OO jDREW Built-in Creation Guide

for OO jDREW Version 0.88

Table of Contents

Introduction and Basic Structure

A built-in relation implementation class should implement the `jdrew.oo.builtins.Builtin` interface, which has two public methods that must be implemented.

This first of these methods - `int getSymbol()` is used to get the integer code for the built-in relation's predicate symbol; this symbol should be added to the symbol table (using the `int SymbolTable.internSymbol(String)` static method). The int that is returned by the `SymbolTable.internSymbol(String)` method should be what is returned by this method.

It is required that the predicate symbol be added to the symbol table properly before registering the built-in with the reasoning engine as the integer code is required to register the bult-in properly. With an incorrect predicate code it is likely that the built-in will either not be called at all, or called incorrectly, causing reasoning to fail.

The second method that must be implemented is `DefiniteClause buidResult(Term);` this method is responsible for interpreting a call to a built-in relation and generating the appropriate response for the reasoning engine. The response that is appropriate depends on the built-in relation and the call to that relation. If the call to the relation should fail then the built-in relation should return null or a fact that will not unify with the call (returning null is generally preferred as it is more efficient); If this call to the relation should succeed then the built-in implementation should return a fact that will unify with the call to the built-in and will result in the appropriate variable bindings (if applicable).

Section 2 (Implementing a Built-In Relation) provides a guide to creating a new built-in relation, using OO jDREW's **greaterThan** built-in as an example; while section 3 (Using Custom Built-Ins) gives instructions on using a custom built-in in OO jDREW (both Top-Down and Bottom-Up).

Implementing a Built-In Relation

A class to implement a built-in relation can be broken into four sections; the class definition, the constructor, the `getSymbol()` method and the `buildResult(Term)` method. This section looks at each of these parts in more detail.

Class Definition

The main thing in the class definition for a built-in relation implementation is to implement the `jdrew.oo.builtins.Builtin` interface. Additionally, an instance variable to store the integer code for the predicate symbol is typically included; while this is not a

requirement as the integer code can be looked-up in the symbol table, it is more efficient to store the integer code and use that than to look-up the symbol every time it is required.

For example the class definition of the `jdrew.oo.builtins.GreaterThanBuiltin` class (implementation of the **greaterThan** built-in is bellow (method bodies are not included):

```
public class GreaterThanBuiltin implements Builtin {
    private int sym;

    public GreaterThanBuiltin() { ... }

    public int getSymbol() { ... }

    public DefiniteClause buildResult(Term t) { ... }
}
```

As you can see the class implements the Builtin interface and the two required methods; additionally it includes an instance variable to store the predicate symbol integer code.

Constructor

The first thing that should be done as part of the built-in implementation class is initialize the predicate symbol with the symbol table; optionally storing the integer code in your variable for the code. Once this is done any built-in specific initialization can be done; although this is not required for many built-ins.

In the constructor for the **greaterThan** built-in the only thing required is to initialize the predicate symbol.

```
public GreaterThanBuiltin() {
    sym = SymbolTable.internSymbol("greaterThan");
}
```

int getSymbol() method

This is generally a very simple method that simply returns the integer code for the predicate symbol; this method is required to successfully register the built-in relation with the reasoning engine.

```
public int getSymbol() {
  return this.sym;
}
```

DefiniteClause buildResult(Term) method

This is the main method for a built-in relation and is responsible for interpreting a built-in call and creating the appropriate response that can be interpreted by the reasoning engine.

The first step in this process is to get the user input and validate that input; the following methods instance variables of the Term class are useful for retrieving data about the built-in call:

- `getSymbol()` and `getSymbolString()`
  These methods get the integer symbol code and the symbol string for the term; in the case of a complex term or atom this is the constructor or relation symbol; for a plex this will always be the `SymbolTable.IPLEX` and `SymbolTable.PLEX` values; for a individual constant this is the integer code for the constant (or the string itself). In the case of a variable the symbol code is a negative number and the symbol string will be of the form ?Varx where x equal to `-(symbol code + 1)`. This can be used to test if parameters are variables (which can be used for output) or constant (input).

- `getRole()` and `getRoleString()`
  These methods get the integer code for the role and the role name for the term; in the case of an oid this is equal to `SymbolTable.IOID` and `SymbolTable.OID` respectively, in the case of a positional parameter this is equal to `SymbolTable.INOROLE` and **“”** respectively; for all slotted terms this is equal to the role code for that slot and the respective role/slot name.

