

Computer Science & Engineering 505 – Final Exam – Answer Key

December 14, 2001

Open book & notes – 110 minutes – 10 points per question

100 points total

1. Consider the CLP(\mathcal{R}) queue problem from Assignment 3. The first question asked you to write predicates `put(Q0, I, Q1)` and `get(Q0, I, Q1)`. The `put` predicate puts an item `I` on a queue `Q0` to give `Q1`. The `get` predicate gets an item `I` from a queue `Q0` to give `Q1`. Here is the solution:

```
put(Q0, I, Q1) :- append(Q0, [I], Q1).
get([I|Q1], I, Q1).
```

```
/* convenience rule to make an empty queue, or test if a queue is empty */
empty([]).
```

```
/* reference definition of append */
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

If we try the goal

```
?- empty(Q0), put(Q0, 100, Q1).
```

this will succeed with the answer `Q0=[], Q1=[100]`.

What will happen when we try the following goal? (Say whether it succeeds or fails, and if it succeeds, give the answer.)

```
empty(Q0), put(Q0, 100, Q1), put(Q1, 200, Q2), get(Q2, I, Q3).
```

This succeeds with the answer:

```
Q0 = []
Q1 = [100]
Q2 = [100, 200]
Q3 = [200]
I = 100
```

2. Now consider the difference list version of the queue rules.

```
/* queue(N,L,T) represents a queue of length N, where the list of elements
   is stored in L with T lopped off the end */
put_dl(queue(N,L,[I|T]), I, queue(N+1,L,T)).
get_dl(queue(N,[I|L],T), I, queue(N-1,L,T)) :- N>0.
```

```
/* convenience rule to build an empty queue, or test if a queue is empty */
empty(queue(0,T,T)).
```

What will happen when we try a similar goal for the difference list version?

```
empty(Q0), put_dl(Q0,100,Q1).
```

This succeeds with the answer:

```
Q0 = queue(0, [100 | _V], [100 | _V])
Q1 = queue(1, [100 | _V], _V)
```

where $_V$ is a CLPR-generated variable name

And what happens for this goal?

```
empty(Q0), put_dl(Q0,100,Q1), put_dl(Q1,200,Q2), get_dl(Q2,I,Q3).
```

This succeeds with the answer:

```
Q0 = queue(0, [100, 200 | _T], [100, 200 | _T])
Q1 = queue(1, [100, 200 | _T], [200 | _T])
Q2 = queue(2, [100, 200 | _T], _T)
Q3 = queue(1, [200 | _T], _T)
I = 100
```

where $_T$ is a CLPR-generated variable name

3. The Pizza paper describes two techniques for translating Pizza into Java. What are they, and what are their advantages and disadvantages? (Please be brief.)

The two techniques are heterogeneous translation and homogeneous translation. In heterogeneous translation, a separate copy of the code is produced for each instantiation of a type variable. The advantage is that the code is faster; the disadvantage is that there is more memory required for the translated code. In homogenous translation, there is just one piece of code. Variables have the Java type Object, and the Pizza pre-processor inserts casts to cast the objects as having the correct type. The advantage is that the code is more compact; the disadvantage is that it is slower.

4. Consider the following expressions in the lambda calculus with let-polymorphism. Give the principal type of each expression, or say why it cannot be given a type.

(a) $\lambda f. \lambda x. (f (f x))$
 $(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \alpha$

(b) $\text{let id} = \lambda x.x$
 $\text{in } ((\text{id id}) \text{true})$

Bool

(c) $\lambda \text{id} . ((\text{id id}) \text{true})$

This expression cannot be typed, because id is a formal parameter and is therefore treated monomorphically, but each occurrence of id in the body requires a different type. (The technical reason the algorithm fails is because typechecking the call (id id) will attempt to unify α (the assumed type of id) with $(\alpha \rightarrow \beta)$ (for some fresh β), which fails because of the occurs check.)

5. Consider the following Smalltalk class definition.

```

Object subclass: #BlockHolder
  instanceVariableNames: 'myblock'
  classVariableNames: ''
  poolDictionaries: ''

setBlock: b
  myblock := b

getValue
  ^ myblock value

```

What is the result of evaluating the following code? (The code evaluates without error. The value in each case will be the value of the last `s getValue` expression.)

```

(a)      | s |
        s := BlockHolder new.
        s setBlock: [3+4].
        s getValue

```

The result is 7.

```

(b)      | s a |
        a := 3.
        s := BlockHolder new.
        s setBlock: [a+10].
        a := a+1.
        s getValue

```

The result is 14. (Note that the block is a closure, and references the variable `a` in the scope in which the block is created. So when the block is evaluated by sending the `getValue` message to `s`, we use the current value of `a`, which is 4.)

6. Joe Mocha is defining an interface `Appendable` in `Pizza` that includes an `append` method. He then defines two classes, `MyString` and `MyList`, which both implement `Appendable`. He wants `Pizza`'s type system to allow a `MyString` to be appended to a `MyString`, and a `MyList` to be appended to a `MyList`, but not a `MyString` to a `MyList`, or a `MyList` to a `MyString`.

Here is his definition of `Appendable`:

```

interface Appendable {
  Appendable append(Appendable a);
}

```

What is wrong with this definition? What is a correct one?

Also write a definition for a class `MyString` that uses the revised definition of `Appendable`. (Just put ... in the body of the method — we only care about the header.)

The problem with the given definition is that it allows a `MyString` to be appended to a `MyList`, and vice versa. Further, all we know about the return type is that it is an `Appendable`.

A correct definition is

