Declarative Programming in Java using JSetL

Elio Panegai and Gianfranco Rossi

Dipartimento di Matematica
Università di Parma, Parma (Italy)
panegai@cs.unipr.it, gianfranco.rossi@unipr.it

Abstract. JSetL is a Java library that endows Java with a number of facilities to support declarative programming like those usually found in constraint logic programming (CLP) languages. In this paper we show, through a number of simple examples, how these facilities can be exploited to support a declarative programming style in Java.

1 Introduction

While it is undeniable that declarative programming (DP) languages (such as Prolog and CLP languages) provide valuable support for programming in many application areas, it is also a reality that most real-world software development is still done using traditional, possibly object-oriented (OO), programming languages, such as C++ and Java.

Development of new programming languages that integrate features of both declarative programming languages and conventional and object-oriented languages is feasible (e.g., Mozart/Oz). This solution however requires a lot of effort and does not provide any guarantee that the new language is accepted: it is known that real-world programmers tend to be quite conservative.

An alternative approach is making DP features available as part of a *library* for some existing language. In this way one can exploit (at least, at some extent) the DP paradigm while retaining all the advantages of constructs for programming and software structuring that are typical of conventional programming languages. One of the first proposal that integrates DP features—specifically, constraints and logical variables—in a conventional OO framework following the *library-based approach* is the ILOG Solver [12, 9]. Other similar, more recent, proposals are, JSolver [2], Choco [1], and the Java Constraints Library (JCL) [10], all focusing on constraint programming. Frappè [3] and Gisela [8], instead, face the problem of declarative programming in a wider sense, though focusing on specific applications. Another proposal that can be cited in this context is `tu`Prolog [5], a Java package that implements Prolog.

In [13] we have presented a Java library—called JSetL—that endows Java with a number of facilities to support declarative programming like those usually found in constraint logic programming (CLP) languages. A detailed presentation of JSetL can be found in [11].[1] Differently from other related proposals which are concerned with only some specific concepts and mechanisms used to support declarative programming (in particular, constraint handling), JSetL aims at providing a comprehensive collection of facilities to support declarative programming. Moreover, our proposal aims to be general-purpose, not devoted to any specific application.

In this paper we describe, through a number of simple examples, how the new facilities provided by JSetL can be exploited to support a declarative programming style in Java. The notion of logical variable, along with unification and constraint solving, are fundamental tools to allow execution order to be immaterial, hence to support a declarative reading of programs. Constraint solving, in particular, allows complex (set) expressions to be build and checked for satisfiability disregarding their order and the instantiation of logical variables possibly occurring in them. Moreover, the use of partially specified data structures,

[1] JSetL is fully implemented in Java and is available at URL `www.math.unipr.it/~gianfr/JSetL`.

along with the nondeterminism naturally supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modeling tool.

In is important to note that the focus of our work is on declarative programming; we are deliberately assuming that execution efficiency is not a primary requirement. Indeed, JSetL is mainly conceived as a tool for rapid software prototyping, where easiness of program development and program understanding prevail over efficiency.

The paper is organized as follows. In Section 2 we introduce the use of logical variables and unification. In Section 3 we introduce the fundamental data structures of JSetL, namely sets and lists, and we show how to deal with partially specified lists and sets. In Section 4 we show how to exploit nondeterminism embedded in set-theoretical operations (in particular we present a solution for the well-known *traveling salesman problem*). In Section 5 we introduce constraint solving in JSetL (showing, in particular, a solution for the well-known *n-queens problem*). In Section 6 we show how to implement intensional set definitions in JSetL. In Section 7 we show how the user can implement nondeterministic methods by exploiting the facilities for new constraint definition provided by JSetL. Finally, in Section 8 we draw some conclusion and we briefly discuss future work.

2 Logical variables

The use of logical variables is at the heart of the declarative computation model (see, e.g., [14]). Logical variables are single-assignment variables. Initially, a logical variable is uninitialized. Once it has been bound to a value, the variable remains bound throughout the computation.

In JSetL logical variables are instances of the `Lvar` class. A value can be assigned to a (uninitialized) logical variable as the result of processing some constraint involving the variable, in particular, equality constraints. Values can be of any type.

In order to illustrate the programming style supported by the use of logical variables we show a simple example of a Java program using JSetL.

