Using Quark with Standalone JARs

Joseph Wong

Using Quark with Standalone JARs

by Joseph Wong

Last modified: November 14, 2007

Introduction

There are a number of ways in which programs written in CAL can be deployed. One way to deploy such a program is to generate a *standalone JAR*, which may contain application and library classes.

A standalone JAR may package up a CAL application by gathering together all the classes necessary for running a specific CAL function, and a class whose `public static void main(String[] args)` method runs the CAL function with the supplied command-line arguments. This makes it possible to package up a CAL function into a standalone application.

The generated main class is able to run the CAL function directly without having to first initialize a CAL workspace. Thus, the start-up time to run the function will be much smaller than in cases where workspace initialization is required (e.g. running the function via `BasicCALServices` from within a Java program, or starting up ICE or the Gem Cutter and running the function.)

A standalone JAR may also package up one or more *library classes* - a library class is a class containing static methods corresponding to the non-private functions and data constructors defined in a particular CAL module. This makes it possible to expose CAL libraries in Java, by defining API modules in CAL (whose functions may include code for marshalling to/from foreign types), from which library classes are generated.

The Standalone JAR Tool is the command-line tool that builds standalone JARs. A standalone JAR produced by this tool includes only the classes necessary for running the specified CAL applications and supporting the specified CAL library modules, and thus does not take up more space than necessary. User resources are also bundled in the generated standalone JAR, so that the CAL functions may load and use localized resources (such as string properties files) via modules such as `Cal.Core.Resource` and `Cal.Utilities.StringProperties`.

The tool also supports the generation of a companion source zip file containing source files for all generated classes. One can use the zip file in an IDE to link the class files in the JAR to sources, or build a version of the JAR with debug information (e.g. line numbers) from the sources for debugging purposes. Moreover, the library classes will have proper Javadoc comments generated from CALDoc comments. Thus it is possible to use these source files with the **javadoc** tool to produce API documentation for the standalone libraries.

We currently support generating standalone applications for CAL functions with the type `[String] -> ()`. The command-line arguments will be marshalled into a CAL list of `String`s and passed to the function.

Unlike other deployment mechanisms (e.g. deploying with Car or Car-jar files), applications and libraries deployed as a standalone JAR does not have access to meta-programming capabilities, such as the ability to query for gems with a given type, the ability to add new modules to the workspace, and the ability to evaluate programmatically constructed CAL expressions.

Document Overview

The rest of this document is organized as follows. The section *Standalone Libraries* discusses the details of the creation of library classes from CAL modules. Then, the sections *Using the Standalone JAR Tool* and *Running Applications in a Standalone JAR* cover the usage of the **quarkc** and **quarklaunch** scripts to respectively build a standalone JAR and run applications in the JAR. Finally, the last two sections (*Example - Running the `SpectralNorm` Benchmark with a Standalone JAR* and *Example - Creating a Standalone Library JAR*) present ready-made examples on producing a standalone application JAR and a standalone library JAR.

Standalone Libraries

The standalone JAR facility enables one to turn any CAL module into a standalone library. A standalone JAR can package up one or more *library classes* - a library class is a final, non-instantiatable class containing static methods corresponding to the non-private functions and data constructors defined in a particular CAL module.

The generated library class and its API methods have the following characteristics:

- Each method takes as its last argument an `ExecutionContext` representing the context and environment in which the CAL evaluation will occur.
  - Execution contexts can be created via the various `makeExecutionContext` factory methods in the support class `org.openquark.cal.runtime.lecc.StandaloneRuntime`.

- Excluding the `ExecutionContext` argument, each method will have an argument for every actual argument of the corresponding CAL function or data constructor. This takes into account the actual arity of the function or data constructor, and not the number of lexically declared parameters.

- Class methods and functions whose types contain *type class constraints* are not handled, and are skipped - no API methods will be generated for them.

- For each function or data constructor that is visible and whose type has no type class constraints, there will be a generated API method where all the arguments (excluding the last `ExecutionContext` argument) will have the Java type `org.openquark.cal.runtime.CalValue`, which represents an opaque value in CAL that may be an unevaluated computation. The return type of the method will follow the Java type mapping scheme (see below).

