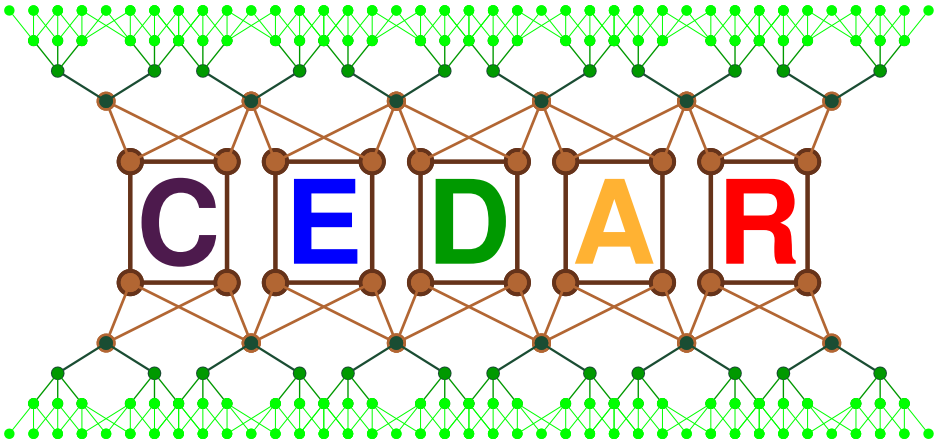


Technical Report Number 10

`Cedar.Gdl`

Java Library User Manual

Kevin Sancho

July 2014

Publication Note

This report is based on the work done by the first author during his internship in the $\mathcal{CEDAR}$ Project toward the obtention of his MSc degree at the Université Claude Bernard Lyon 1, on a topic proposed by Prof. Hassan Aït-Kaci [3, 2].

Contact information:

LIRIS - UFR d'Informatique
Université Claude Bernard Lyon 1
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France

Phone: +33 (0)4 27 46 57 08
Email: kevin.steven.sancho@gmail.com

$\mathcal{CEDAR}$ Project's Web Site: cedar.liris.cnrs.fr

This work was carried out as part of the $\mathcal{CEDAR}$ Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL).

CEDAR **Technical Report Number 10**

`Cedar.Gdl`

Java Library User Manual

Kevin Sancho

kevin.steven.sancho@gmail.com

July 2014

Abstract

This report is a user manual for the `Cedar.Gdl` Java library.

Keywords: Abstract Java Library, User Manual, `Cedar.Gdl`

Résumé

Ce rapport est un guide d'utilisateur de la bibliothèque Java `Cedar.Gdl`.

Mots-Clés: Bibliothèque abstraite Java, Guide d'utilisation, `Cedar.Gdl`

Table of Contents

This document is a step-by-step guide explaining how to use the `Cedar.Gdl` Java library [2]. It is assumed that the reader is familiar with the GDL and how it works [1]. It uses the Fast-Hadamard Transform (FHT) as an illustration.[1]

1 Brief Overview of the Library

This section overviews the contents of the Java packages that make up the `Cedar.Gdl` library.

To use it, the first step is to add the library `Cedar.Gdl.jar` to your project or your `JAVAPATH` java-executable path. The library is composed of several packages:

- `Abstract`—contains all the basic abstract classes described in Section 2. These are all the classes that one must extend in order to instantiate specific functions and operations.
- `Environment`—contains two files: `display` and `environment`, which list all the parameters.[2]
- `GDL`—contains all the methods that perform the GDL algorithm once a GDL problem is instantiated. The only method one needs to invoke is `GDL_Run()`.
- `implemented_xxx`—five packages have names matching this pattern. They contain instantiations of the abstract classes. For example, the usual addition and multiplication of the commutative semiring of the Rational Numbers,[3] functions for the fast Hadamard transform are already available. One can can directly invoke them in one's code or use them as template guides to follow for constructing one's own class.
- `Message`—contains all the classes that manage messages during the Message-Passing Algorithm. **N.B.***: These are not meant to be used nor modified.*
- `Util_graph`—contain all the classes related to graph construction and manipulation. **N.B.***: These are not meant to be used nor modified.*
- `Util_Implemented_Function`—examples of classes that support/enhance the implemented functions. Other possible features for these functions are described in detail.[4]
- `Util_Math`—contains all the classes that represent mathematical objects such as, for instance, local kernels and local domains.

