# JSetL: Declarative Programming in Java with Sets

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

## ABSTRACT

In this paper 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. The paper describes the main features of JSetL and it shows, through a number of simple examples, how these features can be exploited to support a real declarative programming style in Java.

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints*; D.1.5, D.1.6 [**Programming Techniques**]: Object-oriented Programming, Logic Programming

## General Terms

Languages, Algorithms, Theory

## Keywords

Declarative Programming; Constraint Programming; Java; Nondeterminism

## 1. INTRODUCTION

*Declarative programming* (DP) is usually called into play in the context of functional and logic programming languages (e.g., Haskell and Prolog). Intuitively, declarative programming means focusing on *what* a program does, rather than on *how* it does. Notions such as logical variables, side-effect freeness, functional composition, recursion, nondeterminism, etc., are all valuable features of a programming language that supports declarative programming. High-level of control and data abstractions, as well as a clear semantics, are also fundamental features to support the declarative reading of a program. Declarative programming have been mainly exploited in artificial intelligence and automated reasoning applications, but most of its features can be conveniently used also in more general settings to support rapid software prototyping and (automatic) program verification, as well as to allow parallel execution of programs.


*CF'04.* April 14–16, 2004, Ischia, Italy.


Declarative programming is often associated with *constraint programming* (CP), both in the context of logic programming languages (e.g., ECLIPSE [9]), and in the context of functional and functional plus logic programming languages (e.g., Oz [13])). 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. As such, CP languages constitute powerful modelling tool, in particularly suitable to coincisely express solutions for artificial intelligence and constraint-satisfaction problems (e.g, combinatorial problems).

While it is undeniable that DP languages (and, in particular, CP languages) provide valuable support for programming, 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. Efforts to make DP languages more appealing for real-world applications have led to various proposals, mainly intended to include object-oriented features into DP languages: several languages, like Prolog and Haskell, have indeed object-oriented extensions. While this attests to the expressive power of DP languages, it has yet to convert the industry from mainstream OO languages, and seems unlikely to do so in the foreseeable future.

A complementary approach is trying to embed DP features in a more conventional framework—in particular an object-oriented one—in which one can exploit the DP paradigm while retaining all the advantages of constructs for programming and software structuring that are typical of conventional programming languages. This integration can take place according to (at least) two distinct approaches: (*i*) making the new features available as part of a *library* for some existing language; (*ii*) defining a new programming language, or extending an existing one, in such a way DP features are viewed as "first-class citizens" of the language itself. Both approaches have pros and cons and a precise comparison of them is likely to be an interesting topic for future research.

One of the best known proposals that integrate some DP features in a conventional OO framework following the *library approach* is that of the ILOG Solver [14, 12]. In this system, constraints and logical variables are handled as objects and are defined within a C++ class library. Thanks to the encapsulation and operator overloading mechanisms, programmers can view constraints almost as if they really were part of the language. Among other proposals that take a similar approach we can mention INC++ [11], NeMo+ [16], and JSolver [2], as concerns the addition of constraints to OO languages, while Frappè [3] and Gisela [10] are two proposals that face the more general problem of making declarative programming features available in a conventional programming environment, though focusing on some specific applications. Another proposal that can be cited in this context is `tu`Prolog [5], a Java package that implements Prolog.

The *new language approach*, in the context of conventional programming languages (approach (*ii*)), is adopted for instance in the language Alma-0 [1], in Singleton [15], and in DJ (Declarative Java) [17, 18]. A potential advantage of this approach with respect to that based on a library is that it allows a tighter integration between constructs of the host language and DP facilities, making programs simpler and more "natural" to write. On the other hand, however, the design and development of a new language is surely a more difficult task, and the resulting systems are likely to be less easy to integrate with other existing systems and to be accepted by programmers.

The work presented in this paper is another proposal following the OO library approach: we endow an OO language, namely Java, with facilities for supporting declarative programming, by providing them as a library—called JSetL. Differently from other related work 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. The most notable features of JSetL are:

- logical variables;
- list and set data structures, possibly partially specified (i.e., containing uninitialized logical variables)
- unification (in particular, unification over lists and sets)
- a powerful set constraint solver which allows to compute with partially specified data
- nondeterminism (though confined to constraint solving).

We claim that these facilities provide a valuable support to *declarative programming* and we show this with a number of simple examples. In particular the constraint solver allows complex (set) expressions to be checked for satisfiability, disregarding their order and the instantiation of (logical) variables occurring in them. Moreover, the use of partially specified data structures, along with the nondeterminism "naturally" supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modelling tool.

The paper is organized as follows. In Section 2 we give an informal presentation of JSetL by showing a simple Java program using JSetL. In Section 3 we introduce the fundamental data structures of JSetL, namely logical variables, sets and lists. In Section 4, we describe the (set) constraint handling facilities supported by our library and we show how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. The fundamental notion of nondeterminism and its relationship with sets is addressed in Section 5. In Section 6 we show how user defined constraints can be introduced in a program and how they can be used. Finally, in Section 7 we briefly discuss future work.

## 2. AN INFORMAL INTRODUCTION TO PROGRAMMING WITH JSetL

First of all we show a simple example of a Java program using JSetL which allows us to give the flavor of the programming style supported by the library.