  If the function or data constructor takes at least one argument, then an overloaded version of the API method will also be generated. The argument types of this variant, unlike the above, will also follow the Java type mapping scheme.
  - To help facilitate the use of the first variant in scenarios where there is a mix of potentially unevaluated computations held in `CalValues` and plain Java values, there are various `toCalValue` factory methods in the support class `StandaloneRuntime` for converting Java values into `CalValues`.

- *Java type mapping scheme*: the mapping of CAL types to Java types for method arguments and the return value is as follows:
  - If the CAL type is a foreign type, and the foreign type's implementation visibility is either `public` or `protected`, then the corresponding Java type will be the foreign implementation type. This includes the `Prelude` types `Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, and `Double`, which correspond to Java primitive types.
  - If the CAL type is the `Prelude` type `Boolean`, then the corresponding Java type will be the primitive type `boolean`.
  - *(For method return types only)* If the CAL type is the `Prelude` type `Unit`, then the corresponding Java return type will be `void`.
  - If the CAL type is a foreign type whose implementation type is not visible, or if the type is an algebraic type, then the corresponding Java type will be `org.openquark.cal.runtime.CalValue`.

- Each generated API method will have a Java scope that corresponds to the CAL scope of the associated function/data constructor (i.e. `protected` in CAL becomes `protected` in Java, `public` in CAL becomes `public` in Java).

- The generated library class will have proper Javadoc comments generated from the module's CALDoc comments. Thus it is possible to use the source file in the companion source zip file with the **javadoc** tool to produce API documentation for the library.

To export a Java API for a library written in CAL, one can define the desired API for the library in a separate module (whose functions may include code for marshalling to/from foreign types). Then, a standalone JAR can be produced which would include a library class generated from the API module.

A Simple Example

Given the following CAL module:

```
/**
 * This is a demo of some simple functions exposed via a
 * standalone library JAR.
 */
module Cal.Samples.SimpleLibrary;
import Cal.Core.Prelude using
    typeConstructor = Int, JList;
    ;
import Cal.Collections.List;

filterOutEvenInternal :: [Int] -> [Int];
filterOutEvenInternal list = List.filter Prelude.isOdd list;

/**
 * Filters out the even numbers from the given list.
 * @arg list a list of integers.
 * @return the filtered list.
 */
filterOutEven :: JList -> JList;
public filterOutEven list =
    List.outputList (filterOutEvenInternal (List.inputList list));

/** An infinite list of ones. Can be used with {@link take@}. */
ones :: [Int];
public ones = List.repeat 1;

/**
 * Returns the first {@code n@} elements of the given list in a new list.
 * @arg n the number of elements to take.
 * @arg list the list of integers.
 * @return a list of the requested elements.
 */
take :: Int -> [Int] -> JList;
public take n list = List.outputList (List.take n list);
```