2 Parameters

This section contains a list of important parameters which impact the output of the GDL. The following list is not complete. For a complete list please, please refer to the

[1] See [1] and [2].
[2] See Section 2.
[3] These are in fact numbers of type `float` or `double`.
[4] See Section 5.

Javadoc documentation.[5]

All these parameters are optional and have default values if omitted. Therefore, it is possible to use the library without using them. However, it is recommended to have at least a brief overview of the different possibilities to get an idea of what they may be used for.

The DISPLAY class defines:

- DISPLAY_MODE_ON: default value true. This field is used to show or hide the detail of the tree construction on the standard output.
- DISPLAY_RESULT_MODE_ON: default value true. This field is used to show or hide the result of the GDL message-passing algorithm. Even if this field is set to false it is still possible to use one's own own's specific class to display the output.[6]
- GRAPHIC_DISPLAY_MODE_ON: default value true. The Cedar.Gdl library uses a library called gs-core which supports graph displaying. The interesting point of this library is that it positions a graph's vertices on its own: it display graphs without the need to specify specific coordinates for all the vertices. This allows to display the local domain graph, junction tree, and all other kinds of graphs relevant to the GDL in a visual manner in addition to textual form. This parameter activates or deactivates such graph displays in small windowsq.

The ENVIRONMENT file contains:

- ENABLE_REAL_CALCULATION: default value true. This field allow to calculate or not the result of the message passing algorithm. If set to false the output of the message passing algorithm will only be the simplification done by the GDL problem. Therefore, the problem itself will not be calculated.
- ENABLE_MEMO_FUNCTION: default value true. Memo-functions are able to store previously calculated results with the purpose of never calculate twice a same problem. This can allow to spare a huge number of calculations and consequently time. However, they requires memory in order to store the results. This is a trade memory against calculations. Furthermore, the number of operations spared is not defined, it depend of the problem and of it simplification. For more detailed informations about Memo-functions, please refer to Section 6.
- SINGLE_VERTEX: default value false. This field is used to switch from the all vertices problem to the single vertex. If this field is set to true the field INDEX_VERTEX_SVP must be set with the id of one vertex. The Id of the vertex correspond to the Id of one local domain.

There are two others important fields in the class ENVIRONMENT, and these are listOfResultParameter and listResults. The first one is the list of result

[5] http://cedar.liris.cnrs.fr/documents/Cedar.Gdl-javadoc.html
[6] See Section 5.

```
Environment.ENVIRONMENT.setSINGLE_VERTEX(true);
Environment.ENVIRONMENT.setINDEX_VERTEX_SVP(5);
```

Figure 1: Environment and display settings for an example of the FHT.

Local Domain	**Local Kernel**
$\{y_1, y_2, y_3\}$	$f(y_1, y_2, y_3)$
$\{x_1, y_1\}$	$(-1)^{x_1 y_1}$
$\{x_2, y_2\}$	$(-1)^{x_2 y_2}$
$\{x_3, y_3\}$	$(-1)^{x_3 y_3}$
$\{x_1, x_2, x_3\}$	1

Table 1: Local domains/kernels for the FHT's GDL formulation

computed by the Message-Passing Algorithm (in the case where `ENABLE_REAL_CALCULATION` is set to `true`). This field is a list of list of `Objects`. Each list contain the results for a specific vertex. If the problem is set as a single vertex problem only one list would be created. The second field is a list of list of parameter. The two fields are linked because a same index in both list will correspond to a list of parameter and its result. These two field do not have to be used. In most cases, the standard output of the library is enough. However this leave anyone free to create is own output class. The Section 5 discuss some possibilities.

These two classes are here to allow an easy personalization of the library. It is not necessary to use them but they can be useful. In the case of the Fast-Hadamard Transform it is possible to begin the method as described in Figure 1. As can be seen on Figure 1, the number of parameters is low. So this does not take long because all the other parameters simply use their default values.