Example. Translate the name of the days of the week from Italian to English and vice versa.

```
class Translator {
   static SolverClass Solver = new SolverClass();

   public static void weekTranslate(Lvar g, Lvar d) {
       if (Solver.boolSolve(g.eq("lunedi").and(d.eq("monday"))))       return;
       if (Solver.boolSolve(g.eq("martedi").and(d.eq("tuesday"))))     return;
       if (Solver.boolSolve(g.eq("mercoledi").and(d.eq("wednesday")))) return;
       if (Solver.boolSolve(g.eq("giovedi").and(d.eq("thursday"))))    return;
       if (Solver.boolSolve(g.eq("venerdi").and(d.eq("friday"))))      return;
       if (Solver.boolSolve(g.eq("sabato").and(d.eq("saturday"))))     return;
       if (Solver.boolSolve(g.eq("domenica").and(d.eq("sunday"))))     return;
       throw new Failure;
   }
...
}
```

The equality operator `eq` implements unification (actually, *set unification* [7]). To solve equalities, as well as other operations on logical variables, one needs to invoke the `boolSolve` method over an instance of `SolverClass`. `Solver` is the instance of the `SolverClass` class that will be used in examples throughout the paper.

The use of logical variables and unification, in place of assignment and standard equality, allows the same methods to be used for different purposes. For example, if `answer` is an uninitialized logical variable, `lunedi` and `monday` are initialized logical variables, declared as follows:

```
Lvar answer = new Lvar();
Lvar lunedi = new Lvar("", "lunedi");
Lvar monday = new Lvar("", "monday");
```

the following three are all legal invocations of the method `weekTranslate`:

```
weekTranslate(lunedi,answer);
weekTranslate(answer,monday);
weekTranslate(lunedi,monday);
```

That is, the same method can be used both to test a given translation and to translate a given week name into another one. No assumption is made about which are input and which are output parameters. In the first invocation, the first parameter is an input and the second is an output, and vice versa in the second invocation, while in the third invocation both are input parameters. Actually, the method `weekTranslate` defines a binary *relation*, not a function that maps one or more inputs to exactly one output. [2]

If no matches are found for the passed values, the `weekTranslate` method raises a `Failure` exception. The default action for this exception is the immediate termination of the current thread. The exception, however, can be caught by the program and dealt with as preferred. In general, we say that the invocation of a method terminates with *failure* if its execution causes the `Failure` exception to be raised; otherwise we say that it terminates with *success*.

The real power of logical variables should become apparent after considering their usage in defining and handling dynamic data structures, namely lists and sets.

3 Data structures and partially specified values

JSetL provides two distinct kinds of data structures: lists and sets. The main difference between them is that, while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter.

Data structures that contain uninitialized logical variables in place of single elements or as a part of the data structure itself represent *partially specified values*. This is another feature that is often advocated as fundamental for the declarative computational model [14].

List and sets in JSetL are declared as instances of classes `Lst` and `Set`, respectively. Moreover, methods are provided to build lists and sets out of their elements. Elements can be of any type, including `Lvar`, `Lst`, and `Set` objects. Both lists and sets can be partially specified, as shown in the following two examples. Hereafter we will use the Prolog notation as an abstract syntax for lists.

Example. Check whether a list `L` contains at least two elements.

```
public static void at_least_two(Lst L)
throws Failure {
    Lvar x = new Lvar();
    Lvar y = new Lvar();
    Lst R  = new Lst();  // an uninitialized list
    Solver.solve(L.eq(R.ins1(y).ins1(x)));  // L = [x, y|R]
    return;
}
```

The problem is solved—in a truly declarative way—by requiring that the given list `L` satisfies the equality `L` = [x, y|R]. The `ins1` method is used to insert the value of the specified

[2] Note that the use of uninitialized logical variables as output parameters provides a simple solution to the problem caused by the fact that Java always passes primitive types and object references to methods by value.

argument as the first element of the list on which it is invoked. The `solve` method differs from the `boolSolve` method in that it does not return an explicit result, but it can cause a `Failure` exception to be generated if its argument expression turns out to be not satisfiable.

Observe that JSetL methods are declarative. For instance, the list element insertion and removal methods do not modify the list on which they are invoked: rather they build and return a new list obtained by adding/removing the elements to/from the input list.

As two further examples of declarative programs using partially specified lists consider the following two simple problems.

Example. Deterministic concatenation. Given two lists `L1` and `L2` computes the new list `L3` as the concatenation of `L1` and `L2`.

```
public static void concat(Lst L1, Lst L2, Lst L3)
throws Failure {
    if (Solver.boolSolve(L1.eq(Lst.empty))) {
        Solver.solve(L2.eq(L3));   // L3 = L2
        return;
    }
    else {
        Lvar x     = new Lvar();
        Lst R      = new Lst();
        Lst L3new = new Lst();
        Solver.solve(L1.eq(R.ins1(x)).and(L3.eq(L3new.ins1(x))));
                                   // L1 = [x|R] ∧ L3 = [x|L3new]
        concat(R,L2,L3new);
        return;
    }
}
```

Observe that in this version of `concat` `L1` and `L2` are required to be input parameters. `L3` instead can be either an output or an input parameter (i.e., `concat` can be used also to check whether a list is the concatenation of two other lists). A fully invertible definition of `concat` requires non-determinism and will be shown in Sect. 7.

