Input/Output Source Generation Tool

Contributors: Raymond Cypher
Last modified: August 2, 2007

Contents

1 Input/Output Source Generation Tool

When working on a project which combines CAL and Java code there is often a need to marshal data types between CAL and Java. The mechanism which is generally used for this is to make a CAL data type an instance of the Inputable and Outputable type classes.

To make a user defined type inputable and outputable, you would first decide what the Java representation of the Quark type will be. The `input` and `output` methods will convert between the Quark type and an appropriate instance of this Java class.
A common approach is to define a foreign type for the target Java class, write CAL functions that convert to and from that type, using foreign functions, then convert to or from `JObject` using a foreign function that merely casts between the target Java class and `Object`. For more information please see the two documents 'CAL User's Guide' and 'Java Meets Quark'.

The Input/Output code generation tool seeks to streamline the process of making a CAL data type an instance of the Inputable and Outputable type classes by generating the majority of the necessary CAL and Java code.

The source generation tool is still under development. However, at this point in time it has enough functionality to be useful, and is ready for feedback from a wider audience.

1.1 What the Tool Does

The code generation tool takes an existing CAL data type and will generate a corresponding Java class. It will then generate the necessary CAL code to refer to the generated Java class as a CAL foreign type, generate the CAL functions needed to convert between the original CAL data type and the foreign type, and generate the type class instance declarations to make the original data type an instance of the Inputable and Outputable type classes.

2 Using the Code Generation Tool

The I/O code generation is accessed through an extension of ICE (the Interactive CAL Environment). For more on ICE see the 'CAL User's Guide'. The extension is named IO_Generator_ICE and extends ICE by adding two new commands, gio and tm, related to generating I/O code.

When running IO_Generator_ICE the help for the two new commands can be seen by executing ':help utilities'.

2.1 The gio (generate input output) Command

The first new command is the 'gio' command, which generates I/O code.

:gio <module name> <target package> [-rd<root directory>] [excluded type constructors] [-v | -verbose]

The <module name> argument specifies the module for which I/O code is to be generated. I/O code for all data types in the module will be generated, unless a data type has been specifically excluded.

The <target package> argument specifies the package in which the generated Java classes will reside.

An optional argument [-rd<rood directory>] can be used to specify the location to write the generated code. If a root directory is specified a directory named 'gen' will be created under the root directory and the generated code will be placed in appropriate sub-directories of 'gen'. If no root directory is specified the tool will attempt to determine a root directory relative to the location of the CAL source file for the module.

The –v argument optionally sets verbose mode, which gives more feedback about the code generation process.

Additionally any data types in the module which should be excluded from code generation can be listed.

For example:

:gio Shapes com.mystuff.shapes –rdC:\myproject\src Triangle –v

This command will generated I/O code for all the data types in the Shapes module, except Triangle which has been excluded. The generated CAL and Java sources will be placed under the directory c:\myproject\src, and the generated Java classes will be in the package com.mystuff.shapes.

2.2 The tm (type mapping) Command

The second new command is the 'tm' command, which is used to define a type mapping from a CAL type to a Java type. (see the CAL to Java Mappings section) This mapping information is used by the code generation tool when dealing with references to CAL types which have a manual implementation of Inputable/Outputable.

For example:

:tm UniqueIdentifier.UniqueIdentifier org.openquark.cal.RefinedUniqueIdentifier

Specifies that the CAL type UniqueIdentifier in the module UniqueIdentifier inputs from and outputs to the Java type RefinedUniqueIdentifier. It would be necessary to run this command before generating I/O code if the one of the types for which code is being

generated has a field of type UniqueIdentifier, since UniqueIdentifier already has manually implemented Inputable and Outputable.

:tm –show, will show any existing mappings, and :tm –remove <CAL type name> can be used to remove a mapping.

If a large number of mappings need to be specified in order to generate I/O code for a module it makes sense to put the commands into a CAL script, rather than entering them individually.

