Jazy Backend

Levin Fritz

September 28, 2012

Contents

1 Introduction

The Jazy backend generates Java class files to allow Haskell programs to run on the Java Virtual Machine (JVM). The program representation in Core is transformed to Java bytecode, which is then run by a small runtime written in Java. This runtime implements, most importantly, lazy evaluation on the JVM and low-level support for the primitive data types, input/output, exceptions etc.

The Jazy backend consists of several parts:

- a representation of Java class files and support for serializing this representation to `.class` files (in `src/ehc/JVMClass`)
- the compiler driver for compilation with the Jazy backend (in `src/ehc/EHC/CompilePhase/CompileJVM.chs`)
- code generation for the Jazy backend: transforming from Core to Java bytecode (`src/ehc/Core/ToJazy.cag`)
- the runtime system, which consists of a library that supports lazy evaluation (in `src/jazy/uu/jazy/core/`) and runtime support for basic arithmetic, IO and so forth (in `src/jazy/uu/jazy/ehc/`).

Finally, the mapping from Haskell's to the JVM's primtive types, for example from `Int` to `int`, is specified in `src/ehc/Base/Builtin2.chs`.

2 Status

The Jazy backend is, at the moment, still incomplete. It implements the basics of Haskell (functions, numbers, lists, strings etc) and basic IO (opening and closing files, reading from and writing to files as well as reading from stdin and writing to stdout), but other IO operations such as accessing directory information (module `System.Directory`), network IO, and measuring CPU time (module `CPUTime`) are not yet supported. Exceptions are also not implemented.

3 IO in the Jazy backend

The implementation of IO in EHC's libraries is made up of several layers:

1. Haskell code implementing the interface defined in the Haskell standard
2. C code called through the Foreign Function Interface (FFI)
3. operating system libraries and system calls.

To implement IO in the Jazy backend, we have basically three options:

1. Replace all three layers with an implementation in Java. This has the disadvantage that we need to duplicate a large amount of existing Haskell code in Java, which means a lot of extra effort both for initial development and for maintainance. Therefore, we did not persue this any further.
2. Replace the second and third layer by re-implementing the foreign functions necessary, but not the Haskell functions using them, in Java. This means essentially emulating a Posix-like interface and it is what we decided to do in UHC (as of April 2010).
3. Use the Java Native Interface (JNI) to call the C code that's also used by the other backends. This keeps all three layers, but requires considerable glue code to make the FFI calls work in the Jazy backend.

4 Using JNI for IO

To use the JNI for the IO libraries, we need to do the following things:

- globally:
  - Emulate pointers in Java. We can't use real C pointers because JNI does not support "pinning" memory, and pointers become invalid after a native method call returns.
  - Compile the C code used to a dynamically-linked library.
  - Load the library at runtime, using `System.load(..)` or `System.loadLibrary(..)`.
- for each function:
  - Write a wrapper method in Java.
  - Declare a native method in Java.
  - Write a wrapper function in C that calls the C function doing the real work.

As an example, the following code would be needed for the `write` function. This assumes that there is a class `uu.jazy.ehc.NativeMethods` containing the native method declarations.

- in `src/jazy/uu/jazy/ehc/Prim.java`:

```
public static Object write(int fd, int buf, int count) {
    byte[] array = (byte[])pointers.dereference(buf);
    int result = nativeMethods.write(fd, array, count);
    return ioReturn(new Integer(result));
}
```

- in `src/jazy/uu/jazy/ehc/NativeMethods.java`:

```
public native int write(int fd, byte[] buf, int count);
```

- in C code:

```
JNIEXPORT jint JNICALL Java_uu_jazy_ehc_NativeMethods_write(
        JNIEnv *env, jobject obj, jint fd, jbyteArray buf, jint count) {
    jbyte *ptr = (*env)->GetPrimitiveArrayCritical(env, buf, 0);
    int result = write(fd, ptr, count);
    (*env)->ReleasePrimitiveArrayCritical(env, buf, ptr, 0);
    return result;
}
```

5 Emulating primitive functions in Java

When Haskell code is compiled with UHC, each foreign function call to a function in C is transformed to a call to a method in the class `uu.jazy.ehc.Prim`. Implementing IO in the Jazy backend thus means implementing those methods. However, there are also some aspects common to several of those methods:

- Pointers: The Haskell libraries use C pointers, represented as integers, to refer to memory allocated in C code. As there are no pointers in Java, there is a class `Pointers`, which is used to emulate them. It stores a number of objects in a list; the position of each object serves as a pointer to the object. Adding an object in the `Pointer` instance also ensures it will not be garbage-collected.
- Open files: C has a concept of an "open file", which is stored in a per-process "file descriptor table". Since this is not present in Java, there is a class `Handle` that represents an open file by wrapping a `File` object, an `InputStream` object and an `OutputStream` object. There is also a class `FileDescriptorTable` which stores `Handle` objects and assigns a small, non-negative integer to each of them.
- Error handling: Functions in the C standard library signal errors by storing an error code in the global variable `errno`. This variable is represented by the static field `errno` in class `Prim`. The `Errno` class defines constants that can be used to set the field.