Example. Check whether all elements of a set `s` are pairs, i.e., they have the form $[x1, x2]$, for any $x1$ and $x2$.

```
public static void all_pairs(Set s)
throws Failure {
    Lvar x1  = new Lvar();
    Lvar x2  = new Lvar();
    Lvar[] localVars = {x1, x2};
    Lvar x   = new Lvar();
    Solver.solve(forall(x,s,localVars,x.eq(Lst.empty.ins1(x1).insn(x2))));
    return;
}
```

The logical meaning of the `forall` expression in the above example is:

$$\forall \mathtt{x}((\mathtt{x} \in \mathtt{s}) \rightarrow \exists \mathtt{x1}, \mathtt{x2}(\mathtt{x} = [\mathtt{x1}, \mathtt{x2}])).$$

`localVars` specifies the variables which are to be intended as “local” to the `forall` expression. Concretely, a new copy of these variables is created for each element `x` of the set `s`.

4 Nondeterministic operations

Nondeterminism provides a very high-level control abstraction. The ability to state solutions in a nondeterministic way, along with the availability of logical variables, represents another valuable support to obtain declarative programs.

The notion of nondeterminism fits into that of set very naturally. Set unification and many other set operations are inherently and naturally nondeterministic. For example, the evaluation of $x \in \{1,2,3\}$, with x an uninitialized variable, nondeterministically returns one among $x = 1$, $x = 2$, $x = 3$. The main source of nondeterminism in JSetL, therefore, are set operations. In this section we show three examples where set operations—in particular, membership and set unification—are exploited to nondeterministically bind values to logical variables.

Example. Compute the maximum of a set of integers `s`.

```
public static Lvar max(Set s)
throws Failure {
    Lvar x = new Lvar();
    Lvar y = new Lvar();
    Solver.solve(x.in(s));
    Solver.solve(forall(y, s, y.leq(x)));
    return x;
}
```

The declarative reading of this method is: an element `x` of `s` is the maximum of `s`, if for each element `y` of `s` it holds that `y` $\leq$ `x`.

Set unification is nondeterministic. For example, solving the set unification problem {`x,y,z`}={`1,2,3`} nondeterministically returns, one after the other, six different solutions:

```
x = 1, y = 2, z = 3,
x = 1, y = 3, z = 2,
x = 2, y = 3, z = 1, ...
```

and so on. This feature of set unification can be immediately exploited to provide a simple (declarative) solution to the common problem of computing permutations.

Example. Compute all permutations (one at a time) of the set {`1,2,3,4`}.[3]

```
public static void main (String[] args)
throws IOException, Failure {
    int[] I_elems = {1,2,3,4};
    Set I = new Set(I_elems);    // I = {1,2,3,4}
    Lvar[] S_elems = {new Lvar(), new Lvar(), new Lvar(), new Lvar()};
    Set S = new Set(S_elems);    // S = {_S1,_S2,_S3,_S4}
    Solver.solve(S.eq(I));       // {_S1,_S2,_S3,_S4} = {1,2,3,4}
    S.output();

    System.out.print("Do you want another solution(y/n)?");
    InputStreamReader reader = new InputStreamReader(System.in);
    BufferedReader input     = new BufferedReader(reader);

    while (input.readLine().equals("y")) {
        if (Solver.nextSolution()) {
            S.output();
            System.out.print("Do you want another solution(y/n)?");
        }
        else {
            System.out.println("no");
            break;
        }
```

[3] For the sake of simplicity we assume the problem deals with a specific set of integers. However, the solution could be easily generalized to any set: the cardinality `n` of the set can be computed using the `size` method of class `Set`, while the set of `n` uninitialized variables to be used in the set unification problem can be constructed calling the `mkSet(n)` method of class `Set`.

```
        }
    }
```

In this example we show also how to implement interaction with the user that allows the user to get one solution at a time, when requested (a'la Prolog). The `nextSolution` method allows to obtain a new solution, if any, for the last issued `solve` (thus, in the example, a new solution for the equality `S.eq(I))`. If no further solution exists, `nextSolution` returs a false result.

As a more complete example, showing the use of nondeterminism in connection with sets, we show a possible JSetL solution to the well-known problem of the traveling salesman problem.

Example. Given a directed labeled graph G, the *traveling salesman problem (TSP)* consists in determining whether there is a path in G starting from a source node, passing exactly once for every other node, and returning in the initial node (here we are making the simplification of not considering costs attached to edges; the problem, and the proposed solution, however, can be straightforwardly extended to weighted graphs as well, where the goal is that of finding a path of global cost less than a constant k).