The library class generated from the CAL module would look like this:

```
package org.openquark.cal.samples;

import java.util.List;
import org.openquark.cal.internal.runtime.lecc.RTData;
import org.openquark.cal.internal.runtime.lecc.RTExecutionContext;
import org.openquark.cal.internal.runtime.lecc.RTValue;
import org.openquark.cal.internal.runtime.lecc.StandaloneJarGeneratedCodeInfo;
import org.openquark.cal.runtime.CALExecutorException;
import org.openquark.cal.runtime.CalValue;
import org.openquark.cal.runtime.ExecutionContext;
import org.openquark.cal_Cal_Samples_SimpleLibrary.Filter_Out_Even;
import org.openquark.cal_Cal_Samples_SimpleLibrary.Ones;
import org.openquark.cal_Cal_Samples_SimpleLibrary.Take;

/**
 * This is a demo of some simple functions exposed via a
 * standalone library JAR.
 */
public final class SimpleLibrary {
    private SimpleLibrary() {
    }

    /**
     * Filters out the even numbers from the given list.
     * @param list (CAL type: {@code JList})
     *          a list of integers.
     * @return (CAL type: {@code JList})
     *          the filtered list.
     */
    public static List filterOutEven(final List list,
        final ExecutionContext executionContext) throws CALExecutorException {
        // implementation clipped
    }

    /**
     * Filters out the even numbers from the given list.
     * @param list (CAL type: {@code JList})
     *          a list of integers.
     * @return (CAL type: {@code JList})
     *          the filtered list.
     */
    public static List filterOutEven(final CalValue list,
        final ExecutionContext executionContext) throws CALExecutorException {
        // implementation clipped
    }

    /**
     * An infinite list of ones. Can be used with {@link #take take}.
     * @return (CAL type: {@code [Int]})
     */
    public static CalValue ones(final ExecutionContext executionContext)
        throws CALExecutorException {
        // implementation clipped
    }

    /**
     * Returns the first <code>n</code> elements of the given list in a new list.
     * @param n (CAL type: {@code Int})
     *          the number of elements to take.
     * @param list (CAL type: {@code [Int]})
     *          the list of integers.
     * @return (CAL type: {@code JList})
     *          a list of the requested elements.
     */
    public static List take(final int n, final CalValue list,
        final ExecutionContext executionContext) throws CALExecutorException {
        // implementation clipped
    }

    /**
     * Returns the first <code>n</code> elements of the given list in a new list.
     * @param n (CAL type: {@code Int})
     *          the number of elements to take.
     * @param list (CAL type: {@code [Int]})
     *          the list of integers.
     * @return (CAL type: {@code JList})
     *          a list of the requested elements.
     */
    public static List take(final CalValue n, final CalValue list,
        final ExecutionContext executionContext) throws CALExecutorException {
        // implementation clipped
    }

    // private configuration checking code clipped
}
```

Some salient points in this example are:

1. The method `ones` returns a `CalValue` because the CAL type of the corresponding function is the algebraic type `[Int]`.

2. To get a `java.util.List` to be returned, one must employ CAL marshalling functions such as `List.outputList`, as in the case of `filterOutEven` and `take`.

3. The definition of the CAL function `ones` (`List.repeat 1`) by itself has a type `Num a => [a]`, which has a type class constraint. To make the function exportable, the function's type is specialized by its type declaration to be the non-polymorphic type `[Int]`.

4. Both `filterOutEven` and `take` have two corresponding API methods generated for them: one that takes only `CalValue`s as arguments, and another that takes values of Java types such as `int` or `java.util.List`.

5. The module's CALDoc comments are transformed into appropriate Javadoc comments for documentation purposes.

An End-to-End Example: The Directed Graph Library

The `CAL_Samples` project in the Open Quark source distribution includes a sample standalone library based on the directed graph library written in CAL. It is an end-to-end sample that includes:

1. an API module written in CAL (`Cal.Samples.DirectedGraphLibrary`) that shows how one can expose an existing CAL library as a Java library with appropriate marshalling,

2. a Java wrapper class (`ImmutableDirectedGraph`) which provides an object-oriented interface on top of the standalone library, and

3. a simple demo app which uses the exposed functionality.

The Companion Source Zip File

The standalone JAR facility also supports the generation of a companion source zip file containing source files for all generated classes. One can use the zip file in an IDE to link the class files in the JAR to sources. Also, it is possible to use these source files with the **javadoc** tool to produce API documentation for the standalone libraries.

Note that the generated Java sources in the zip file may not correspond exactly to the bytecode in the JAR. However, the classes as defined in source are semantically identical to the classes in the JAR. This makes it easy to debug into the internal operation of a standalone application or library: simply build a version of the JAR with debug information (e.g. line numbers) from the unpacked sources, and debug with this newly created JAR rather than the one produced by the standalone JAR facility. Doing so would allow IDEs and other source-level debuggers to support single-stepping through the entire standalone application or library.

Using the Standalone JAR Tool

The binary distribution of Open Quark comes with a script for launching the Standalone JAR Tool: **quarkc.bat** for Windows, and **quarkc.sh** for Unix, Linux, and Mac platforms. The tool supports the packaging of multiple CAL applications and/or multiple CAL libraries into a single standalone JAR.