**Problem**: Compute and print the maximum of a set of integers `s`.

A truly declarative solution for this problem can be stated as follows: an element `x` of `s` is the maximum of `s`, if for each element `y` of `s` it holds that `y` $\leq$ `x`. The program below shows how this solution can be immediately implemented in Java using JSetL. Observe that here we are deliberately assuming that execution 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.

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

    public static void main (String[] args)
    throws IOException, Failure
        {
        int[] sample_set_elems = {1,6,4,8,10,5};
        Set sample_set = new Set(sample_set_elems);
        System.out.print(" Max = ");
        max(sample_set).print();
        }
    }
```

For the sake of simplicity we assume that the set of integers is directly supplied by the program (instead of being read for instance from a file). Hence we will focus on the definition of the method `max` that computes the maximum of `s`. `x` and `y` in `max` are two logical variables and both are uninitialized. Invocation of the `add` method adds the constraint `x.in(s)` (i.e., `x` $\in$ `s`) to the current constraint store. This constraint is evaluated to `true` if `s` is a set and `x` belongs to `s`. If `x` is uninitialized when the expression is evaluated this amounts to *nondeterministically* assign an element of `s` to `x`. Invocation of the `forall` method allows us to add to the constraint store a new constraint `y.leq(x)` (i.e., `y` $\leq$ `x`) for each `y` belonging to `s`. As soon as the `solve` method is invoked the constraint solver checks whether the current collection of constraints in the constraint store is satisfiable or not. If it is, the invocation of the `solve` method terminates with success. The value of `x` represents the integer we are looking for and it is returned as the result of `max`. If, on the contrary, one of the constraints in the constraint store is evaluated to false, backtracking takes place and the computation goes back till the nearest choice point. In this case, the nearest and only choice point is the one created by the `x.in(s)` constraint. Its execution will bind nondeterministically `x` to each element of `s`, one after the other. If all values of `s` have been attempted, there is no further alternative to explore and the computation of `max` terminates raising an exception `Failure`. If no **catch** clause for this exception is provided, the whole computation terminates reporting a failure (actually this is not the case of the `max` method, since a value of `x` for which all the constraints hold surely exists—exactly the maximum of `s`).

Executing the program with the sample set of integers declared in the **main** method causes the message `Max = 10` to be printed to the standard output.

## 3. LOGICAL VARIABLES AND COMPOSITE DATA OBJECTS

JSetL provides logical variables and two new kinds of data structures: sets and lists. These new features are implemented by three classes, `Lvar`, `Lst`, and `Set`, for creation and manipulation of logical variables, lists and sets, respectively.

Lists and sets represent two different data abstractions: while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. Thus, for instance, $\{1,2\}$, $\{2,1\}$, and $\{2,1,2\}$ all denote the same set, while the analogous list expressions denote different lists. Although sets and lists can be often used interchangeably, there are cases—especially when sets and lists contain unknown elements—in which one choice can be more appropriate than the other one. For example, if one wants to represent an undirected graph, arcs are likely to be represented as sets rather than as lists. As another example, if one wants to state that a collection $C$ must contain the number 1, disregarding the position of 1 in $C$, $C$ is conveniently represented as a partially specified set (cf., e.g., set `s2` in Example 3.2).

### 3.1 Logical variables

A *logical variable* is an instance of the class `Lvar`, created by the statement

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

where VarName is the variable name, VarNameExt is an optional external name of the variable, and VarValue is an optional `Lvar` value associated with the variable.

The *external name* is a string value which can be useful when printing the variable and the possible constraints involving it (if omitted, a default name of the form "Lvar_$n$", where $n$ is a unique integer, is assigned to the variable automatically). `Lvar` are not typed. An *Lvar value* can be either a primitive type value, or any library or user defined class object (provided it supplies a method `equals` for testing equality between two instances of the class itself). In particular, an `Lvar` value can be an instance of `Lvar`, `Lst`, or `Set`.

A logical variable which has no `Lvar` value associated with it, or whose `Lvar` value is an uninitialized logical variable (or an uninitialized list or set), is said to be *uninitialized* (or an *unknown*). Otherwise, the logical variable is *initialized*. `Lvar` values other than uninitialized logical variables (or lists or sets) are said *known values*. Uninitialized logical variables will possibly assume a known value (i.e., they become initialized) during the computation, in consequence of some constraints involving them.

### 3.2 List and set definitions

A *list* is a finite (possibly empty) sequence of arbitrary `Lvar` values (i.e., the *elements* of the list). In JSetL a list is an instance of the class `Lst`, created by the statement

```
Lst LstName = new Lst(LstNameExt,LstElemValues);
```

where LstName is the list name, LstNameExt is an optional *external name* of the list, and VarElemValues is an optional array of `Lvar` values $c_1,\ldots,c_n$ of type $t$, which constitute the elements of the list. The constant `Lst.empty` is used to denote the *empty list*. No typing information on elements of a list are provided.

A list can be either initialized or uninitialized. An uninitialized list is like a logical variable, but constrained to be (possibly) initialized by list objects only.

A *set* is a finite (possibly empty) collection of arbitrary `Lvar` values (i.e., the *elements* of the set). In JSetL a set is an instance of the class `Set`, created by the statement

```
Set SetName = new Set(SetNameExt,SetElemValues);
```

where SetName, SetNameExt, and SetElemValues have the same meaning than in lists. The constant `Set.empty` is used to denote the *empty set*. Like a list, a set can be either initialized or uninitialized, and no typing information are associated with the elements of a set. Differences between lists and sets become evident when operating on them through list/set operations (e.g., list/set unification—see Sect. 4).

EXAMPLE 3.1. `Lvar`, `Lst`, *and* `Set` *definitions*