A directed labeled graph G can be represented as a pair $\langle N, E\rangle$ where N is the set of nodes and E is the set of edges, and each edge has the form $\langle n_1, n_2\rangle$, with $n_1, n_2 \in N$. This representation of graphs has an immediate implementation using JSetL's data structures: N can be implemented as a `Set` object containing simple elements (e.g., strings), and E as a `Set` object whose elements are lists (i.e., `Lst` objects) of two nodes. If E contains the list $[a, b]$ it means that the graph contains an arc from node a to node b (note that arcs can be conveniently implemented using sets—instead of lists—if the graph is undirected).

The following is the core part of the Java program which is able to solve the (simplified) TSP problem using JSetL.

```
import JSetL.*;
import java.io.*;

class Tsp {
   static Set visited = Set.empty;
   static Lst path    = Lst.empty;
   static int n;

   public static void main (String[] args)
   throws IOException, Failure {
       InputStreamReader reader = new InputStreamReader(System.in);
       BufferedReader in        = new BufferedReader(reader);
       Set nodes   = readNodes(in);
       Set edges   = readArcs(in, nodes);
       Lvar source = readSource(in);

       Solver.solve(source.in(nodes));
       visited = visited.ins(source);
       path    = path.ins1(source);
       tsp(edges, source, source);

       showPath();
       return;
   }

   public static void tsp(Set edges, Lvar source, Lvar startNode)
   throws Failure {
       Lvar nextNode = new Lvar();
       Lst newArc    = new Lst(Lst.empty.ins1(startNode).insn(nextNode));
```

```
        Solver.solve(newArc.in(edges).and(nextNode.nin(visited)));
        visited = visited.ins(nextNode);
        path    = path.insn(nextNode);
        if (path.size() < n) tsp(edges, source, nextNode);
        else tspLast(edges, source, nextNode);
        return;
    }

    public static void tspLast(Set edges, Lvar source, Lvar lastNode)
    throws Failure {
        Lst backArc = new Lst(Lst.empty.ins1(source).ins1(lastNode));
        Solver.solve(backArc.in(edges));
        return;
    }
}
```

The set `visited` is the set of all the already examined nodes, while the list `path` is used to store the computed path. Initially they are both empty. The `main` method first asks the user to supply the nodes and the edges of the graph, and then the initial node which is stored in the logical variable `source`. We assume the interaction with the user is implemented by methods `readNodes`, `readArcs`, `readSource` in some reasonable way.

Methods `tsp` and `tspLast` provide our solution to the TSP. Statements in the `main` method preceding the invocation of the `tsp` method state that the source node must belong to the set of nodes of the given graph G and that it must be added to the set of visited nodes and to the list that represents the computed path (as the first element). Then the `tsp` method is called with `source` as the start node. The definition of the `tsp` method simply states that an edge leaving from node `startNode` and ending in a node `nextNode` not yet visited must belong to the set of edges of G. If these conditions are true, then `nextNode` is added to the set of visited nodes as well as to the computed path, and the `tsp` method is called recursively with `nextNode` as the new initial node, unless all nodes have been already visited. In the last case, the `tspLast` method is called instead; this method simply states that there must be an edge (`backArc`) from the last visited node to the initial one. If the problem admits at least one solution the invocation of the `tsp` method within the `main` method terminates successfully. Thus the `main` method can invoke `showPath()` which prints the solution stored in `path`.

The following is an example of a possible run of the program.

```
Insert the set of all nodes (separated by blanks): a b c d e

Insert the nodes reachable from a : b c
Insert the nodes reachable from b : a e
Insert the nodes reachable from c : b
Insert the nodes reachable from d : a
Insert the nodes reachable from e : d

Insert the initial node: a

Path: a -> c -> b -> e -> d -> a
```

5 Constraints

Declarative programming is often associated with *constraint programming* (CP). As a matter of fact, the ability to deal with constraints allows to compute with partially specified values. For example, the disequality

$$[x] \neq [1],$$

with x an uninstantiated logical variable, can be solved if one can state (and remember) that it must hold that $x \neq 1$.[4] As another example, the expression

$$\texttt{less}(\{x\}, 1, r),$$

where x is an uninstantiated logical variable, r is an uninitialized set, and the intended meaning of $\texttt{less}(s, e, r)$ is $r = s \backslash \{e\}$, admits two possible solutions, corresponding to two possible constraints: $x = e \wedge r = \{\}$ and $x \neq e \wedge r = \{x\}$. Furthermore, an expression such as

$$\texttt{un}(x, y, z),$$

with x, y, z uninstantiated logical variables, can not be further simplified and must be kept unaltered as a constraint over the possible values that can be subsequently assigned to the variables.