Synopsis

`(Windows)`

`quarkc` *`workspaceFileName`* `[-verbose] (-main` *`functionName mainClassName`* `| -lib` *`moduleName libClassScope libClassName`*`)+` *`outputJarName`* `[-src` *`outputSrcZipName`*`]`

`(Unix/Linux/Mac)`

`quarkc.sh` *`workspaceFileName`* `[-verbose] (-main` *`functionName mainClassName`* `| -lib` *`moduleName libClassScope libClassName`*`)+` *`outputJarName`* `[-src` *`outputSrcZipName`*`]`

Description

`workspaceFileName`
the name of the workspace declaration file (just the name, no paths are accepted). This file should be in a "`Workspace Declarations`" subdirectory of a directory on the classpath.

`-verbose`
an optional flag for displaying more detailed status information.

`-main`
specifies a CAL application to be included with the standalone JAR.

`functionName`
the fully-qualified name of the CAL function that is the main entry point for the application.

`mainClassName`
the fully-qualified name of the Java main class to be generated.

`-lib`
specifies a CAL library to be included with the standalone JAR.

`moduleName`
the fully-qualified name of the CAL library module.

`libClassScope`
the scope of the generated library class. Can be one of:

`public` or **`package`**.

`libClassName`
the fully-qualified name of the Java main class to be generated.

`outputJarName`
the name of the output JAR file (can specify a path).

-src ***outputSrcZipName***
(optional) specifies the name of the output zip file containing source files for the generated classes.

If additional JARs or classpath entries are required for the foreign types and functions defined in the workspace, they can be specified via a `QUARK_CP` environment variable.

For example:

```
(Windows)

set QUARK_CP=foo.jar;bar.jar
quarkc foobar.cws -main Foo.Bar foo.bar.MainClass -lib Foo.Baz public foo.Baz foobar.jar

(Unix/Linux/Mac, using sh/bash)

QUARK_CP=foo.jar:bar.jar ./quarkc.sh foobar.cws \
  -main Foo.Bar foo.bar.MainClass -lib Foo.Baz public foo.Baz foobar.jar
```

Running Applications in a Standalone JAR

To run a CAL application packaged as part of a standalone JAR, simply include the standalone JAR, the Quark Platform JARs, and any required external JARs on the classpath, and specify the name of the generated class as the one to run. Command-line arguments will be passed into the CAL function.

The binary distribution of Open Quark provides a launch script to facilitate the setup of the classpath: **quarklaunch.bat** for Windows, and **quarklaunch.sh** for Unix, Linux, and Mac platforms.

Synopsis

`(Windows)`

`quarklaunch` *`jarName mainClassName [args...]`*

`(Unix/Linux/Mac)`

`quarklaunch.sh` *`jarName mainClassName [args...]`*

Description

`jarName`
the name of the standalone JAR. If additional JARs are required, add them to the end, separated by the appropriate separator ('**`;`**' for `Windows`, '**`:`**' for `Unix/Linux/Mac`).

`mainClassName`
the name of the main class in the standalone JAR.

`args...`
0 or more arguments to be passed into the main class.

If additional Java VM arguments are required, they can be specified via a `QUARK_VMARGS` environment variable.

For example, to specify that Java should run with the server VM:

`(Windows)`

```
set QUARK_VMARGS=-server
quarklaunch foobar.jar foo.bar.MainClass
```

`(Unix/Linux/Mac, using sh/bash)`

```
QUARK_VMARGS=-server ./quarklaunch.sh foobar.jar foo.bar.MainClass
```

Things to note

A standalone JAR must be run with the same Quark system properties as those specified to the Standalone JAR Tool when it built the JAR. If the system properties specified when running the standalone JAR do not match those specified during the build operation, then running a standalone application in the JAR will display an error message (with hints on how to fix the issue) and exit. Similarly, library classes packaged in the JAR will not load successfully in such a case, as the static initializer in each library class would check the properties and throw an exception with the error message.