```
Lvar x = new Lvar();               // uninitialized l. var.
Lvar y = new Lvar("y",'a');        // initialized l. var.
                                   // (value 'a'),
                                   // with ext'l name "y"
Lvar t = new Lvar(x);              // uninitialized l. var.
                                   // (same as variable x)
Lst l = new Lst("l");              // uninitialized list,
                                   // with ext'l name "l"
int[] s_elems = {2,4,8,3};
Set s = new Set("s",s_elems);      // initialized set
                                   // (value {2,4,8,3}),
                                   // with ext'l name "s"
```

Hereafter, we will often make use of an abstract notation—which closely resembles that of Prolog—to write lists in a more convenient way. Specifically, $[e_1,e_2,\ldots,e_n]$ is used to denote the list containing $n$ elements $e_1$, $e_2$, …, $e_n$, while $[]$ is used to denote the *empty list*. Moreover, $[e_1,e_2,\ldots,e_n|R]$, where $R$ is a list, is used to denote a list containing the $n$ elements $e_1$, $e_2$, …, $e_n$, plus elements in $R$. In particular, if $R$ is uninitialized, $[e_1,e_2,\ldots,e_n \mid R]$ represents an "unbounded" list, with elements $e_1$, …, $e_n$ and an unknown part $R$. Similar abstract notation will be introduced also to represent sets (with square brackets replaced by curly brackets).

Elements of a list or of a set can be also logical variables (or lists or sets), possibly uninitialized. For example, the following declarations

```
Lvar x = new Lvar();
Object[] pl_elems = {new Integer(1),x};
Lst pl = new Lst(pl_elems);
```

create the list `pl` with value `[1,x]`, where `x` is an uninitialized logical variable. A list (resp., set) that contains some elements which are uninitialized logical variables (or lists, or sets) is said a *partially specified list (set)*. Note that in a partially specified set the cardinality is not completely determined. For example, the partially specified set `{1,x}` has cardinality 1 or 2 depending on whether `x` will get value `1` or different from `1`, respectively (actually, each partially specified set/list denotes a possibly infinite collection of different sets/lists, that is all sets/lists which can be obtained by assigning admissible values to the uninitialized variables).

## 3.3 List and set constructor expressions

A list (resp., set) can be also obtained as the result of evaluating a list (resp., set) constructor expression.

Let $e$ be an *Lvar expression* (i.e. an expression returning a `Lvar` value), `l` and `m` be *list expressions* (i.e., expressions returning a list object or a logical variable whose value is a list object), and `x` be an uninstantiated logical variable. A *list constructor* is an expression of one of the forms:

| | | |
|---|---|---|
| $(i)$ | `l.ins1(`$e$`)` | (head element insertion) |
| $(ii)$ | `l.insn(`$e$`)` | (tail element insertion) |
| $(iii)$ | `l.ext1(x)` | (head element removal) |
| $(iv)$ | `l.extn(x)` | (tail element removal) |

Expressions $(i)$ and $(ii)$ denote the list obtained by adding $val(e)$ as the first and the last element of the list `l`, respectively, whereas expressions $(iii)$ and $(iv)$ denote the list obtained by removing from `l` the first and the last element, respectively. Evaluation of expressions $(iii)$ and $(iv)$ also causes the value of the removed element to become the value of `x`. [1] It is important to notice that these 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 (the same will hold for sets, too).

Constructor expressions for sets are simpler than those for lists. In fact, in lists we can distinguish between the first (the *head*) and the last (the *tail*) element of a list, while in sets the order of elements is immaterial. Moreover, only the element insertion method is provided since element extraction may involve a nondeterministic selection of the element to be extracted that is better handled using set constraints (see Section 4). Let $e$ be an `Lvar` expression and `s` be a *set expression* (i.e., an expression returning a set object or a logical variable whose value is a set object). A *set constructor* is an expression of the form:

`s.ins(`$e$`)` (element insertion)

which denotes the set obtained by adding $val(e)$ to `s` (i.e., `s` $\cup \{val(e)\}$).

Set/List insertion and extraction methods can be concatenated (left associative). In fact these methods always return a `Set/Lst` object, and the returned object can be used as the invocation object as well.

[1]Extraction methods for lists require that the invocation list `l` is initialized and that `x` is not initialized. If one of these conditions is not respected an exception is raised (namely, `NotInitVarException` and `InitLvarException`, respectively). Moreover, if `l` is the empty list, a `EmptyLstException` exception is raised.

Using the insertion methods it is possible to build *unbounded* partially specified sets/lists, that is data structures with a certain number of (either known or unknown) elements $e_1, ..., e_n$, and an unknown "rest" part, represented by an uninitialized set/list $r$ (i.e., using the abstract notation, $\{e_1, \ldots, e_n \mid r\}$ or $[e_1, \ldots, e_n \mid r]$ for sets and lists, respectively).

EXAMPLE 3.2. *Set/List element insertion and removal*

