The JSetL library:
supporting declarative programming in Java

E. Panegai, E. Poleo, G. Rossi
Dipartimento di Matematica
Università di Parma, Parma (Italy)
panegai@cs.unipr.it, gianfranco.rossi@unipr.it

1 Introduction

In this demonstration we present a Java library, called JSetL, that offers a number of facilities to support declarative programming like those usually found in logic or functional declarative languages: logical variables, list and set data structures (possibly partially specified), unification and constraint solving over sets, nondeterminism.

Declarative programming is often associated with *constraint programming*. As a matter of fact, constraints provide a powerful tool for stating solutions as sets of equations and disequations over the selected domains, which are then solved by using domain specific knowledge, with no concern to the order in which they occur in the program.

Differently from other works (e.g., [1, 4]) in JSetL we do not restrict ourselves to constraints, but we try to provide a more comprehensive collection of facilities to support a real declarative style of programming. Furthermore, we try to keep our proposal as general as possible, to provide a general-purpose tool not devoted to any specific application.

2 Main features of JSetL

The most notable characteristics of JSetL are:

- *Logical variables.* The notion of logical variable (like that usually found in logic and functional programming languages) is implemented by the class `Lvar`. An instance of the class `Lvar` is created by the Java declaration

  ```
  Lvar VarName = new Lvar(VarNameExt,VarValue);
  ```

  where VarName is the variable name, VarNameExt is an optional external name, and VarValue is an optional value for the variable. The value of a logical variable can be of any type, provided it supports the `equals` method. Logical variables which have no value associated with are said to be *uninitialized*. A logical variable remains uninitialized until it gets a value as the result of processing some constraint involving it (in particular, equality constraints). Constraints are the main operations through which `Lvar` objects can be manipulated. Constraints can be used to set the value of a logical variable as well as to inspect it. No constraint, however, is allowed to modify the value of a logical variable.

- *Lists* and *sets*. List and set data structures (like those in [3]) are provided by classes `Lst` and `Set`, respectively. Instances of these classes are created by declarations of the form

  ```
  Lst LstName = new Lst(LstNameExt,LstElemValues);
  Set SetName = new Set(SetNameExt,SetElemValues);
  ```

where the different fields have the same meaning proposed for `Lvar` variables (obviously, transported on the new domains). `Lst` and `Set` objects are dealt with as logical variables, but limited to `Lst` and `Set` values. Moreover, list/set constructor operations are provided to build lists/sets out of their elements at run-time. The main difference between lists and sets is that, while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. Both lists and sets can be *partially specified*, i.e., they can contain uninitialized logical variables in place of single elements or as a part of the list/set.

- *Unification.* JSetL provides unification between logical variables—as well as between `Lst` and `Set` objects—basically in the form of equality constraints: `O1.eq(O2)`, where `O1` and `O2` are either `Lvar`, or `Lst`, or `Set` objects, unifies `O1` and `O2`. Unification allows one both to test equality between `O1` and `O2` and to associate a value with uninitialized logical variables (lists, sets) possibly occurring in the two objects. Note that solving an equality constraint implies the ability to solve a *set unification* problem (cf., e.g., [3]).

- *Constraints.* Basic set-theoretical operations, as well as equalities, inequalities and integer comparison expressions, are dealt with as *constraints*. Constraint expressions are evaluated even if they contain uninitialized variables. Their evaluation is carried on in the context of the current collection of active constraints $\mathcal{C}$ (the *constraint store*) using a powerful (set) constraint solver, which allows us to compute with partially specified data.

  The approach adopted for constraint solving in JSetL is basically the same developed for CLP($\mathcal{SET}$) [2]. To add a constraint C to the constraint store, the `add` method of the `Solver` class can be called as follows:

  ```
  Solver.add(C)
  ```

  The order in which constraints are added to the constraint store is completely immaterial. After constraints have been added to the store, one can invoke their resolution by calling the `solve` method:

  ```
  Solver.solve()
  ```

  The `solve` method nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store. If there is no solution, a `Failure` exception is generated; otherwise the computation terminates with *success* when the first solution is found. The default action for this exception is the immediate termination of the current thread. Constraints that are not solved are left in the constraint store: they will be used later to narrow the set of possible values that can be assigned to uninitialized variables.

- *Nondeterminism.* Constraint solving in JSetL is inherently nondeterministic: the order in which constraints are added/solved does not matter (*dont care nondeterminism*); solutions for set constraints (e.g., equality, membership,...) are computed nondeterministically, using choice points and backtracking (*dont know nondeterminism*).

  A simple way to exploit nondeterminism is through the use of the `Setof` method of the `Solver` class. This method allows one to explore the whole search space of a nondeterministic computation and to collect into a set all the computed solutions for a specified logical variable `x`.

All these features strongly contribute to support a real declarative programming style in Java. In particular, the use of partially specified dynamic data structures, the ability to deal with constraint expressions disregarding the order in which they are encountered and the instantiation of the logical variables possibly occurring in them, as well as the nondeterminism "naturally" supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modeling tool.

3 Programming with JSetL

Example 3.1 *All pairs*

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

In mathematical terms, a (declarative) solution for this problem can be stated as follows: $(\forall x \in s)(\exists x1, x2 \;\; x = [x1, x2])$*. The program below shows how this solution can be immediately implemented in Java using JSetL.*

```
public static void all_pairs(Set s) throws Failure
    {
    Lvar x1 = new Lvar();
    Lvar x2 = new Lvar();
    Lvar x = new Lvar();
    Lst pair = Lst.empty.ins1(x2).ins1(x1);

    Lvar[] LocalVars = {x1,x2};
    Solver.forall(x,s,LocalVars,x.eq(pair));

    Solver.solve();
    return;
    }
```

Example 3.1 shows the declaration of three logical variables, `x1`, `x2`, `x`, and the creation of a new object of type `Lst`, representing a partially specified list of two elements. The constraints to be solved are introduced by the `forall` method, using the `LocalVars` array to specify existentially quantified (logical) variables. The call to the `solve` method allows to check satisfiability of the current collection of constraints.

The next example is a method to compute the set of all subsets of a given set. In this case we use the `add` method to introduce a new constraint and the `setof` method to get all possible solutions (not only the first one).

Example 3.2 *All solutions*

Compute the powerset of a given set `s`*.*

```
public static Set powerset(Set s) throws Failure
    {
    Set r = new Set();
    Solver.add(r.subset(s));
    return Solver.setof(r);
    }
```

If `s` *is the set* `{'a','b'}`*, the set returned by* `powerset` *is* `{{},{'a'},{'b'}, {'a','b'}}`*.*

References

[1] A.Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.

[2] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.

[3] 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`).

[4] Neng-Fa Zhou. Building Java Applets by using DJ—a Java Based Constraint Language. Available at `www.sci.brooklyn.cuny.edu/~zhou`.