3 Local Kernel and Local Domain

This section describes how to specify a GDL problem consisting of local domains and local kernels, into the library. But let us first go through a quick refresher on the GDL formulation of the Fast-Hadamard-Transform.[7] The first thing to do is to create an instance of a commutative Semiring. In that case, the commutative semiring is the usual additions and multiplications in real. Because this set is already implemented in the package `implemented_Semiring` the only thing to do is to instantiate it and create a new GDL problem: If your looking for a commutative Semiring which is not implemented in the library please refer to the Section 4. Once this is done, the variables which will be needed must be instantiated. A class call `Variable`, in the package `Util_Math`, is already in the library. For the FHT we now need to create 6

[7] See Table 1.

```
Abstract_CommutativeSemiring semiRing = new
    UsualAdditionMultiplicationInR();
GDL gdl = new GDL(semiRing);
```

Figure 2: Instantiation of a new GDL problem

```
Abstract_SetK binarySet = new Binary();

Variable x1 = new Variable();
x1.setLabel("x1");
x1.setSetK(binarySet);
Variable x2 = new Variable();
x2.setLabel("x2");
x2.setSetK(binarySet);
Variable x3 = new Variable();
x3.setLabel("x3");
x3.setSetK(binarySet);
Variable y1 = new Variable();
y1.setLabel("y1");
y1.setSetK(binarySet);
Variable y2 = new Variable();
y2.setLabel("y2");
y2.setSetK(binarySet);
Variable y3 = new Variable();
y3.setLabel("y3");
y3.setSetK(binarySet);
```

Figure 3: Creation of the variables for the FHT

variables which takes value in the binary set. This can be done easily as describe in Figure 3.

The first step is the instantiation of sets. For the FHT, all the variables are taking values in the binary set. Consequently, only this set need to be instantiated. Then for each created variables a unique label and a reference to a set K must be defined(For Set creation please refer to Section 4.3). Once all the variables are created, one must not forget to set the number of variables used with the command:

```
gdl.setNbVariable(6);
```

This is important since the path to find a junction tree will depend on the number of variables.

We now have the variables needed to defines the local domains. The only thing left is to

K	$\langle +, 0 \rangle$	$\langle \times, 1 \rangle$
{true, false}	⟨OR, false⟩	⟨AND, true⟩

Table 2: The AND/OR semiring

have the functions required to create the local kernels. Again, two options are possible: either using an instantiated function of the library, or creating a new function.[8]

Figure 4 gives more detail on how to input the local domains and local kernels.

All the step described in Figure 4 must be done in order to create a association local domain, local kernel. In the case of the FHT, we still have to input the 4 others local domains and local kernels:

At this point, one now has created a complete instance of a GDL problem. Hence, the Cedar.Gdl library's GDL algorithm can now be invoked with the method call:

```
gdl.GDL_Run();
```

The section showed how to create a GDL instance. Section 4 explains how to create and instantiate one's own specific semiring classes, and use them with the library.

4 Creating a Commutative Semiring

This section is a step-by-step explanation of how to create a commutative semiring instance. We will use the example of how implement the conjunctive Boolean semiring described in Table 2.

An abstract commutative semiring structure must be instantiated by providing implemented (concrete) classes for two commutative monoids and one set K. Figure 7 displays the code for an "AND" Semiring.

4.1 Creating a commutative monoid

This section explains the construction of a commutative monoid. The creation of commutative monoid instances is part of that of commutative semiring instance. The code shown in Figures 8 and 9 is an example of doing so for the creation of concrete class "ANDMonoid" implementing abstract class Abstract_Co-mutativeMonoid.

The constructor is used to set the identity element and define a label for the monoid. The three other methods's purpose is to perform the monoid operation over different data structures. The method Abstract_Atom calculate(), is the most important. It creates a result object from two input objects. It is the method that is used during the message-passing algorithm. Its implementation is simple and relies on the class Abstract_Operation. It consists of instantiating a new object of type Abstract_Operation.

[8] Regarding the second case, please refer to Section 4.4.

```
//Creation of the first local domain and local kernel
LdAndLk LdAndLk1 = new LdAndLk();

//Creation of the function
Abstract_Function f1 = new AddThreeVar(y1, y2, y3);

//Create the local kernel
LocalKernel localKernel1 = new LocalKernel();

//set the function as local kernel
localKernel1.setFunction(f1);

//add the local kernel to the LdAndLk
LdAndLk1.setLocalKernel(localKernel1);

//Create a new local domain
LocalDomain localDomain1 = new LocalDomain();

//Add all the variable to the local domain
localDomain1.addVariable(y1);
localDomain1.addVariable(y2);
localDomain1.addVariable(y3);

//add the local domain to the LdAndLk
LdAndLk1.setLocalDomain(localDomain1);

//add an id
LdAndLk1.setIndentificator(1);

//input the created couple local domain / local kernel in
   the gdl problem
gdl.getListLocalDomainAndLocalKernel()
  .addOneLdAndLk(LdAndLk1);
```