```
Lst nil = Lst.empty;          // the empty list
Lst l1 = nil.ins1(3+2).ins1(x);
                              // the p.s. list [x,5]
                              // (x uninitialized var.)
Lst l2 = l1.ext1(y).insn(y);
                              // the p.s. list [5,x]
                              // (y uninitialized var.)
Set s1 = Set.empty.ins(1).ins('a');
                              // the set {'a',1}
Set r = new Set();            // an uninitialized set
Set s2 = r.ins(1);            // the unbounded set {1 | r}
```

Note that `s2` in the above example is a partially specified set containing one element, `1`, and an unknown part `r`; in this case, the cardinality of the denoted set has no upper bound (the lower being 1).

Special forms of the insertion and extraction methods are provided to simplify their usage. In particular, the method `ins1All(a)`, applied to a list `l`, where `a` is an array of elements of a type $t$, returns a list obtained from `l` by adding all elements of `a` as the head elements of `l`, respecting the order they have in `a`. Similarly, `insAll(a)`, applied to a set `s`, is used to insert more than one element at a time into `s`. In addition, an alternative form is provided for specifying the value for a set or list object. When creating the object it is possible to specify the limits $l$ and $u$ of an interval $[l, u]$ of integers: the elements of the interval will be the elements of the set/list (if $u < l$ the set/list is empty).

A number of utility methods are also provided by classes `Lvar`, `Lst`, and `Set`. These methods are used, for example, to print a set/list object, to know whether a logical variable is initialized or not, to get the external name associated with a `Lvar`, `Lst`, or `Set` object, and so on.

Logical variables, sets, and lists are used mainly in conjunction with constraints. Constraints are addressed in more details in the next section.

# 4. PROGRAMMING WITH (SET) CONSTRAINTS

Basic set-theoretical operations, as well as equalities and inequalities, are dealt with as *constraints* in JSetL. The evaluation of expressions containing such operations is carried on in the context of the current collection of active constraints $\mathcal{C}$ (the global *constraint store*) using domain specific constraint solvers. Those parts of these expressions, usually involving one or more uninitialized variables, which cannot be completely solved are added to the constraint store and will be used to narrow the set of possible values that can be assigned to the uninitialized variables.

## 4.1 JSetL constraints

The JSetL *constraint domain* is the $\mathcal{SET}$ domain defined in [6], extended with a few new constraints over lists and integers. An *atomic constraint* of this domain is an expression of one of the forms:

- $e_1.op\,(e_2)$
- $e_1.op\,(e_2, e_3)$

where *op* is one of a collection of predefined methods provided by classes `Lvar`, `Lst` and `Set`, and $e_1$, $e_2$ and $e_3$ are expressions whose type depends on *op*. More precisely, let $l, r$ be `Lvar` expressions, $s, s_1, s_2, s_3$ set expressions, $l_1, l_2$ list expressions, and $i_1, i_2$ integer expressions. JSetL provides the following atomic constraints:
$l.\mathsf{eq}\,(r)$ (equality) for comparing `Lvar` values;
$l.\mathsf{in}\,(s)$ (membership), $s_1.\mathsf{subset}\,(s_2)$ (subset), $s_1.\mathsf{union}\,(s_2, s_3)$ (union, i.e. $val(s_1) = val(s_2) \cup val(s_3)$)), $s_1.\mathsf{inters}\,(s_2, s_3)$ (intersection), and a few other basic set-theoretic operations, for dealing with sets;
$i_1.\mathsf{le}\,(i_2)$ ($\leq$), $i_1.\mathsf{ge}\,(i_2)$ ($\geq$), ..., for comparing integer values.
Moreover, for most of them, also their negative counterparts are provided: $l.\mathsf{neq}\,(r)$ (inequality), $l.\mathsf{nin}\,(s)$ (not membership), $s_1.\mathsf{nsubset}\,(s_2)$ (not subset), and so on.

If an expression $e_i$ of an atomic constraint $e_1.op\,(e_2, e_3)$ (or $e_1.op\,(e_2)$) is evaluated to a value of the wrong type, a suitable exception is raised by the constraint solver.

A *constraint* is either an atomic constraint or (recursively) the conjunction of two or more atomic constraints $c_1$, $c_2$,..., $c_n$:

- $c_1.\mathsf{and}\,(c_2)$ ... $.\mathsf{and}\,(c_n)$

EXAMPLE 4.1. *JSetL constraints*

*Let* `x`, `y`, `z` *be logical variables and* `r`, `s`, *and* `t` *be sets.*

