just using the one error variable.