Figure 4: Local domain kernel for the FHT

```
LdAndLk LdAndLk2 = new LdAndLk();
Abstract_Function f2 = new MinusOnePower(x1, y1);
LocalKernel localKernel2 = new LocalKernel();
localKernel2.setFunction(f2);
LdAndLk2.setLocalKernel(localKernel2);
LocalDomain localDomain2 = new LocalDomain();
localDomain2.addVariable(x1);
localDomain2.addVariable(y1);
LdAndLk2.setLocalDomain(localDomain2);
LdAndLk2.setIndentificator(2);
gdl.getListLocalDomainAndLocalKernel()
   .addOneLdAndLk(LdAndLk2);

LdAndLk LdAndLk3 = new LdAndLk();
Abstract_Function f3 = new MinusOnePower(x2, y2);
LocalKernel localKernel3 = new LocalKernel();
localKernel3.setFunction(f3);
LdAndLk3.setLocalKernel(localKernel3);
LocalDomain localDomain3 = new LocalDomain();
localDomain3.addVariable(x2);
localDomain3.addVariable(y2);
LdAndLk3.setLocalDomain(localDomain3);
LdAndLk3.setIndentificator(3);
gdl.getListLocalDomainAndLocalKernel()
   .addOneLdAndLk(LdAndLk3);
```

Figure 5: Instantiation of the Fast-Hadamard Transform—Part 1

```
LdAndLk LdAndLk4 = new LdAndLk();
Abstract_Function f4 = new MinusOnePower(x3, y3);
LocalKernel localKernel4 = new LocalKernel();
localKernel4.setFunction(f4);
LdAndLk4.setLocalKernel(localKernel4);
LocalDomain localDomain4 = new LocalDomain();
localDomain4.addVariable(x3);
localDomain4.addVariable(y3);
LdAndLk4.setLocalDomain(localDomain4);
LdAndLk4.setIndentificator(4);
gdl.getListLocalDomainAndLocalKernel()
   .addOneLdAndLk(LdAndLk4);

LdAndLk LdAndLk5 = new LdAndLk();
Abstract_Function f5 = new
    Implemented_Function.Number(1.0);
LocalKernel localKernel5 = new LocalKernel();
localKernel5.setFunction(f5);
LdAndLk5.setLocalKernel(localKernel5);
LocalDomain localDomain5 = new LocalDomain();
localDomain5.addVariable(x1);
localDomain5.addVariable(x2);
localDomain5.addVariable(x3);
LdAndLk5.setLocalDomain(localDomain5);
LdAndLk5.setIndentificator(5);
gdl.getListLocalDomainAndLocalKernel()
   .addOneLdAndLk(LdAndLk5);
```

Figure 6: Instantiation of the Fast-Hadamard Transform—ctd. Part 2

```
public class OR_AND_SemiRing extends
    Abstract_CommutativeSemiring {
   public OR_AND_SemiRing() {
      this.additionMonoid = new ORMonoid();
      this.multiplicationMonoid = new ANDMonoid();
      this.setK = new Implemented_SetK.Boolean();
      }
}
```

Figure 7: Example instantiation of a semiring

```
public class ANDMonoid extends Abstract_CommutativeMonoid {

  public ANDMonoid() {
     this.neutralElement = true;
     this.label = "AND";
  }

  @Override
  public Abstract_Atom calculate(Abstract_Atom atom1,
      Abstract_Atom atom2) {
     return (new AND(atom1, atom2));
  }

  @Override
  public Abstract_Atom calculateOnSet(List<Variable>
      variable, Abstract_Atom atom1) {
     return (new
        GenericMultiplicationOperationOnSet(variable,
        atom1));
  }
```

Figure 8: Code for the creation of an "AND" monoid—Part 1

Method `Abstract_Atom calculateOnSet()` is similar, but used for performing the operation over a set. Two generic methods offer this basic service for the semiring's two operations: `GenericMultiplicationOperationOnSet()` and `GenericAdditionOperationOnSet()`. These functions execute the operations defined for the current commutative monoid over a set of base elements. However, it is possible to override these defaults in one's own specific subclass. The last method, `operation()`, implements the operation on two objects and returns a result. It manages the (unevaluated) syntactic expression of the operation, as well as whether or not to evaluate this expression, and the value this calculation returns when performed.