Example - Running the `SpectralNorm` Benchmark with a Standalone JAR

The Open Quark distribution contains implementation for many of the benchmarks included in the *Computer Language Benchmarks Game* (`http://shootout.alioth.debian.org`). We will demonstrate how to run one of the benchmarks, `Cal.Benchmarks.Shootout.SpectralNorm`, with a standalone JAR.

To build the standalone application JAR:

1. Launch a Command Prompt on Windows, or a terminal on Unix, Linux, or Mac platforms.

2. Change into the `Quark` directory inside the Open Quark binary distribution.

3. Run the Standalone JAR Tool as follows:

   ```
   (Windows)

   quarkc cal.benchmark.cws -main Cal.Benchmarks.Shootout.SpectralNorm.main \
     shootout.SpectralNorm spectralnorm.jar

   (Unix/Linux/Mac)

   ./quarkc.sh cal.benchmark.cws -main Cal.Benchmarks.Shootout.SpectralNorm.main \
     shootout.SpectralNorm spectralnorm.jar
   ```

   You should see the following output:

   ```
   Open Quark Standalone Jar Tool (version 1.7.0_0)
   Initializing the CAL workspace...
   ...CAL workspace initialized.
   Building the standalone jar D:\openquark\Quark\spectralnorm.jar
   Done building the standalone jar D:\openquark\Quark\spectralnorm.jar
   ```

Now that the standalone JAR is built, we can run the benchmark with it via the launch script **quarklaunch.bat** for Windows, or **quarklaunch.sh** for Unix, Linux, and Mac platforms. For this example, we will run the benchmark with the argument **`1500`**:

```
(Windows)

quarklaunch spectralnorm.jar shootout.SpectralNorm 1500

(Unix/Linux/Mac)

./quarklaunch.sh spectralnorm.jar shootout.SpectralNorm 1500
```

You should see the following output:

```
1.274224151
```

Example - Creating a Standalone Library JAR

The Open Quark distribution contains a ready-made sample standalone library that exposes a directed graph library written in CAL. We will demonstrate how to create the JAR using the Standalone JAR Tool.

To build the standalone library JAR:

1. Launch a Command Prompt on Windows, or a terminal on Unix, Linux, or Mac platforms.

2. Change into the `Quark` directory inside the Open Quark binary distribution.

3. Run the Standalone JAR Tool as follows:

   ```
   (Windows)

   quarkc cal.samples.cws -lib Cal.Samples.DirectedGraphLibrary \
     public org.openquark.cal.samples.directedgraph.DirectedGraphLibrary \
     directedgraph.jar -src directedgraph.src.zip

   (Unix/Linux/Mac)

   ./quarkc.sh cal.samples.cws -lib Cal.Samples.DirectedGraphLibrary \
     public org.openquark.cal.samples.directedgraph.DirectedGraphLibrary \
     directedgraph.jar -src directedgraph.src.zip
   ```

   You should see the following output:

   ```
   Open Quark Standalone Jar Tool (version 1.7.0_0)
   Initializing the CAL workspace...
   ...CAL workspace initialized.
   Building the standalone jar D:\openquark\Quark\directedgraph.jar
   Done building the standalone jar D:\openquark\Quark\directedgraph.jar
   ```

We can inspect the library classes generated by opening the source zip file `directedgraph.src.zip` and viewing the files inside. These files contain Javadoc comments generated from the source module's CALDoc comments, and can in fact be used with the **javadoc** tool to produce the standalone library's API documentation.

Now that the standalone JAR is built, we can run the directed graph demo with it via the launch script **quarklaunch.bat** for Windows, or **quarklaunch.sh** for Unix, Linux, and Mac platforms:

```
(Windows)

set QUARK_CP=.\samples\simple\
quarklaunch directedgraph.jar org.openquark.cal.samples.directedgraph.DirectedGraphDemo

(Unix/Linux/Mac)

./quarklaunch.sh directedgraph.jar:./samples/simple/ \
  org.openquark.cal.samples.directedgraph.DirectedGraphDemo
```