- `getType()`
  This method gets the integer code for the type of the term; The following types are defined in the system by default (`Types.IINTEGER` “Integer”, `Types.IFLOAT` “Real”, `Types.ISTRING` “String”, `Types.ITHING` “Thing” and `Types.INOTHING` “Nothing”). The use of types can allow for the creation of generic built-in relations; for example the **greaterThan** built-in works across both number types (Integer and Real) as well as Strings and does the appropriate comparison based upon the types of the input.

- `isCTerm()`
  This method is used to check if the term is a complex term (or atom or plex); if the term is a complex term (or atom or plex) then `true` is returned; otherwise (variable or individual constant) `false` is returned.

- `subTerms[]`
  For non-simple terms (complex terms, plexes and atoms) this array contains the associated parameters, indexed from 0; for `exampl t.subTerms[0]` is the first parameter (in the normalized parameter list[1]) of the complex term.

```
public DefiniteClause buildResult(Term t) {
   if (t.getSymbol() != addsym) {
       return null;
   }
```

The first thing that is done in the implementation for **greaterThan** is to verify the predicate symbol of the call to the built-in is valid for this built-in; while this should always be true as the handler should not be invoked otherwise, it is still good to verify this incase of an erroneous invocation of the built-in handler. If the predicate symbol is not valid `null` is returned to indicate that the call was unsuccessful.

```
   if (t.subTerms.length != 3) {
       return null;
   }
```

The next thing that is done is to verify the number of parameters of the built-in call; while normally you would consider there to be 2 parameters for a greater than comparison there will actually be three parameters to the call of the built-in relation as all atoms (built-in calls are represented as atoms) must have an `oid`; and if it is omitted it will be automatically inserted by the system (although it may be omitted again in output). For example a call to the **greaterThan** built-in `greaterThan(2, 3)` is equivalent to `greaterThan(?^ 2, 3)` as a variable place holder for the oid will automatically be inserted by the parsers. If the number of parameters is incorrect for a call to the built-in then `null` is returned to indicate failure.

```
   Term p1 = t.subTerms[1].deepCopy();
   Term p2 = t.subTerms[2].deepCopy();
```

The next thing that is done is to get the parameters for the built-in call, in this case a copy of the parameters are created (using the deepCopy() method of the Term class) so that they can be used later in constructing the output clause if the comparison is successful.

```
   if (p1.getSymbol() < 0 || p2.getSymbol() < 0) {
       return null;
   }
```

---

[1] Parameters are normalized in the following order: oid, positional parameters in input order, positional rest parameter, slotted parameters in encounter order, slotted rest parameter.

Since the input parameters to a call to the **greaterThan** built-in should always be ground the next step is to check that the two input parameters are not variables. Since variables have a negative integer code this can be checked by comparing the integer code to `0`; if either of the parameters is less than `0` then the call to the built-in relation should fail and `null` is returned.

```
String p1s = p1.getSymbolString();
String p2s = p2.getSymbolString();
```

Our final step in validating and retrieving input is to get string representation of the input parameters; this can be achieved by calling the `getSymbolString()` method of the `Term` objects that represents the parameters. Once this is done the input can be interpreted and an appropriate response generated.

The next sections of code interpret the call to the built-in an determine the appropriate response to generate. Since the **greaterThan** built-in is generic and can handle both numeric and string comparisons the types of the input parameters are checked to determine the appropriate comparison.

```
if ((p1.getType() == Types.IFLOAT || p1.getType() == Types.IINTEGER) &&
    (p2.getType() == Types.IFLOAT || p2.getType() == Types.IINTEGER)) {
      double d1;
      double d2;
      try {
          d1 = Double.parseDouble(p1s);
          d2 = Double.parseDouble(p2s);
      } catch (Exception e) {
          return null;
      }
      if (d1 <= d2) {
          return null;
      }
}
```

In this block of code numerical comparisons are handled; first the `getType()` method of the `Term` objects is used to get the type codes; these codes are compared to the `Types.IFLOAT` and `Types.IINTEGER` values to determine if they are numeric. Once this is done parsing the string representations to a numerical type for the comparison (in this example double precision floating point numbers are used for all numerical comparisons) is attempted, if parsing is unsuccessful then `null` is returned to indicate failure. Once this is done the two parsed numbers are compared; if the first number is less than or equal to (i.e. not greater than) `null` is returned to indicate failure; otherwise processing will continue and a result clause will be generated.

```
else if (p1.getType() == Types.ISTRING &&
           p2.getType() == Types.ISTRING) {
    if (p1s.compareTo(p2s) <= 0) {
       return null;
    }
}
```

In this block of code string comparison is handled; the two strings are compared using Java's `String.compareTo(Object)` method. If the result is less than or equal to `0` then `null` is returned to indicate failure; otherwise processing continues and the appropriate response is generated.

```
else {
    return null;
}
```

If the types do not meet either of our cases then `null` is returned to indicate failure. This **greaterThan** implementation does not define a comparison for untyped values.