```
r.eq(s);                        // equality between sets
t.union(r,s);                   // t = r ∪ s
x.eq(y).and(x.eq(3)).and(y.neq(z))
                                // x = y ∧ x = 3 ∧ y ≠ z
```

Note that solving an equality constraint implies the ability to solve a *set unification* problem (cf., e.g., [7]). Set unification of two (possibly partially specified) sets $s$ and $r$ means finding an assignment of values to uninitialized variables occurring in them (if any), such that $s$ and $r$ become equal in the underlying set theory. Intuitively, in any reasonable set theory, two sets are equal if they have the same elements, disregarding their order and possible repetitions. Thus, for instance, the set unification problem

$$\{x, y\} = \{1, 2\}$$

where $x$ and $y$ are uninitialized logical variables, admits two solutions: assign 1 to $x$ and 2 to $y$, or assign 2 to $x$ and 1 to $y$.

## 4.2 Constraint solving

The approach adopted for constraint solving in JSetL is the one developed for CLP($\mathcal{SET}$) [6]. Logically, the constraint store is a conjunction of atomic formulae built using basic set-theoretic operators, along with equality and inequalitie. Satisfiability is checked in a set-theoretic domain, using a suitable constraint solver which tries to reduce any conjunction of atomic constraints to a simplified form—the *solved form*—which is guaranteed to be satisfiable. The success of this reduction process allows one to conclude the satisfiability of the original collection of constraints. Conversely, the detection of a failure (logically, the reduction to $\mathsf{false}$) implies the unsatisfiability of the original constraints. Solved form constraints are left in the current constraint store and passed ahead to the new state. A successful computation, therefore, may terminate with a not empty collection of solved form constraints in the final constraint store.

The JSetL constraint solver basically implements in Java the constraint solver of CLP($\mathcal{SET}$), extended with simple constraints over integers. Suitable restrictions, however, are imposed on the latter so that they can be always eliminated. Namely, expressions $e_1$ and $e_2$ in $e_1.op\,(e_2)$, where *op* is an integer comparison operator, can not contain any uninstantiated variable when they are evaluated; otherwise an exception is raised.[2] Therefore, since the constraint solver of CLP($\mathcal{SET}$) is proved to be complete (as well as correct and terminating) [6], the same holds also for the JSetL constraint solver.

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. We say that the invocation of a method, calling (directly or indirectly) the `solve` method, terminates with *failure* if its execution causes the `Failure` exception to be raised; otherwise we say that it terminates with *success*. 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.

To find a solution, the constraint solver tries to reduce the atomic constraints in the constraint store to a simplified form - called the *solved form* (see [6]). This reduction is *nondeterministic*. Nondeterminism is handled through choice points and backtracking. Once the constraint reduction process detects a failure, the computation backtracks to the most recently created choice point (chronological backtracking). If no choice point is left open the whole reduction process fails (i.e., the `Failure` exception is generated).

EXAMPLE 4.2. *Constraint solving*

*Let* `s` *be the set* `{x,y,z}`, *where* `x`, `y`, `z` *are uninitialized logical variables, and* `r` *be the set* `{1,2,3}`.

```
Solver.add(r.eq(s));      // set unification r = s
Solver.add(x.neq(1));     // x ≠ 1
Solver.solve();           // calling the constraint solver
x.output();
```

[2] Actually, the new version of the CLP($\mathcal{SET}$) solver [4], which integrates the $\mathcal{SET}$ solver with an efficient solver over *finite domains* ($\mathcal{FD}$), is able to deal with basic operations over integers as constraints with almost no restriction on the instantiation of expressions that can occur in them. Following the same approach, and extending the JSetL constraint solver accordingly, it would allow us to deal with arithmetic constraints with no restrictions also in JSetL.

*`x.output()` prints the (external) name of the variable `x` followed by its value (if any; otherwise, followed by '_'). Therefore the output generated by this code fragment is:*

```
x = 2
```

In the above example, the value for `x` is computed through backtracking. Solving the set unification problem $\{\mathtt{x}, \mathtt{y}, \mathtt{z}\} = \{1, 2, 3\}$ nondeterministically returns one of the six different solutions:

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

and so on. Assuming the first computed value for `x` is `1`, then the other constraint, `x.neq(1)`, turns out to be not satisfied. Thus, backtracking forces the solver to find another solution for `x`, namely `x = 2`. In this case, the conjunction of the two given constraints is satisfied, and the invocation of the `solve` method terminates successfully. If later on a new constraint, e.g., `[x]` $\neq$ `[2]`, is added to the constraint store, and the constraint solver is called again:

```
Solver.add(Lst.empty.ins1(x).neq(Lst.empty.ins1(2)));
                                        // [x] ≠ [2]
Solver.solve();
x.output();
```

the choice points left open by the previous call to the solver are still open and they are explored by the new invocation. The output generated at the end of the computation of this new fragment of code is therefore:

```
x = 3
```

Note that every time the `solve` method is invoked it does not restart solving the constraint from the beginning but it restart from the point reached by the last invocation to `solve`.

At the end of the computation the constraint store may contain solved form constraints. To print these constraints, other than equality constraints, one can use the static method `showStore()` of class `Solver` (actually this method allows to visualize the content of the constraint store at any moment during the computation).

JSetL provides also a more convenient way to introduce more than one constraint at a time, through the `forall` method. Let `x` be an uninitialized variable, $S$ a set expression which is evaluated to a bounded set, $C$ a constraint containing `x`, and $C_s$ the constraint obtained from $C$ by replacing all occurrences of `x` with element $s$ of $S$. The statement

`Solver.forall(x,`$S$`,`$C$`)`

adds the constraint $C_s$ to the constraint store, for each element $s$ of $S$. Logically, `forall(x,`$S$`,`$C$`)` is the so-called Restricted Universal Quantifier (cf., e.g., [6]): $\forall x((x \in S) \rightarrow C)$ (see the sample program in Section 2 for a simple use of `forall`).

It is common also to allow *local* variables $y_1, \ldots, y_n$ in $C$, which are created as new for each element of the set (logically, $\forall x((x \in S) \rightarrow \exists y_1, \ldots, y_n(C))$ that is $y_1, \ldots, y_n$ are existentially quantified variables). For this purpose, JSetL provides also the method

`Solver.forall(x,`$S$`,`$Y$`,`$C$`)`

where `x`, $S$, and $C$ are the same as in the simpler `forall` method, while $Y$ is an array of all the local uninitialized logical variables $y_1, \ldots, y_n$ occurring in $C$ (see Example 4.4).

## 4.3 Programming with constraints

Let us see how the solver works on a number of simple examples, possibly involving also constraints in the computed result.

EXAMPLE 4.3. *In difference*

*Check whether an element `x` belongs to the difference between two sets, `s1` and `s2` (i.e., `x` $\in$ `s1\s2`) .*