In JSetL basic set-theoretical operations, as well as equalities, inequalities and integer comparison expressions, are dealt with as *constraints*. Constraints are handled by a *constraint solver*, which is an instance of the class `SolverClass`. The current collection $\mathcal{C}$ of active constraints for a constraint solver S is stored in the *constraint store* (CS) of S. Constraints are added to the CS of a solver using the `add` method. After constraints have been stored in the CS, one can invoke their resolution by calling the `solve` method on the related solver. The solver nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store, by rewriting them to a simpler form. Constraint solving in JSetL is basically the same developed for CLP($\mathcal{SET}$) [6].

The ability to solve constraints disregarding the instantiation of variables possibly occurring in them allows the programmer to ignore (in some cases) the order of execution of statements in a program (hence, to have a more declarative reading of programs). For example, if `x` is an uninstantiated logical variable, executing

```
x.read();
Solver.add(x.neq(0));
Solver.solve();
```

is the same as executing

```
Solver.add(x.neq(0));
x.read();
Solver.solve();
```

Programming with constraints allows the programmer to express solutions to a possibly complex problem as a collection of equations and disequations, that will be solved in a subsequent moment, with no concern to the order in which they occur in the program. As such, CP languages constitute powerful modeling tool, particularly suitable to coincisely express solutions for artificial intelligence and constraint-satisfaction problems (e.g, combinatorial problems). The following is an example of one such well-known problem.

Example. The n-queens problem.

The n-queens problem consists in trying to place n queens on a $n \times n$ chess board such that no two queens attack each other, i.e., no two queens are placed on the same row, the same column or the same diagonal.

To model the problem we observe that any solution will have a queen on each row and one on each column. We can then represent each row with a logical variable, $r_0, ..., r_{n-1}$. For every row there are n columns on which we can place the queen. If we identify the columns with the integer $0, ..., n-1$, the resolution of the problem consists in assigning a value among 0 and $n-1$ to each of the n logical variables $r_0, ..., r_{n-1}$ such that

(*i*) $r_i \neq r_j$
(*ii*) $r_j + j - r_i \neq i$
(*iii*) $r_i + j - r_j \neq i$

[4] The alternative would be to prohibit evaluation of not completely specified expressions, raising an exception indicating an instantiation error.

for each $i \neq j$, $0 \leq i,j \leq n-1$. The first inequality states that two queens can not be placed on the same column, while the other two inequalities state that two queens can not be placed on the same diagonal.

The following is a Java program for the n-queens problem using JSetL (with $n = 4$).

```
import JSetL.*;
import java.io.*;

class Queens {
  public static final int n = 4;

  public static void main (String[] args)
  throws IOException, Failure {
    Set colmns = new Set(0, n-1); // a set of n integers representing columns
    Lst rows   = Lst.mkLst(n);     // a list of n Lvar's representing rows

    for (int i = 0; i < n; i++) // assign a distinct int. in 0..n - 1 to each var. in rows
      Solver.add(((Lvar)rows.get(i)).in(colmns));
    Solver.allDifferent(rows);

    for (int i = 0; i < n-1; i++)  // add constraints on diagonals
      for (int j = i+1; j < n; j++) {
        Solver.add(((Lvar)rows.get(j)).sum(j).sub((Lvar)rows.get(i)).neq(i));
        Solver.add(((Lvar)rows.get(i)).sum(j).sub((Lvar)rows.get(j)).neq(i));
    }
    Solver.solve(); // check constraints

    for (int i = 0; i < n; i++) {  // print a solution
      System.out.print("\n Queen "+i+" = ");
      ((Lvar)rows.get(i)).print();
    }
    return;
  }
}
```

`get` is an utility methods for lists: `l.get(i)` returns the `i`-th element of the list `l`. `allDifferent` is used to add constraints to the CS: `Solver.allDifferent(a)`, where `a` is either a (bounded) list or set containing n elements $e_1,\ldots,e_n$, adds the atomic constraints $e_i \neq e_j$ to the CS of `Solver`, for all i, j in $1,\ldots,n$. `sum` and `sub` are methods of class `Lvar` which implement addition and subtraction over integers, respectively: they allow to write usual arithmetic expressions but involving logical variables.

