Appendix A

Implementation of `Integer`

Here is an example implementation of `Integer`, Haskell's arbitrary precision integral value type: it is essentially a wrapper around Java's `BigInteger` class. The copious uses of underscores is explained in Section **??**.

```
import java.math.BigInteger;

public class _Integer extends Data {
    public BigInteger value;
    public static _Integer _make_Integer(BigInteger x) {
        _Integer i = new _Integer();
        i.value = x;
        return i;
    }
    public static _Integer _make_Integer(String x) {
        return _make_Integer(new BigInteger(x));
    }

    public static _Integer add(_Integer x, _Integer y) {
        return _make_Integer(x.value.add(y.value));
    }
    ... // Analogous functions for subtraction and multiplication

    public static boolean eq(_Integer x, _Integer y) { ... }

    public static String show(_Integer x) { ... }
}
```

The `_make_Integer(String)` function is used to construct `Integer` literals: it allows a Java `_Integer` object to be constructed from a Java string representation. For example, the bytecode that creates the Haskell literal `2` would load the string `"2"` from the constant pool then invoke the creation method:

```
ldc          210       // String 2
invokestatic 16        // Method
↪  tmp/_Integer._make_Integer:(Ljava/lang/String;)Ltmp/_Integer;
```

The `add`, `eq`, etc. methods are Java implementations of the functions required by Haskell's `Num`, `Eq` and `Show` typeclass instances for `Integer`. For 'builtin' types, the implementation of these typeclass functions need to be given in Java, as they cannot be expressed in Haskell. Section **??** on Hooks covers this aspect of code generation in more detail.

Appendix B

The `Function` class

```
import java.util.ArrayList;
import java.util.function.BiFunction;

public class Function extends HeapObject {
    private BiFunction<HeapObject[], HeapObject[], HeapObject> inner;
    private HeapObject[] freeVariables;
    private ArrayList<HeapObject> arguments;
    private int arity = 0;
    private HeapObject result = null;

    public Function(BiFunction<HeapObject[], HeapObject[], HeapObject> inner, int arity,
    ↪  HeapObject[] freeVariables) {
        this.inner = inner;
        this.arity = arity;
        this.freeVariables = freeVariables;
        arguments = new ArrayList<>();
    }

    @Override
    public HeapObject enter() {
        // Check if we've got a cached value
        if (result != null) {
            return result;
        }

        if (arguments.size() < arity) {
            return this;
        }
        else if (arguments.size() > arity) {
            try {
                Function fun = (Function)inner
                    .apply(arguments.subList(0, arity).toArray(new HeapObject[0]),
                    ↪  freeVariables)
                    .enter()
                    .clone();
                for (HeapObject arg : arguments.subList(arity, arguments.size()))
                    fun.addArgument(arg);
                result = fun.enter();
                return result;
            }
            catch (CloneNotSupportedException e) {
                throw new RuntimeException(e);
            }
        }
        else {
```

```
44              result = inner.apply(arguments.toArray(new HeapObject[0]),
                ↪  freeVariables).enter();
45              return result;
46          }
47      }
48
49      public void addArgument(HeapObject arg) {
50          arguments.add(arg);
51      }
52
53      @Override
54      public Object clone() throws CloneNotSupportedException {
55          Function f = (Function)super.clone();
56          f.inner = inner;
57          f.arity = arity;
58          f.freeVariables = freeVariables.clone();
59          f.arguments = new ArrayList<>(arguments);
60          return f;
61      }
62  }
```