```
public static void in_difference(Lvar x, Set s1, Set s2)
throws Failure
    {
    Solver.add(x.in(s1));
    Solver.add(x.nin(s2));
    Solver.solve();
    }
```

*If the following code fragment is executed (for instance, in the **main** method)*

```
in_difference(x,s,r);
x.output();
Solver.showStore();
```

*and `s` and `r` are the sets `{1,2}` and `{1,3}`, respectively, and `x` is an uninitialized variable, the output generated is:*

```
x = 2
Store:  empty
```

*Conversely, if `s` is an uninitialized set, then executing the same program fragment as above, will produce the following output*

```
x = unknown
s = {x | Set_1}
Store:  x.neq(1) x.neq(3)
```

*which is read as: `s` can be any set containing the element `x` and `x` must be different from `1` and `3`.*

The ability to solve constraints disregarding the fact logical variables occurring in them are initialized or not allows methods involving constraints to be used in a quite flexible way, e.g., using the same method both for testing and computing solutions This flexibility strongly contributes to support a declarative programming style.

EXAMPLE 4.4. *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`.*

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

*Let `sample_set` be the set `{[1,3],[1,2],[2,3]}`. The following fragment of code tests whether `sample_set` is composed only of pairs and prints a message ``All pairs'' or ``Not all pairs'' depending on the result of the test.*

```
boolean res = true;
try {
    all_pairs(sample_set);
    }
catch(Failure e)
    {res = false;}
if (res) System.out.print("All pairs");
else System.out.print("Not all pairs");
```

Example 4.4 shows also how a statement, namely the invocation all_pairs(sample_set), can be used, in a sense, as a condition. In fact, if execution of the statement fails (i.e., not all elements in the given set are pairs), then an exception Failure is raised and the associated exception handler executed. The latter can easily set a boolean variable to be used in the next **if** statement. Thus, if the statement terminates with success then a true value is returned (in res); otherwise, the statement terminates with failure and a false value is returned. This is analogous to the use of statements as expressions found in some languages, such as Alma-0 [1] and SINGLETON [15].

Constraints and other JSetL facilities can be used in conjunction with the usual control structures of Java. This situation is illustrated by the following example.

EXAMPLE 4.5. *Symmetrical list*

*Check whether a list* l *is symmetrical or not.*

```
public static boolean symmetrical(Lst l)
throws Failure
    {
    try {
        while(l.size()>1)
            {
            Lvar z1 = new Lvar();
            Lvar z2 = new Lvar();
            Lst r = l.ext1(z1).extn(z2);
                // extract the first and last element of l
            Solver.add(z1.eq(z2));
                // the first and the last elem's
                // must be equal
            Solver.solve();
            l = r;
                // continue with the rest of l
            }
        return true;
        }
    catch(Failure e)
        {
        return false;
        }
    }
```

*If, for example,* l *is* ['r','a','d','a','r'] *the value returned by* symmetrical *is* **true**. *List* l *can contain also some unknown values. For example, with* l = [x,1,3,y,2], x *and* y *uninitialized variables, invocation of the* symmetrical *method returns* **true** *and as a side-effect it initializes variables* x *and* y *to* 2 *and* 1, *respectively. Note that within the* **while** *loop we use an assignment between two logical variables,* l = r*: this forces* l *at the next iteration to be replaced by the new (shorter) list* r.

## 5. NONDETERMINISM

A computation in JSetL can be nondeterministic, though nondeterminism in JSetL is confined to constraint solving. Precisely, like in SINGLETON, nondeterminism is mainly supported by set operations. As a matter of fact, 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 $x = 1$, $x = 2$, $x = 3$. Since the semantics of set operations is usually well understood and quite "intuitive", making nondeterministic programming the same as programming with sets can contribute to make the (not trivial) notion of nondeterminism easier to understand and to use.

Nondeterminism is another key feature of a programming language to support declarative programming. A simple way to exploit nondeterminism in JSetL is through the use of the Setof method. 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. Then the collected set can be processed, e.g., by iterating over all its elements using the forall method.

EXAMPLE 5.1. *All solutions*

*Compute the set of all subsets (i.e., 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'}}.

As a more comprehensive example, using nondeterminism and set constraints, we show a possible JSetL solution to the well-known combinatorial problem of the coloring of a map.

EXAMPLE 5.2. *Coloring of a map*

*Given a map of $n$ regions $r_1,\ldots,r_n$ and a set of $m$ colors $c_1,\ldots c_m$ find an assignment of colors to regions such that neighboring regions have different colors.*

*The regions are represented by a set of $n$ uninitialized logical variables and the colors by a set of $m$ constant values (e.g.,* {"red","blue"}*). The map is modeled by an undirected graph and it is represented as a set whose elements are sets containing two neighboring regions. At the end of the computation each* Lvar *representing a region will be initialized with one of the given color.*