Integers in set `colmns` indicate the columns on which the queens can be placed. Logical variables in list `rows` indicate the rows of the chess board. A value c in the i-th `Lvar` of `rows` indicates a queen placed in row i at column c. The `allDifferent` method assures that only one queen is placed on each different column. The other two constraints added to the CS through the `add` method implement inequalities (ii) and (iii) (note that `rows.get(i)` and `rows.get(j)` denote the i-th and the j-th `Lvar` of the list `rows`, respectively).[5]

A possible solution printed by this program is:

```
Queen0 = 1
Queen1 = 3
```

[5] The current version of the JSetL constraint solver is not able to deal with integer arithmetic constraints in a complete way: specifically, it is not able to decide satisfiability of integer constraints containing uninitialized variables when the `solve` method is invoked. These limitations will be possibly overcome in future releases, following, for example, the approach described in [4], where the $\mathcal{SET}$ solver is integrated with an efficient solver over *finite domains* which allows the language to deal with arithmetic constraints with almost no restriction on the instantiation of expressions that occur in them.

```
Queen2 = 0
Queen3 = 2
```

that represents the situation depicted below

	Q		
			Q
Q			
		Q	

JSetL allows the user to define new constraints. User-defined constraints are dealt with as the built-in constraints: they can be added to the constraint store using the `add` method and solved using the `SolverClass` methods for constraint solving. Definition of user-defined constraints, however, requires some programming conventions to be respected. In particular, all definitions must be provided as part of a class named `NewConstraints`. This class extends the `SolverClass` and must be defined as part of the JSetL package (see [11] for a detailed description).

The user can exploit the ability to define new constraints, in particular, whenever he/she wants a part of his/her program to be "sensible" to backtracking. As an example, let us consider the following trivial program fragment, where we assume that `x` is an uninitialized logical variable and `s` is the set {`0,1`}:

```
Solver.solve(x.in(s));
x.output();
Solver.solve(x.neq(0));
```

Executing this program will print only one of the value nondeterministically assigned to `x` by the `x.in(s)` constraint, namely, the first one. The problem is caused by the fact that nondeterminism in JSetL is confined to constraint solving: backtracking allows the computation to go back to the nearest open choice point *within* the constraint solver, but it does not allow to "re-execute" user program code. In this example, the `output` statement is not re-executed when the second alternative for the `x.in(s)` constraint is taken into account. The problem can be solved using the `NewConstraints` class. The piece of code that requires to be re-executed is defined as the body of a new constraint in the `NewConstraints` class, with proper parameter passing:

```
public static void printVar(Lvar x) {
    x.output();
    return;
}
```

Then the sample program fragment is modified as follows:

```
Solver.solve(x.in(s));
Solver.solve(NewConstraints.printVar(x));
Solver.solve(x.neq(0));
```

With these changes, execution of this program causes <u>all</u> values assigned to `x` to be printed, that is:

```
x = 0
x = 1
```

Observe that the definition of the `printVar` method shown above is not the Java code which is really executed. The real code is obtained through a straightforward preprocessing of the NewConstraints class code, that inserts all auxiliary definitions which allow to deal with methods defined by the user in this class as new constraints. Refer to [11] for more details.[6]

Another, more sophisticated, usage of the new constraint facility will be presented in Section 7.

6 Intensional sets

A very common way to represent a set is by its *intensional definition*, that is by providing a condition φ that is necessary and sufficient for an element x to belong to the set itself: $\{ x : \varphi \}$. Intensional set definitions represent a powerful tool for supporting a declarative programming style, as shown for instance in [6].

JSetL provides a limited form of intensional set definition through the use of the setof method. The invocation Solver.setof(x,C), allows the program to collect into a set all values of the logical variable x, satisfying the given constraint C.

As an example, the following code defines, in a truly declarative way, a method to compute and print the set of all pairs [x,y] such that x and y belong to a given s and x $\neq$ y:

```
public static Lvar allPairs(Set s)
throws Failure {
    Lvar x    = new Lvar();
    Lvar y    = new Lvar();
    Lst pair  = new Lst(Lst.empty.ins1(x).insn(y));  // a pair [x,y]
    Set Pairs = new Set();
    Solver.setof(pair, x.in(s).and(y.in(s)).and(x.neq(y)));
                        // collect all solutions for x
                        // satisfying the constraint x ∈ s ∧ y ∈ s ∧ x ≠ y
    Pairs.output();
    return;
}
```

If s is the set {1,2,3,4} (and the CS is initially empty), the set printed by executing this method is

{[1,2],[1,3],[1,4],[2,1],[2,3],[2,4],[3,1],[3,2],[3,4],[4,1],[4,2],[4,3]}.

Observe that, the logical meaning of the above constraint is:

$$\forall \mathtt{pair}((\mathtt{pair} \in \mathtt{s}) \leftrightarrow \exists \mathtt{x}, \mathtt{y}(\mathtt{x} \in \mathtt{s} \land \mathtt{y} \in \mathtt{s} \land \mathtt{x} \neq \mathtt{y})).$$

That is, the uninitialized logical variables possibly occurring in the constraint are considered as existentially quantified within the scope of the intensional definition of the set itself.