4.2 Creating an operation

This section explains how to extend the `Abstract_Operation` class and create operators.

A monoid is an operation with some rules and a set. However in the library three cases have to be separated:

1. the operation over two `Abstract_Atoms`;
2. the operation over two `Objects`; and,
3. the operation over a set.

```
   @Override
   public Object operation(Object number1, Object number2) {
      if (number1==null) {
         if (number2==null) {
            return null;
         } else {
            return number2;
         }
      } else {
         if (number2==null) {
            return number1;
         } else {
            if ((Boolean)number1) {
               if ((Boolean)number2) {
                  return true;
               } else {
                  return false;
               }
            } else {
               if ((Boolean)number2) {
                  return false;
               } else {
                  return false;
               }
            }
         }
      }
   }
}
```

Figure 9: Code for the creation of an "AND" monoid—std. Part 2

An operation on Abstract_Atoms returns a new (unevaluated) Abstract_Atom constructed from the operation's arguments. An operation on Objects returns the evaluated result of the actually performing the operation's calculation. An operation over a set is used in order to apply an operation over a set of variables and return an (unevaluated) Abstract_Atom result. Similar to the first method, this one is just here to simplify the enumeration of the element in the variable's set.

Let us have a closer look at the operation on two Abstract_Atoms. It is necessary to implement three methods:

1. `Object calculate(List<Parameter> parameters)`, which is the logic behind the operation;
2. `returnVariableUsed()`, which returns the variables used; and,
3. `String toString()`, which returns a printable string form of the syntactic expression corresponding to the operation.

These methods were also in other classes seen previously. This is because the class Abstract_Operation also extends Abstract_Atom. This allows the library to perform nearly all the calculations with just four methods, which are present in all the classes. For some code example, see Figure 10.

The method that calculate the result of two Object do not need to be created as a specific class but can be directly written in the monoid class. For the last one there is two possibilities. Either you only need execute the operation that you have created over a set without further modifications. In such a case you can simply use the already created classes. However, if you want to optimize or do some modification you will have to create a second class extending Abstract_Operation. Figures 11, 12, and 13 give an example of instantiation that you can use as example. The only thing that changes is the `calculate` method, which must be able to use all the set elements of all the variables and compute the monoid operation on them.

4.3 Creating a set K

This section is about the creation of classes extending the Abstract_SetK class.

In the library Cedar.Gdl, variables have to be associated with a set. Every set must implement the abstract class Abstract_SetK and implement the method `getSetContent()`. This method's sole purpose is to return the contents of the set. Figure 14 shows an example of instantiation of a set.

4.4 Creating a function

The creation of functions is mandatory if you want to use the library Cedar.Gdl. Indeed, local kernels are represented by function, and even the identity element extends Abstract_Function. However, the creation of new functions is simpled. The first step is to extend the class Abstract_Function; then, implement the method:

```
public class AND extends Abstract_Operation {

   protected Abstract_Atom atom1;
   protected Abstract_Atom atom2;

   public AND(Abstract_Atom atom1, Abstract_Atom atom2) {
      this.atom1 = atom1;
      this.atom2 = atom2;}

   @Override
   public String toString() {
      return "{"+this.atom1.toString() + "} AND {" +
         this.atom2.toString()+"}";}

   @Override
   public Object calculate(List<Parameter> parameters) {
      Environment.ENVIRONMENT.incresseNbCalculus_Operation();
      boolean resultAtom1 =
          (boolean)this.atom1.calculate(parameters);

      if (resultAtom1) {
         boolean resultAtom2 =
             (boolean)this.atom2.calculate(parameters);
         if (resultAtom2) {
            return true;
         } else {
            return false;
         }
      } else {
         return false;
      }
   }

   @Override
   public List<Variable> returnVariableUsed(List<Variable>
       listVariablesUsed) {
      listVariablesUsed =
          atom1.returnVariableUsed(listVariablesUsed);
      listVariablesUsed =
          atom2.returnVariableUsed(listVariablesUsed);
      return listVariablesUsed;
   }
}
```

