Java Virtual Machine support in FPC

Michaël Van Canneyt

November 5, 2011

Abstract

FPC is a pascal compiler that creates native binaries: it outputs machine instructions for a certain CPU and operating system. Recently, the capability to output Java Bytecode was added to FPC, thus opening a wide rane ofnew possibilities.

1 Introduction

Free Pascal is a compiler, which means it converts source code to binary machine code, much as GCC does. This contrasts with programming languages such as Java, C# or python and Ruby. These languages take the source code and convert it to an intermediary byte code, which is then passed on to a special execution engine - usually all in one step.

The output of compilers such as GCC and FPC is executed directly by the system's CPU. This makes programs generally faster and less resource consuming; It also avoids the use of an extra runtime environment such as the Java Virtual Machine.

Despite the advantages of a compiled binary, there are cases when the use of a runtime environment is required or useful: For instance, to develop for the Android platform one must produce Java Bytecode, which is transformed for use in the Android runtime system (called Dalvik). It is also possible one wants to use third-party classes which runs only inside the Java Virtual Machine. The Java community has a huge number of classes which contain a wealth of routines.

Recent developments in Free Pascal make it possible to create code that can be compiled not only to native machine but also to Java Bytecode that can be run inside a Java Virtual machine, meaning that development for e.g. Android or an application server as Tomcat becomes a possibility.

2 Getting started

Information about the JVM (Java Virtual Machine) backend can be found in the Free Pascal wiki:

`http://wiki.freepascal.org/FPC_JVM`

It allows to download a ready-to use compiler for a select number of platforms. Instructions to compile the compiler for Java Bytecode can be found on

`http://wiki.freepascal.org/FPC_JVM/Building`

Some external tools (such as the Java SDK and an external Java assembler, called Jasmin) are needed in order to create the JVM backend. These tools are included in the archives with the pre-compiled compiler binaries. The compiler binary is called ppcjvm.

The compiler is not yet available as a regular release; it is still under development, and is still in a separate branch of the subversion repository. Nevertheless, the code is usable.

3 What will work

Most Object Pascal features are supported. That is, simple types, records, arrays, strings and of course classes are supported, as well as the basic pascal constructs: simple procedures, loops. The Java classes are available for use in your code. In fact, a small tool (`javapp`) exists which creates import units for Java class files: it can be used to import arbitrary Java class definitions in Pascal. A restricted form of resources are also available.

4 What will not work

There are 3 kinds of restrictions:

1. a virtual machine places severe restrictions on what can be done in code: random memory access (such as can be had with pointers) is not allowed. That means the use of pointers is restricted. The java bytecode engine has no mechanism for passing variables by reference (var/out) which means that any code using this mechanism is emulated, which may have side-effects.
2. some pascal constructs require helper routines in the system unit (writeln being an important one). These helper routines have not yet been created; In fact, most of the RTL units are not compilable by the JVM backend. They must be ported and emulated using Java classes. This, in turn, means that any advanced class library at the time of writing will not compile if it depends on the RTL routines.
3. The last restriction is that inline assembler is not supported.

The restrictions due to the missing RTL units will be lifted in time, as more and more units are ported to the Java runtime engine.

5 Hello, World!

Despite the current restrictions, it is perfectly possible to create a working program and have it run in the Java Virtual machine:

```
program hello;

uses jdk15;

begin
  jlsystem.fout.println('Hello, world !');
end.
```

In this code, the jdk15 unit imports some standard java classes, and one of these is used to write a friendly greeting on the console.

The program can be compiled using the ppcjvm compiler and run:

```
home: >ppcjvm /home/michael/FPC/jvmbackend/rtl/units/jvm-java/ -Sm hello.pp
Generated: ./hello.class
home: >java -cp /home/michael/FPC/jvmbackend/rtl/units/jvm-java/:. hello
Hello, world !
```

Note that the support units compiled by the JVM must be in the Java CLASSPATH.

6 Conclusion

Although a running version of Lazarus and its LCL are still far away, the first steps towards a working JVM have been succesfully set: After creating a working JVM bytecode compiler, getting a working JVM RTL is the second hurdle to be taken.