RDF Gears documentation

3 februari 2012

Eric Feliksik

1 RDF Gears plugin development

Third party programmers can implement new RGL functions in Java. First, we will discuss examples of how functions can be implemented. There are two ways to do this: either by extending the `SimplyTypedRGLFunction` or the `AtomicRGLFunction` class. Then we will discuss how such a function can be used in RDF Gears.

1.1 Function lifecycle

The workflow execution consists of two phases. The first phase is the workflow loading. The second phase is the workflow execution. For every processor in a workflow, an instance of the appropriate RGLFunction is created. The lifecycle of such an instance is as follows:

1. instantiation
2. initialization
3. typechecking
4. execution
5. termination

The first and the last step are JVM issues that do not need our attention: functions do not need an explicit constructor, and the garbage collection and/or program termination are of no concern now. We will now discuss the other phases: initialization, typechecking and execution phases. For these phases, appropriate methods must be implemented.

1.2 Simple functions

The simplest way to implement an RGL function in Java is by extending the `SimplyTypedRGLFunction` class. This approach is good if your function will always receive and return the same input types, and the output must be *null* if any of the input values is *null*.

The three steps that must be implemented for `SimplyTypedRGLFunction` are the following:

- instantiation: nothing to be done
- initialization: to be implemented by the method `inititialize(Map<String,String> config)`
- typechecking: to be implemented by the method `getOutputType(TypeRow inputTypeRow)`
- execution: to be implemented by the method `simpleExecute(ValueRow input)`
- termination: nothing to be done

We will discuss this lifecycle by an example function that calculates the Jaro Similarity of two input strings. The implementation is given in figure 1 (the implementation of the *jaro* method is omitted, for brevity).

```
 1 public class JaroSimilarityFunction extends SimplyTypedRGLFunction {
 2   public static final String INPUT_1 = "s1";
 3   public static final String INPUT_2 = "s2";
 4
 5   public void initialize(Map<String, String> config) {
 6     this.requireInputType(INPUT_1, RDFType.getInstance());
 7     this.requireInputType(INPUT_2, RDFType.getInstance());
 8   }
 9
10   public RGLType getOutputType() {
11     return RDFType.getInstance();
12   }
13
14   public RGLValue simpleExecute(ValueRow inputRow) {
15     RGLValue val1 = inputRow.get(INPUT_1);
16     RGLValue val2 = inputRow.get(INPUT_2);
17     if (!val1.isLiteral() || ! val2.isLiteral())
18       return ValueFactory.createNull("JaroSimilarity can only compare literals");
19
20     String str1 = val1.asLiteral().getValueString();
21     String str2 = val2.asLiteral().getValueString();
22     double d = jaro(str1, str2);
23     return ValueFactory.createLiteralDouble(d);
24   }
25
26   private static double jaro(String string1, String string2) { /* omitted */ }
27 }
```

Figure 1: The JaroSimilarity function

1.3 Initialization

In the initialization phase, the `initialize()` function is called. This function fulfills two purpose.

The first purpose of this method is the registration of function inputs and their expected types. This must happen for every function that has inputs (and most functions do, of course). In the example, two inputs are registered, both of type RDFValue. As another example, consider a function that has an input named "bag" that accepts a value of type **Bag**(T), for some element-type T. The output of the function is the size of the bag. The `initialize` method would call `requireInputType("bag", BagType.getInstance(new SuperTypePattern()))` to type the input port "bag" as a Bag containing elements of arbitrary type.

The second purpose of this method is the configuration of the function. Configuration parameters can be passed to the `config` map. For most SimplyTypedRGLFunction implementations, this will not be relevant and this method parameter should be ignored.

Remark 1. A `SimplyTypedRGLFunction` that does use the config parameter is the `SPARQLFunction` implementation. The ins and outs of RGL function configuration are explained in section 1.4.1.

1.3.1 Typechecking

The second call is your function is `getOutputType`(). It should just return the output type of the function, as in line 11. Note that the output type is fixed, once you implemented this function. If you want the output type to depend on the types of the input values, the SimplyTypedRGL-Functions is not the right choice.

In some circumstances it is desirable to have the output type depend on the configuration parameters set in the `initialize()` method. This is possible. As an example, see the `SPARQLFunction` implementation.

1.3.2 Execution

In the execution phase, the `simpleExecute()` function is called. It may be called multiple times, if its processor (or a workflow that uses that processor) is marked for iteration. In line 14 of figure

1 we an example implementation.

This method defines the semantics of the RGL function. For all implementations of SimplyTypedRGLFunction, it holds that the output is NULL if any of the input values is NULL[1]. This is guaranteed by the RGL engine, and thus the method implementation does not need to deal with NULL values. If any of the inputs of a SimplyTypedRGLFunction is NULL, then the `simpleExecute()` function is *not* called. Instead, the first found NULL value is propagated to the output of the processor.