Figure 10: Code for the creation of an "AND" operation

```
@Override
public Object calculate(List<Parameter> parameters) {
   if (ENVIRONMENT.isENABLE_MEMO_FUNCTION()) {
      boolean foundResultAlreadyCalculated = false;
      boolean exit1 = false;
      int indexOldResult = 0;
      while (!foundResultAlreadyCalculated && indexOldResult <
          this.alreadyCalculated.size()) {
         List<Parameter> oneResult =
             this.alreadyCalculated.get(indexOldResult);
         int indexInResult = 0;
         exit1=false;
         while (!exit1 && indexInResult < oneResult.size()) {
            boolean okForThisParameter = false;
            int indexInParameters = 0;
            while (!okForThisParameter && indexInParameters <
                parameters.size())
               if (oneResult.get(indexInResult).equalLabelValue
                      (parameters.get(indexInParameters)))
                  okForThisParameter = true;
               else
                  indexInParameters++;
            if (okForThisParameter==false)
               exit1=true;
            indexInResult++;
         }
         if (exit1 == false)
            foundResultAlreadyCalculated = true;
         else
            indexOldResult++;
      }
      if (foundResultAlreadyCalculated)
         return this.alreadyCalculatedResults
                .get(indexOldResult);
   }

   Object result = null;
   List<List<Object>> listOfContentSet = new ArrayList<>();

   for (Variable oneAdditionvar: this.additionVariables)
      listOfContentSet.add(oneAdditionvar
                  .getSetK().getSetContent());

   int nbSet = listOfContentSet.size();
   List<Integer> listIndex = new ArrayList<>();

   for (List<Object> oneSet: listOfContentSet)
      listIndex.add(oneSet.size()-1);

   boolean allCalculationDone = false;
   boolean readyToCalculate;
   int currentIndex = nbSet-1;
```

Figure 11: Code for the creation of an operation over a set—Part 1

```
while (!allCalculationDone) { // outer while loop
   int numberOfLoops = 0;
   List<Parameter> listParameter = new ArrayList<>();
   String resultString = "OR => ";
   for (Integer oneIndex: listIndex) {
      Parameter newParameter = new Parameter();
      newParameter
        .setName(this.additionVariables.get(numberOfLoops));
      newParameter
        .setValue(listOfContentSet.get(numberOfLoops)
                            .get(oneIndex));
      listParameter.add(newParameter);
      numberOfLoops = numberOfLoops+1;
      resultString = resultString
                + newParameter.getName().getLabel() + ": "
                + newParameter.getValue().toString() + " || ";}
   listParameter.addAll(parameters);
    Object calculationResult =
        additionOn.calculate(listParameter);
    result = ENVIRONMENT.getSemiRing()
                   .doAdditionOperation(result,
                       calculationResult);
    if ((Boolean)result) {
       if (Environment.DISPLAY
                  .isDISPLAY_CONSTRAINTSOLVING_ON())
         System.out.println(resultString);
      allCalculationDone = true;
      readyToCalculate = true;
      } else
      readyToCalculate = false;
```

Figure 12: Code for the creation of an operation over a set—ctd. Part 2

```
      while (!readyToCalculate) {
         if (listIndex.get(currentIndex)==0) {
            if (currentIndex==0) {
               allCalculationDone=true;
               readyToCalculate = true;
            } else {
               listIndex.set(currentIndex,
                         listOfContentSet.get(currentIndex)
                                         .size()-1);
               currentIndex--;}
         } else {
            listIndex.set(currentIndex,
                listIndex.get(currentIndex)-1);
            currentIndex=nbSet-1;
            readyToCalculate = true;}
      }
   } // end of outer while loop

   if (Environment.ENVIRONMENT.isENABLE_MEMO_FUNCTION()) {
      List<Parameter> listSaveResult = new ArrayList<>();
      for (Parameter oneParameter: parameters)
         if (this.listVariablesNeeded
               .contains(oneParameter.getName()))
            listSaveResult.add(oneParameter);

      this.alreadyCalculated.add(listSaveResult);
      this.alreadyCalculatedResults.add(result);}
   return result;
}
```