```
public static void coloring(Set regions,
                            Set map,
                            Set colors)
throws Failure
    {
    Lvar x = new Lvar();
    Set single = Set.empty.ins(x);
    Solver.add(regions.eq(colors));
    Solver.forall(x,colors,(single.nin(map));
    Solver.solve();
    return;
    }
```

*The solution uses a pure "generate & test" approach. The* regions = colors *constraint allows us to find a valuable assignment of colors to regions. Invocation of the* forall *method allows us to test whether the constraint* {x} $\notin$ map *holds for all* x *belonging to* colors. *If it holds, it means that for no pair* $\{r_i, r_j\}$ *in* map, $r_i$ *and* $r_j$ *have got the same color.*

*If* coloring *is called with* regions = {r1,r2,r3}, r1, r2, r3 *uninitialized logical variables,* map = {{r1,r2},{r2,r3}}, *and* colors = {"red", "blue"}, *the invocation terminates with success, and* r1, r2, r3 *are initialized to* "red", "blue", *and* "red", *respectively (actually, also the other solution which initializes* r1, r2, r3 *to* "blue", "red", *and* "blue", *respectively, can be computed through backtracking, if the first computed solution turns out to cause a failure).*

*Note that the set of colors can be also partially specified. For example, if* colors = {c1,"blue"}, *with* c1 *an uninitialized variable, executing* coloring *will generate the constraint:*
r1 = Lvar_1, r2 = blue, r3 = Lvar_1, Lvar_1.neq(blue).

## 6. DEFINING NEW CONSTRAINTS

Nondeterminism in JSetL is confined to constraint solving. One consequence of this is that 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. For example, let us consider the following program fragment, where we assume that s is the set {0,1}, and $c_1$, $c_2$ are two constraints:

```
Solver.add(x.in(s));
Solver.solve();
if (x.value().equals(new Integer(0)) Solver.add(c1);
else Solver.add(c2);
```

If, when evaluating the **if** condition, the value of the logical variable x is 0 then the constraint $c_1$ is added to the constraint store. If, subsequently, a failure is detected, backtracking will allow to consider a different value for x, namely 1, but the **if** condition is no longer evaluated. The constraint solver will examine the constraint store again, with the new value for x but still with constraint $c_1$ added to it.

Basically the problem is caused by the fact that we cannot guarantee a tight integration between the constraint solver (which is defined in a library) and the primitive constructs of the language. This is probably the main difference between what we called the "library" approach and the approach based on the definition of a new language (or the extension of an existing one). As a matter of fact the problem illustrated by the above program fragment is easily programmed in a language such as SINGLETON where nondeterminism and logic variables are embedded in the language.

However, JSetL provides a solution to overcome this difficulty. The solution is based on the possibility to introduce user-defined new constraints. Whenever a method which the user wants to define requires some nondeterministic action embedded in a non-trivial control structure, one can define the method as a new constraint, so that its execution is completely performed in the context of constraint solving.

User-defined constraints are defined as usual Java methods except that: (*i*) they are all declared within a class named NewConstraints; (*ii*) they can use the JSetL facilities for handling nondeterminism. To make the use of these facilities simpler, we assume a special construct is provided (similar to that found for instance in Alma-0 [1]): ***either*** $S_1$ ***orelse*** $S_2$ ... ***orelse*** $S_n$, where $S_1, \ldots, S_n$ are Java statements. The logical meaning of the ***either-orelse*** construct is the disjunction $S_1 \vee, \ldots, \vee\ S_n$, while its computational interpretation is that of exploring, through backtracking, all possible alternatives $S_1, \ldots, S_n$, starting from $S_1$. Actually this construct is easily replaced by usual Java code endowed with JSetL facilities for handling nondeterminism (see Appendix A for an example). This replacement can be performed automatically through a straightforward *preprocessing* phase that takes the extended Java code of the NewConstraints class as its input and it generates the corresponding standard Java code.

Let us see how the user can define a new constraint using a simple example: a fully nondeterministic recursive definition of the classical list concatenation operation (concat). The solution, however, can be easily generalized to other cases.

```
class NewConstraints
  {
  public static void concat(Lst l1, Lst l2, Lst l3)
  throws Failure
      {
        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));
                                  // concat(l1new,l2,l3new)
          }
      return;
      }
  }
```

The method concat takes three lists as its parameters: l1, l2 and l3. l3 is the concatenation of l1 and l2. concat can be used both to check if a given concatenation of lists holds and to build any of the three lists, given any of the other two (as usual, for instance, in Prolog). Such flexibility is obtained by using unification (instead of standard assignment) and nondeterminism (instead of a deterministic **if** statement). Nondeterminism is implemented through the ***either-orelse*** construct: the first alternative states that when l1 is the empty list, l2 and l3 must be equal; the second alternative deals with the case in which the first element of l1 is x so that l3 is obtained by inserting x as the head element of the list l3new which is obtained (recursively) by concatenating the rest of l1 (i.e., l1new) with l2. The actual Java code that implements the new constraint concat (generated through preprocessing of the NewConstraints class) is shown in Appendix A.

Execution of the statement

```
Solver.add(NewConstraints.concat(l1,l2,l3))
```

causes the user-defined constraint concat to be added to the current constraint store. If, for instance, l1 is [1,2,3], l2 is [4,5], and l3 is an uninitialized list, a subsequent call to Solver.solve() will set l3 equal to [1,2,3,4,5].

As an example of the possible different usages of concat we show a completely declarative version of a method for checking the sublist relation.

EXAMPLE 6.1. *Sublist relation*

*Check whether a list* p *is a sublist of a list* s *and return the position* k *where* p *starts.*

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

public static int sublist(Lst p, Lst s)
throws Failure
    {
    Lst A = new Lst();
    Lst B = new Lst();
    prefix(A,s);
    Solver.add(NewConstraints.concat(A,p,B));
    prefix(B,s);
    Solver.solve();
    return A.size()+1;
    }
```