Assuming that all values are non-*NULL* (although bags and records may still *contain* NULL values), the `simpleExecute(ValueRow inputRow)` method is called. For all input names that were registered with `requireInputType()`, the named input can be fetched from the inputRow with the `get()` call.

Note that this function expects literals, and that the typechecking system guarantees only that the inputs are of type RDFValue. For this reason, in order to be typesafe, the inputs must be checked with the `isLiteral()` call before they can be cast to a Literal with the `asLiteral()` method. [2]This section also describes how the ValueFactory can be used to create return values.

1.4 Advanced functions

Another way to implement RGL functions is by extending the `AtomicRGLFunction`. This approach gives all the possibilities that SimplyTypedRGLFunction does. In addition, it allows polymorphism and the possibility to deal with NULL values manually. This comes at the cost of a slightly more complex interface. It is worth to study the implementation of the RGL core operators, as they provide good examples for polymorphic functions. Let us take a look at the RecordProject function of figure 2. We will discuss how each lifecycle phase is implemented for this function.

1.4.1 Initialization

The registration of inputs does not require specifying a type. As line 7 shows, the `requireInput()` call just receives a name. It registers a function input (and thus a processor port). Later, in the typechecking phase, the required type of the port will be registered.

An unrelated thing that happens in this example, is that a configuration parameter is used (line 8) to determine what field should be projected. This allows the RecordProject function to be configured for the projection of any record field. The `Map<String,String>` parameter that is passed is build by using the `<config param="key">value</config>` elements from the RGL workflow XML document.

1.4.2 Typechecking

The function `getOutputType(TypeRow inputTypeRow)` receives a row over types. For all field names registered in the initialization phase, the inputTypeRow will contain a type. In lines 14-16, the the expected type is constructed. Note how a SuperTypePattern() is used to match any type for the record-field. Also note that in line 15, the `fieldName` variable that was set as a configuratin parameter is now used.

1.5 Implementing streaming bags

The RGL data transformation of an RGL function can be implemented using the RGL API. It provides a conceptual view over the RGL values and does not in any way require understanding of the engine optimizations. For functions that return a bag value, the user merely needs to provide a mechanism to iterate over this bag once. So the implementation does not actually *construct* the

[1]Note that this is about RGL NULL values. The Java *null* value can and may never occur in place of an RGL value.

[2]For more details on the API for RGL values, study the function documentation in the Java source (or the javadoc generated therefrom). Of particular interest are the classes ValueFactory, RGLValue, BagValue, RecordValue and the related FieldIndexMapFactory, LiteralValue, URIValue, GraphValue.

```java
public class RecordProject extends NNRCFunction {
  public static final String CONFIGKEY_PROJECTFIELD = "projectField";
  public static final String INPUT_NAME  = "record";
  private String fieldName; // the field we project

  public void initialize(Map<String, String> config) {
    this.requireInput(INPUT_NAME);
    this.fieldName = config.get(CONFIGKEY_PROJECTFIELD);
  }

  public RGLType getOutputType(TypeRow inputTypeRow) throws FunctionTypingException {
    RGLType actualInputType = inputTypeRow.get(INPUT_NAME);

    TypeRow typeRow = new TypeRow();
    typeRow.put(fieldName, new SuperTypePattern()); /* any type will do */
    RGLType requiredInputType = RecordType.getInstance(typeRow);

    if (actualInputType.isSubtypeOf(requiredInputType)){
      RGLType fieldType = ((RecordType)actualInputType).getFieldType(fieldName);
      assert(fieldType!=null);
      return fieldType;
    } else {
      throw new FunctionTypingException(INPUT_NAME, requiredInputType, actualInputType);
    }
  }

  public RGLValue execute(ValueRow input) {
    RGLValue r = input.get(INPUT_NAME);
    if (r.isNull()) return r; // propagate the error
    return r.asRecord().get(fieldName);
  }
}
```

Figure 2: The RecordProject function

bag, but does specify an iterator that defines it. The bag implementations are thus agnostic to the way they are materialized by RDF Gears for multiple iterations. This makes implementation of third-party functions simple and allows them to seamlessly integrate with the typechecking mechanism and to benefit from the engine optimizations.

1.6 Allowing dynamic function loading

When loading a workflow or when listing the available custom functions, the ServiceLoader class is used. If you distribute your custom function in a JAR file, a file `/META-INF/services/nl.tudelft.rdfgears.rgl.functi` must exist and it must contain a line the containing the full name (including package path) of your function. And example is `nl.tudelft.rdfgears.rgl.function.standard.BagSize` . You can use any package for your function, as long as it extends RGLFunction.

Note that the rdfgears-plugins package already contains this file and it will automatically be included in the jar built by `$ mvn package`.