Figure 13: Code for the creation of an operation over a set—ctd. Part 3

```
public class IntegerFromXToY extends Abstract_SetK {

    protected int from;
    protected int to;

    public IntegerFromXToY(int from, int to) {
       this.from = from;
       this.to = to;
    }

    @Override
    public List<Object> getSetContent() {
       List returnSetContent = new ArrayList();
       for (int i=from; i<=to;i++) {
         returnSetContent.add(Double.valueOf(i));
       }
       return returnSetContent;
    }
}
```

Figure 14: Code for the creation of an "AND" monoid

`functionImplementation(List<Parameter> parameters)`. The second method (`returnVariableUsed()`), returns the variables used. The purpose of this method is that every `Atom` can access all the variables that occur in it.

5 More Possibilities

This section discusses some possibilities offered by the `Cedar.Gdl` library. Although it has an abstract nature, one may consider the construction of still abstract though more specific classes. This allows one to enhance its capabilities as well as its performance. For example, it could be useful to have variables carry supplementaries informations. This is what happens for the N-Queens problem where variables are associated to a column in order to save calculations [2]. In such a case, a way to proceed is to create a subclass of the class `Variable`. This new class can implement new features and support special calculations—see Figure 15.

In Figure 15, one can see that the field `this.extended` is set to to "`true`." This field is here is order to specify when the class variable is extended or not. If this field is "`true`," this means that the variable can be cast into a more specific class. Of course, when extending the variable class you still have to provide a complete definition for the variable. One will also need to implement specific functions which will use the specific fields of the new class.[9]

[9] See Section 4.4 for functions creation.

```
public class FixedVariable extends Variable {
   private Object fixedValue;

   public FixedVariable() {
      this.extended = true;
   }

   public Object getFixedValue() {
      return fixedValue;
   }

   public void setFixedValue(Object fixedValue) {
      this.fixedValue = fixedValue;
   }
}
```

Figure 15: Code for the creation of a subclass of the class `Variable`

The creation of variables that implement specific fields is not the only feature. As explained in Section 1, there is a field meant to contain results, and the parameters related to it. In some cases (*i.e.*, the Fast-Hadamard Transform), the results provided by the library are not in "final" form. It may be necessary yet to transform further the provided results into a specific desired data structure for further consumption (if only reporting). In that case, it is up to the library's user to provide such display methods as appropriate to one's needs.

One is free to specify how to going from a result in graph form to a specific text output. Next, two classes are described that display the result for the N-Queens Problem.[10] These two classes use fields of the class `ENVIRONMENT` in order to get the output of the library. It is then possible to use them as one wishes.

6 Memo-Functions

This section is about so-called *memo-functions*:[11] what are they and how to use them in the `Cedar.Gdl` library.

A memo-function is a useful implementation concept that enables saving repeated computations. It stores previously calculated results in a local cache and so can return them without recomputing them if the function is again invoked on the same arguments. In this way, it "remembers" the old result and returns it. This feature is present in the already implemented examples of the GDL.

However, memo-functions trade memory for computation. Thus, although they are

[10] See Figure 16 and Figure 17.

[11] http://en.wikipedia.org/wiki/Memoization

```
public static void DisplayResultNQueenProblem() {
    int nbOfQueen = Environment.ENVIRONMENT.getNbVariable();
    List<List<List<Parameter>>> listOfListOfListOfParameter
                          = Environment
                            .ENVIRONMENT
                            .getListOfResultParameter();
    List<List<Object>> listOfListResults =
        Environment.ENVIRONMENT.getListResults();

    int indexList = 0;
    for (List<Object> oneListOfResult: listOfListResults) {
       int index = 0;
       for (Object oneResult: oneListOfResult) {
          if (oneResult.equals(true)) {
             List<Parameter> oneSolution
                          = (listOfListOfListOfParameter
                              .get(indexList))
                              .get(index);
             Chessboard chess
                          = new Chessboard(nbOfQueen,
                                           oneSolution);
             chess.setVisible(true);
          }
          index++;
       }
       indexList++;
    }
 }
```