The same effect of Solver.setof(x,C) can be obtained by invoking first the add method on C, Solver.add(C), then the setof method with only one parameter, Solver.setof(x), which implicitly considers the constraint stored in the current CS. This form of the setof method can be particularly convenient when the constraint to be dealt with is quite complex.

Example. Prime numbers.

Compute the set of all prime numbers smaller or equal to a given integer n.

```
public static Set primes(Lvar n)
throws Failure {
    Lvar x = new Lvar();
```

[6] This preprocessing facility is not yet provided by the JSetL library; hence, the required program transformation of methods of the NewConstraints class must be performed at hand.

```
        Solver.add(x.in(Set.interv(2,n)));    // x in 2..n
        // check whether x is prime
        Lvar y = new Lvar();
        Lvar m = new Lvar();
        Solver.add(m.eq(x.sub(1)));           // m = x - 1
        Solver.forall(y, Set.interv(2,m),(x.mod(y)).neq(0));
        // compute the set of all primes numbers in 2..n
        return Solver.setof(x);
    }
```

In more abstract terms, the `setof` method collects the set $\{\mathtt{x} : \mathtt{x} \in \mathtt{2..n} \wedge \forall \mathtt{y}\ (\mathtt{y} \in \mathtt{2..x-1})\ \mathtt{x\ mod\ y} \neq \mathtt{0}\}$.

7 Nondeterminism between statements

Besides exploiting the nondeterminism embedded in the predefined constraints, the user can implement her/his own nondeterministic procedures by exploiting the facilities for the new constraint definition provided by the JSetL library. In fact, JSetL allows the user to specify, within methods of the `NewConstraints` class, the nondeterministic choice (with backtracking) among two or more different blocks of statements.

As an example, the following code fragment shows a fully nondeterministic version of the list concatenation operation shown in Section 3. To make the code more readable, we assume a special construct, ***either*** $\mathtt{S}_1$ ***orelse*** $\mathtt{S}_2$ ... ***orelse*** $\mathtt{S}_\mathtt{n}$, where $\mathtt{S}_1,\dots,\mathtt{S}_\mathtt{n}$ are Java statements, is available. The logical meaning of this construct is the disjunction $\mathtt{S}_1\vee,\dots,\vee\ \mathtt{S}_\mathtt{n}$, while its computational interpretation is that of exploring, through backtracking, all possible alternatives $\mathtt{S}_1,\dots,\mathtt{S}_\mathtt{n}$, starting from $\mathtt{S}_1$. The *either-orelse* construct can be used only within the `NewConstraints` class. This construct must be replaced, through a straightforward preprocessing phase, by usual Java code endowed with JSetL facilities for handling nondeterminism. Refer to [11] for more details.

```
class NewConstraints {
    public static void concat(Lst l1, Lst l2, Lst l3) {
        either {
            Solver.add(l1.eq(Lst.empty));
            Solver.add(l2.eq(l3));
        }
        orelse {
            Lvar x     = new Lvar();
            Lst l1new = new Lst();
            Lst l3new = new Lst();
            Solver.add(l1.eq(l1new.ins1(x)));
                                  // l1 = [x | l1new]
            Solver.add(l3.eq(l3new.ins1(x)));
                                  // l3 = [x | l3new]
            Solver.add(concat(l1new, l2, l3new));
        }
        return;
    }
}
```

This version of the `concat` method can be used not only to compute `l3` out of `l1` and `l2` or to test whether `l3` is the concatenation of l1 and l2, but also to compute (nondeterministically) all possible lists l1 and l2 from l3. Such flexibility is obtained by using unification, instead of standard assignment, and nondeterminism, instead of a deterministic **if** statement.

A simple example where we use `concat` to split a list into two sub-lists is the following definition of a method `prefix` that checks if a list `l1` is a prefix of a list `l2`:

```
public static void prefix(Lst l1, Lst l2)
throws Failure {
    Lst l = new Lst();
    Solver.add(NewConstraints.concat(l1, l, l2));
    Solver.solve();
    return;
}
```

Generally speaking, the ability to express nondeterminism allows the user to define methods which are completely "invertible", that is that do not impose any distinction between input and output parameters. In particular we can define methods that map a value to a set of different values (differently from functions where every element of the domain can be mapped to at most one element of the range). For instance, as shown above, we can use `concat` as a mapping from a list of n elements (`l3`) to the set of $n+1$ pairs of lists (`l1` and `l2`) in which the given list can be splitted.