The final step in the process is to generate the appropriate response for the reasoning engine. In this example data structures to represent the resulting clause are manually created; it is possible to create a POSL (or RuleML) representation of the resulting fact and parse it using a parser to avoid dealing with manual data structure creation, although this would be less efficient.

```
Term roid = new Term(SymbolTable.internSymbol("$jdrew-gt-" + p1s + ">"
                                              + p2s),
                     SymbolTable.IOID, Types.ITHING);
```

Since all atoms in OO jDREW must have an oid (object identifier), an oid for the resulting fact (Atom) is generated here. A generated symbol (for the standard built-ins the following format is used: "`$jdrew`" followed by the built-in name (sometimes a short form such as "`gt`") followed by the input parameters) is used for the symbol, the oid role (`SymbolTable.IOID`) and the generic Thing type (`Types.ITHING`).

```
Vector v = new Vector();
v.add(roid);
v.add(p1);
v.add(p2);

Term atm = new Term(sym, SymbolTable.INOROLE, Types.ITHING, v);
atm.setAtom(true);
```

The next step is to create the Atom for the resulting fact. The first thing is to create a `Vector` that will hold the parameters for the atom, and then add the parameters to it; in this case the generated oid and copies of the two input parameters (these will always unify) are added (in the correct order). Then a `Term` object is created to represent the atom, the predicate symbol (stored in `sym`) is used as the symbol code, `SymbolTable.INOROLE` is used as the role code for all atoms, and `Types.ITHING` is used as the type code for all atoms, and the previously created vector is used as the parameter list. Finally, a call to `setAtom(boolean)`, passing true, is used to set the term as being an Atom.

```
    Vector v2 = new Vector();
    v2.add(atm);
    return new DefiniteClause(v2, new Vector());
}
```

The final step is to create the `DefiniteClause` object and return it to the caller (this is done automatically by the reasoning engine). A Vector is created to store the atoms of the clause and the previously created Term object is added to the vector. A `DefiniteClause` object is created using the Vector of term objects as the atom list and a new empty Vector as the variable name list (as this is a ground fact); the created `DefiniteClause` object is then returned to the caller.

Using Custom Built-Ins

Once a built-in relation has been created it must be registered with the reasoning engine before it can be used by rules. There are two methods which can be used to register a built-in: the first option is to include registration code in the application that uses the OO jDREW library, this is available to both the bottom-up and top-down engines; the second option is dynamic loading and registering of built-ins, this is only supported by the top-down engine.

Dynamic Loading and Registering of Built-in Relations

The top-down engine is capable (through the use of a system built-in) to dynamically load built-in implementations and register them in the engine. One limitation of the dynamic implementation is that the built-in relation must not require any parameters to be passed to it's constructor. In this case a built-in can be dynamically loaded and registered by calling the `registerBuiltin` built-in, passing the class name (including package) of the class that implements the built-in, either as a separate query or as part of a rule body, once `registerBuiltin` is called successfully then the built-in can be used as part of rules and queries.

For example, if you wanted to register the built-in implemented by the `TestBuiltin` class of the `jdrew.test` package (this is not a real class or package), then you would use the following call:

```
registerBuiltin("jdrew.test.TestBuiltin")
```

Notice that the complete package name is included, also since the symbol contains a POSL delimiter it must be contained in quotation marks ".

Registering Built-in Relations in Java Code

Built-ins can be registered with the reasoning engine in Java code that integrates OO jDREW libraries into Java applications (such as the Swing front-ends for the engines). Both the `ForwardReasoner` and `BackwardReasoner` objects have `registerBuiltin(Builtin)` methods that can be used to register a built-in with the reasoning engine. One advantage to this style is that built-in's that require parameters to the constructor can be used, as you pass the actual built-in object, not the name of the class.

To register a built-in on a created engine (either `ForwardReasoner` or `BackwardReasoner`), simple create an instance of the object that implements the built-in and pass it to the `registerBuiltin(Builtin)` method of the appropriate reasoning engine object.

For example, if you wanted to register the built-in implement by the `TestBuiltin` class of the `jdrew.test` package (this is not a real class or package), then you would use the following code in your Java application implementation:

```
import jdrew.test.TestBuiltin;
...
ForwardReasoner fr = ...;
...
TestBuiltin tb = new TestBuiltin();
fr.registerBuiltin(tb);
...
```