Figure 16: Code for a custom result-display class—Part 1

```
public Chessboard(int size, List<Parameter> solution) {
    initComponents();
    List<List<JPanel>> chessboard = new ArrayList<>();
    this.setSize(350, 350);
    ((GridLayout) jPanel1.getLayout()).setRows(size);
    ((GridLayout) jPanel1.getLayout()).setColumns(size);
    for (int i=0;i<size;i++) {
       List<JPanel> oneRow = new ArrayList();
       for (int j=0;j<size;j++) {
          JPanel newJPanel = new JPanel();
          if (i%2==0) {
             if (j%2==0) {
                newJPanel.setBackground(Color.lightGray);
             } else {
                newJPanel.setBackground(Color.WHITE);
             }
          } else {
             if (j%2==0) {
                newJPanel.setBackground(Color.WHITE);
             } else {
                newJPanel.setBackground(Color.lightGray);
             }
          }
          jPanel1.add(newJPanel);
          oneRow.add(newJPanel);
       }
       chessboard.add(oneRow);
    }
    int y;
    int x;
    for (Parameter oneParam: solution) {
       y = ((VariableWithSupplementaryFixedField)oneParam
          .getName()).getColumn();
       x = (int)Math.round((Double)oneParam.getValue());
       chessboard
            .get(y-1)
            .get(x-1)
            .setBackground(Color.yellow);
    }
 }
```

Figure 17: Code for a custom result-display class—ctd. Part 2

natively implemented in `Cedar.Gdl`, one has the option to choose whether or not to use this feature. This choice must be made depending on memory/CPU resources, a problem's specific nature, and the topology of the junction tree. Consequently, the `Abstract_Atom` class has two fields:

- `List<Variable> listVariablesNeeded`, and
- `List<List<Parameter>> alreadyCalculated`.

Figure 18 gives an example of code implementing memo-functions.

Because the fields are in the `Abstract_Atom` class, one can use memo-functions everywhere in the library.

References

[1] Srinivas M. Aji and Robert J. McEliece. The generalized distributive law. *IEEE Transactions on Information Theory*, 46(2):325–343, March 2000. [Available online[12]].

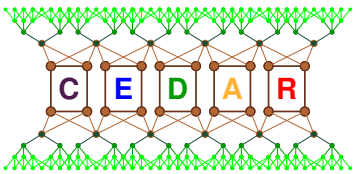
[2] Kevin Sancho and Hassan Aït-Kaci. The `Cedar.Gdl` Java Library for the Generalized Distributive Law—Design and Implementation. $\mathcal{CEDAR}$ Technical Report Number 9, Université Claude Bernard Lyon 1, Computer Science Department, Villeurbanne, France, July 2014. [Available online[13]].

[3] Kevin S. Sancho. The `Cedar.Gdl` Java Library for the Generalized Distributive Law. Master's thesis, Université Claude Bernard Lyon 1, Computer Science Department, Villeurbanne, France, June 2014.

[12] http://authors.library.caltech.edu/1541/1/AJIieeetit00.pdf
[13] http://cedar.liris.cnrs.fr/documents/ctr09.pdf

```
if (ENVIRONMENT.isENABLE_MEMO_FUNCTION()) {
   boolean foundResultAlreadyCalculated = false;
   boolean exit1 = false;
   int indexOldResult = 0;
   while (!foundResultAlreadyCalculated && indexOldResult
      < this.alreadyCalculated.size()) {
      List<Parameter> oneResult =
         this.alreadyCalculated.get(indexOldResult);
      int indexInResult = 0;
      exit1=false;
      while (!exit1 && indexInResult < oneResult.size()) {
         boolean okForThisParameter = false;
         int indexInParameters = 0;
         while (!okForThisParameter && indexInParameters <
            parameters.size()) {
            if
               (oneResult.get(indexInResult).equalLabelValue
                  (parameters.get(indexInParameters))) {
              okForThisParameter = true;
            } else {
               indexInParameters++;
            }
         }
         if (okForThisParameter==false) {
            exit1=true;
         }
         indexInResult++;
      }
      if (exit1 == false) {
         foundResultAlreadyCalculated = true;
      } else {
         indexOldResult++;
      }
   }
   if (foundResultAlreadyCalculated) {
      //System.out.println("SPARE CALCULATION for: " +
         this.toString());
      return
         this.alreadyCalculatedResults.get(indexOldResult);
   }
}
```

Figure 18: Code example for a memo-function



Technical Report Number 10

`Cedar.Gdl`

Kevin Sancho

July 2014