2.3 Using the Generated Sources

Once the Java and CAL sources have been generated for a module they need to be integrated into a project.

For the Java classes this is a matter of making sure their location is included in the classpath. If the location of the generated classes is not already in the classpath the sources can be moved to a location in the classpath or the classpath can be updated.

The generated CAL source takes the form of a CAL module, where the name is the original module name with "_JavaIO" appended.

The generated CAL can be integrated in two ways. The simplest way is to copy the generated source into the original module. However, if the code is going to be regenerated frequently this can be something of a maintenance hassle. Alternatively the generated CAL module can be added to the CAL workspace. If this route is chosen the original module will need to be modified in two ways. The original module will need to declare the generated module as a friend module, also the data types and data constructors in the original module will need to be public or protected, so that the generated module can reference them.

3 CAL to Java Mappings

In order to generate correct I/O code the tool needs to know what Java class corresponds to each CAL data type. This information can automatically be determined for any CAL foreign types and for any CAL types for which I/O code is generated using the tool. For other CAL types referenced by the types for which I/O code is being generated a mapping must be provided. The tool has built in mappings for the most basic types; List, Tuple, Array, Maybe, Unit, etc. and for the CAL types which correspond to Java primitives; Int, Double, Long, etc. Other mappings must be set by the user.

If, for example, on of the data types in the module for which code is being generated refers to a CAL data type called Address in the module UserInfo, which already has a manually created implementation of Inputable and Outputable, a mapping would need to be created. If the Address data type inputs/outputs to/from the Java class com.mystuff.userinfo.Address the following command would need to be issued before generating the I/O sources.


```
:tm  UserInfo.Address   com.mystuff.userinfo.Address
```

The mapping will stay in effect until the end of the session in IO_Source_Generator_ICE, or until it is removed using ':tm –remove <CAL type name>'.

Because generating I/O code for a module may involve specifying several such mappings it is usually best to put these commands into an ICE script. This makes it much easier to regenerate the I/O code as the originating module is modified.

4 Generated Java Classes

The form of the generated Java class, or classes, for a CAL data type will vary depending on the nature of the data type.

4.1 Enumaration Types

For an enumeration CAL data type, (i.e. a data type where all the data constructors have zero fields), a single Java class will be generated which follows the *Typesafe Enum* pattern (see *Effective Java* item 21). The class will be named using the name of the data type with a '_' character appended.

4.2 Albebraic Types

For a non-enumeration data type an abstract Java class corresponding to the data type will be generated. It will be named using the name of the data type with a '_' character appended. This class will contain a static final inner class, extending the abstract class, for each data constructor in the data type.

Fields in the Java classes will correspond to fields in the data constructors and will have appropriate access methods (i.e. getFieldName(), etc). The type of the fields in the Java class should correspond to the Java type which the CAL type outputs to. If a mapping from CAL to Java has been defined (see section on CAL to Java mappings) that Java type will be used. For foreign types the underlying Java type will be used. If the corresponding Java type cannot be determined it is assumed that the fields CAL type will have I/O code generated by the tool and a class reference is created based on the code generation class naming conventions.

5 Generated CAL Code

The generated CAL code takes the form of a CAL module named by appending "_JavaIO" to the originating module name. This new module will contain the necessary CAL code to make each of the data types in the originating module an instance of Inputable and Outputable.

The generated CAL code for each data type breaks down into three parts:

1) The generated Java class is declared as a foreign type. The class constructors and field access methods are declared as foreign functions.
2) Making use of the foreign declarations input and output functions are defined. The fields of the data type are marshaled by applications of the type class methods Prelude.input and Prelude.output.
3) Type class instance declarations for Prelude.Inputable and Prelude.Outputable are declared. This maps Prelude.input and Prelude.output to the generated input and output functions, for this data type.

In addition various 'cast' functions are declared to cast between JObject, the CAL for type for Object, and the foreign types of the data type and its fields.