

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 ??.

```
1 import java.math.BigInteger;
2
3 public class _Integer extends Data {
4     public BigInteger value;
5     public static _Integer _make_Integer(BigInteger x) {
6         _Integer i = new _Integer();
7         i.value = x;
8         return i;
9     }
10    public static _Integer _make_Integer(String x) {
11        return _make_Integer(new BigInteger(x));
12    }
13
14    public static _Integer add(_Integer x, _Integer y) {
15        return _make_Integer(x.value.add(y.value));
16    }
17    ... // Analogous functions for subtraction and multiplication
18
19    public static boolean eq(_Integer x, _Integer y) { ... }
20
21    public static String show(_Integer x) { ... }
22 }
```

The `_make_Integer(String)` function is used by the compiler 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:

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

```
1 import java.util.ArrayList;
2 import java.util.function.BiFunction;
3
4 public class Function extends HeapObject {
5     private BiFunction<HeapObject[], HeapObject[], HeapObject> inner;
6     private HeapObject[] freeVariables;
7     private ArrayList<HeapObject> arguments;
8     private int arity = 0;
9     private HeapObject result = null;
10
11     public Function(BiFunction<HeapObject[], HeapObject[], HeapObject> inner, int arity,
12     ↪ HeapObject[] freeVariables) {
13         this.inner = inner;
14         this.arity = arity;
15         this.freeVariables = freeVariables;
16         arguments = new ArrayList<>();
17     }
18
19     @Override
20     public HeapObject enter() {
21         // Check if we've got a cached value
22         if (result != null) {
23             return result;
24         }
25
26         if (arguments.size() < arity) {
27             return this;
28         }
29         else if (arguments.size() > arity) {
30             try {
31                 Function fun = (Function)inner
32                     .apply(arguments.subList(0, arity).toArray(new HeapObject[0]),
33                     ↪ freeVariables)
34                     .enter()
35                     .clone();
36                 for (HeapObject arg : arguments.subList(arity, arguments.size()))
37                     fun.addArgument(arg);
38                 result = fun.enter();
39                 return result;
40             }
41             catch (CloneNotSupportedException e) {
42                 throw new RuntimeException(e);
43             }
44         }
45     }
46 }
```

```

44         result = inner.apply(arguments.toArray(new HeapObject[0]),
45                               ↪ freeVariables).enter();
46         return result;
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 }

```

Appendix C

Repository Structure Diagrams

The top-level repository structure is within 4 directories:

```
./
├─ app
├─ benchmarks
├─ src
└─ test
```

C.1 compiler

```
src/
├─ Compiler.hs
├─ Backend/
│   ├── CodeGen.hs
│   ├── Deoverload.hs
│   ├── ILAANF.hs
│   ├── ILA.hs
│   └─ ILB.hs
├─ Optimisations/
│   ├── BindingDedupe.hs
│   ├── LetLifting.hs
│   └─ UnreachableCodeElim.hs
├─ Preprocessor/
│   └─ Renamer.hs
└─ Typechecker/
    └─ Typechecker.hs
```

C.2 compiler-exe

```
app/
└─ Main.hs
```

C.3 Tests

```
test/
├─ AlphaEqSpec.hs
├─ Backend/
│   └─ DeoverloadSpec.hs
│       └─ ILAANFSpec.hs
│           └─ ILASpec.hs
├─ Preprocessor/
│   └─ DependencySpec.hs
│       └─ RenamerSpec.hs
├─ Typechecker/
│   └─ TypecheckerSpec.hs
├─ Spec.hs
└─ WholeProgram.hs
```

C.4 Benchmarks

```
benchmarks/
├─ benchmark.py
├─ etabenchmark.py
├─ fregebenchmark.py
├─ fregec.jar
├─ javabenchmark.py
├─ jhaskellbenchmark.py
├─ jmhbenchmark.py
├─ Main_Template.java
├─ plot.py
├─ programs
│   └─ ackermann.eta
│   └─ ackermann.fr
│   └─ ackermann.hs
│   └─ factorial.eta
│   └─ ...
├─ results
│   └─ ...
├─ results.py
└─ runbenchmarks.py
```

C.5 hs-java

```
./
├── JVM/
│   ├── Assembler.hs
│   ├── Builder/
│   │   ├── Instructions.hs
│   │   └── Monad.hs
│   ├── Builder.hs
│   ├── ClassFile.hs
│   ├── Common.hs
│   ├── Converter.hs
│   ├── Dump.hs
│   └── Exceptions.hs
└── Java/
    └── ...
```