```

interface Appendable<t> {
  t append(t a);
}

```

Using this we can define the append method in MyString:

```
class MyString implements Appendable<MyString> {
    MyString append(MyString a);
}
```

7. Suppose that we have a Java class `Vehicle`, and another class `Bus` that extends `Vehicle`. Consider the following pieces of Java code. In each case, say whether the code runs correctly, gives a compile time error, or gives a run time error. If there is an error, identify the line on which it occurs.

(a) `Vehicle[] v = new Vehicle[10];`
`v[0] = new Bus();`
`v[1] = new Vehicle();`

This runs correctly.

(b) `Bus[] b = new Bus[10];`
`b[0] = new Vehicle();`
`b[1] = new Bus();`

This gives a compile-time error at the assignment `b[0] = new Vehicle()`, since the type of “new Vehicle” is `Vehicle`, but `b` is an array of `bus`.

(c) `Vehicle[] v = new Bus[10];`
`v[0] = new Vehicle();`
`v[1] = new Bus();`

This compiles, but gives a runtime error at the assignment `v[0] = new Vehicle()`. Java uses covariant typing for arrays, so that an array of `Bus` is considered as a subtype of array of `Vehicle`. So the initialization of `v` is correct. Further, the assignment `v[0] = new Vehicle()`; type checks. But then when we try to execute this and assign a new vehicle to `v[0]`, we get a run time error since the Java runtime environment keeps track of the fact that `v` is actually an array of `Bus`, and raises an exception.

8. Suppose we are writing a simulation of a local network. We have two flavors of packet: an ordinary type `Packet`, and a subtype `EncryptedPacket`. We also have a type `Ethercard` that has a `receive` method with an argument of type `Packet`, and another type `DecryptingEthercard` that defines a `receive` method with an argument of type `EncryptedPacket`.

We might sketch these classes as follows (using Java-like syntax):

```
class Packet
{
    ....
    String getHeader () { ... }
}

class EncryptedPacket
{
    ...
    String getHeader () { ... }
    int keylength () { ... }
}

class Ethercard
{
    void receive(Packet p)
    {
        ...
    }
}
```

```

        h = p.getHeader();
        ...
    }
}

class DecryptingEthercard
{
    void receive(EncryptedPacket p)
    {
        ...
        h = p.getHeader();
        k = p.keylength();
        ...
    }
}

```

Using structural subtyping rules (i.e. the ones given in class to define covariance and contravariance), clearly `EncryptedPacket` is a subtype of `Packet`.

- (a) What is the subtyping relation, if any, between `Ethercard` and `DecryptingEthercard` under the contravariant typing rule?
Somewhat counter-intuitively, `Ethercard` is a subtype of `DecryptingEthercard`. (Anywhere we expect a `DecryptingEthercard`, we can use an `Ethercard`.)
- (b) What is the subtyping relation if any between `Ethercard` and `DecryptingEthercard` under the covariant typing rule?

`DecryptingEthercard` is a subtype of `Ethercard` under the covariant rule — a much more intuitive result. Alas, though, this result is unsound. Suppose we have a variable `e` of type `Ethercard`. We can assign a `DecryptingEthercard` to `e`. Then suppose we evaluate `e.receive(p)` for an packet `p`. This type checks. However, at run time, we will try to find the `keylength` method for `p`, and get an error.

If one of the rules gave an unsound result, describe an example that is statically correct but that has a runtime type error. (Continue your answer on the other side if necessary.)

- 9. Suppose we're given that `Int` is a subtype of `Num`. Formally, we add the following rule to the structural subtyping rules given in class for the simply typed lambda calculus with records:

$$\frac{}{\text{Int} \leq \text{Num}} \text{ (S-IntNum)}$$

Show a derivation of the fact that $\{x:\text{Num} \rightarrow \text{Int}, y:\text{Int}\}$ is a subtype of $\{x:\text{Int} \rightarrow \text{Num}\}$.

$$\begin{array}{c}
 \frac{}{\text{Int} \leq \text{Num}} \quad \frac{}{\text{Int} \leq \text{Num}} \\
 \hline
 \text{Num} \rightarrow \text{Int} \leq \text{Int} \rightarrow \text{Num} \\
 \hline
 \frac{}{\{x:\text{Num} \rightarrow \text{Int}, y:\text{Int}\} \leq \{x:\text{Num} \rightarrow \text{Int}\}} \quad \frac{}{\{x:\text{Num} \rightarrow \text{Int}\} \leq \{x:\text{Int} \rightarrow \text{Num}\}} \\
 \hline
 \{x:\text{Num} \rightarrow \text{Int}, y:\text{Int}\} \leq \{x:\text{Int} \rightarrow \text{Num}\}
 \end{array}$$

- 10. I'm deciding whether to make my new programming language be statically typed or dynamically typed. What benefits would I get by choosing static typechecking over dynamic typechecking? What benefits would I get by choosing dynamic typechecking over static typechecking?

With static typechecking, we get compile time detection of (most) type errors, and hence a guarantee that certain kinds of errors cannot occur at run-time. (The qualifier “most” is because most statically typed languages will still do a bit of runtime type checking. For example, casts in Java introduce runtime type checks.) Further, the type declarations are useful documentation, which is checked for correctness whenever we compile. There is also the potential for generating more efficient code. With dynamic typechecking, we may have more rapid prototyping because we don’t need to insert type declarations and keep them updated. Further, we may gain some expressiveness, because some programs will run correctly that would still be conservatively rejected by the static type system.