Moreover, the ability to choice nondeterministically one action among many can be conveniently exploited whenever one has to implement an algorithm which requires to explore a search space. Again, the presence of a high-level control abstraction such as that represented by the ***either-orelse*** construct can be of great help to devise solutions in a highly declarative style. As an example, the following is a completely nondeterministic method to find a path between two nodes in a directed labeled graph, represented as a set of pairs constituting the edges of the graph, as done in Section 4. To allow nondeterminism to be fully exploited, `path` is defined as a new constraint in the `NewConstraints` class.

```
public static void path(Set G, Lvar source, Lvar dest) {
    either {
        Lst finalEdge = Lst.empty.ins1(source).insn(dest));
        Solver.add(finalEdge.in(G));
    }
    orelse {
        Lvar x = new Lvar();
        Lst intermediateEdge = Lst.empty.ins1(source).insn(x));
        Solver.add(intermediateEdge.in(G));
        Solver.add(path(G, x, dest);
    }
    return;
}
```

As an example, if `g1` is the graph

{['a','b'],['a','c'],['c','b'],['b','d'],['c','d']},

`s1` is an `Lvar` with value `'a'`, and `d` is an uninitialized variable, the invocation

`Solver.solve(NewConstraints.path(g1,s1,d))`

nondeterministically assigns to `d` the values `'b'`, `'c'`, `'d'`, which represent all nodes reachable from `'a'`.

8 Conclusions and future work

We have shown how facilities provided by the JSetL library can be used to write programs that exhibit a quite good declarative reading, while maintaining all the features of conventional Java programs.

As noted in the Introduction, computation efficiency has not been a primary requirement in the development of JSetL. As a matter of fact, various improvements to the current implementation can be devised, such as the use of more sophisticated techniques to handle choice points and backtracking, the use of a more efficient data representation and processing of sets and lists in the ground case, and so on. A more critical source of possible inefficiency, however, lies in the kind of constraint solving technique supported at present by our constraint solver, which is basically a "generate & test" technique (see, for instance, solution of the n-queens problem in Sect. 5). To overcome limitations of this approach—which are the same present in CLP($\mathcal{SET}$)—we plan to move along the lines suggested in [4]: integrating our (set) constraint solver with a more efficient constraint solver over *finite domains* (FD). Whenever the JSetL solver detects constraints that can be handled as FD constraints (e.g., membership constraints over finite sets of integer numbers) it delegates their resolution to the FD solver. This allows us to maintain the expressive power and flexibility of the JSetL constraint domain in the general case, while permitting efficient FD-constraint processing in many particular cases.

As another future work, simple preprocessing tools should be developed to make JSetL facilities simpler and more natural to use.

Finally, making the solver an instantiable class (not a static one, as in the first release) opens a number of interesting new possibilities:

- more solvers can coexist and cooperate
- solvers can be different threads
- constructors of the solver class can be parametric w.r.t. constraint solving policies.

Acknowledgments

This work is partially supported by MIUR project *AIDA—Abstract Interpretation Design and Applications.*

References

1. Choco. Available at `http://www.choco-constraints.net`
2. A.Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.
3. A.Courtney. Frappé: Functional reactive programming in Java. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 29–44, 2001.
4. A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *PPDP'03 — Proc. of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229, 2003.
5. E.Denti, A.Omicini, and A.Ricci. `tu`Prolog: a light-weight Prolog for internet applications and infrastructures. In *Practical Aspects of Declarative Languages, PADL 2001*, LNCS, Vol. 1990, Springer-Verlag, 184–198, 2001.
6. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.
7. A. Dovier, E. Pontelli, and G. Rossi. Set unification. TR-CS-001/2001, Dept. of Computer Science, New Mexico State University, USA, January 2001 (available at `www.cs.nmsu.edu/TechReports`).
8. G.Falkman, O.Torgersson. Enhancing Usefulness of Declarative Programming Frameworks through Complete Integration. In *Proc. of the 12th Int. Workshop on Logic Programming Environments*, July 2002 (available at `http://xxx.lanl.gov/abs/cs.SE/0207054`).
9. ILOG Optimisation Suite - White Paper. Available at `www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf`.
10. JCL - the Java Constraints Library. Available at `http://liawww.epfl.ch/JCL/`

11. E.Panegai, E.Poleo, and G.Rossi. JSetL User's Manual - Version 1.0. Research Report "Quaderno del Dipartimento di Matematica", n. 384, Università di Parma, November 2004.
12. J.-F.Puget and M.Leconte. Beyond the Glass Box: Constraints as Objects. In *Proc. of the 1995 Int'l Symposium on Logic Programming*, MIT press, pp. 513-527.
13. G.Rossi and E.Poleo. JSetL: Declarative Programming in Java with Sets. *In CF '04, 2004 ACM International Conference on Computing Frontiers*, ACM Press, 2004.
14. P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming.* The MIT Press, 2004.