*where* `A.size()` *yields the number of elements of the list denoted by* `A`*. The first invocation of the* `prefix` *method in* `sublist` *allows us to compute (nondeterministically) a possible prefix* `A` *of* `s`*, while the second invocation of the* `prefix` *method is used to check whether the computed sublist* `B` *is a prefix of* `s`*. If this is not the case, a different value for* `A` *is attempted next.*

*As an example, if* `s` *is the list* `[1,1,3,1,4,5,1,2,1,1,1,4]` *and* `p` *is the list* `[1,2,1]` *the invocation of* `sublist` *will return 7 as its result.*

# 7. CONCLUSIONS AND FUTURE WORK

We have presented the main features of the JSetL library and we have shown how they can be used to write programs that exhibit a quite good declarative reading, while maintaining all the features of conventional Java programs. In particular we have described the (set) constraint handling facilities supported by our library and we have shown how constraint solving can be accomplished, and how it interacts with the usual notion of program computation. Furthermore we have shown how to exploit nondeterminism, possibly by introducing new used-defined constraints. JSetL is fully implemented in Java and is available at URL `www.math.unipr.it/~gianfr/JSetL`.

All features provided by JSetL are present also in the CLP($\mathcal{SET}$) language [6], but embedded in a CLP framework. An attempt to "export" these features outside CLP is represented by the definition of the SINGLETON language [15], a declarative language that combines most of the features considered in this paper with "traditional" features of imperative programming languages, such as the iterative control structures and the block structure of programs. SINGLETON, however, is a completely new language, with its own syntax and its own semantics. A possible side-effect of the current work on JSetL is to allow us to compare the approach followed in SINGLETON with the library based approach followed in JSetL, in order to evaluate the gain in the expressive power related to the effort needed to develop the new facilities and the easiness to use them.

As a future work the constraint solving capabilities of JSetL could be strongly enhanced by enlarging the constraint domain from that of sets to that of *finite domains.* Following [4], this enhancement could be obtained by integrating an existing constraint solver for finite domains, possibly written in Java, with the JSetL constraint solver over sets. As shown in [4] this would allow us to have, in many cases, the efficiency of the finite domain solvers, while maintaining the expressive power and flexibility of the set constraint solvers (which in turn is inherited from CLP($\mathcal{SET}$)).

On a different side, another concrete improvement could be obtained by using flexible preprocessing tools for the Java language that would allow us to develop suitable syntax extensions that would make it simpler and more natural using the JSetL facilities.

# 8. ACKNOWLEDGMENTS

The work is partially supported by MIUR projects *Automatic Aggregate—and number—Reasoning for Computing* and *CoVer—Constraint-based Verification of Reactive Systems.*

# 9. REFERENCES

[1] K.R. Apt, J. Brunekreef, V. Partington, and A. Schaerf. Alma-0: An imperative language that supports declarative programming. *ACM TOPLAS*, 20(5), 1014–1066, 1998.

[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] M.Dincbas, P.Van Hentenryck, H.Simonis, et al. The constraint logic programming CHIP. In *Proc. of the 2nd Int'l Conf. On Fifth Generation Computer Systems*, 683-702, 1988.

[9] ECLiPSe, User Manual. Tech. Rept., Imperial College, London. August 1999. Available at `www.icparc.ic.ac.uk/eclipse`.

[10] 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`).

[11] E.Hyyonen, S.DePascale, and A.Lehtola. Interval constraint satisfaction tool INC++. In *Proc. of the 5th ICTAI*, IEEE Press, 1993.

[12] ILOG Optimisation Suite - White Paper. Available at www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf.

[13] G.Smolka. The Oz programming model. In *Current Trends in Computer Science*, J. van Leeuwen, Ed., LNCS, vol. 1000, Springer-Verlag, 1995.

[14] 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.

[15] G.Rossi. Set-based Nondeterministic Declarative Programming in SINGLETON. In *11th Int.l Workshop on Functional and (constraint) Logic Programming*, Electronic Notes in Theoretical Computer Science, Vol. 76, Elsevier Science B. V., 17 pages, 2002.

[16] I.Shvetsov, V.Telerman, and D.Ushakov. NeMo+: Object-oriented constraint programming environment based on subdefinite models. In *Artificial Intelligence and Symbolic Mathematical Computations* (G.Smolka, ed.), LNCS 1330, Springer-Verlag, 534-548.

[17] Neng-Fa Zhou. DJ: Declarative Java, Version 0.5, User's manual. Kyushu Institute of Tecnology, 1999. Available at www.cad.mse.kyutech.ac.jp/people/zhou/dj.htlm.

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

# APPENDIX

## A. IMPLEMENTING USER-DEFINED CONSTRAINTS

In this section we present the actual code used to implement the user-defined constraint concat.

```
class NewConstraints
  {
  public static StoreElem concat(Lst l1, Lst l2, Lst l3)
      {
      StoreElem s = new StoreElem(n,l1,l2,l3);
      return s;
      }

  protected static void user_code(int c, StoreElem s)
  throws Failure
      {
      switch(c)
          { ...
          case n:  concat(s); break;
          ...}
      return s;
      }

  public static void concat(StoreElem s)
  throws Failure
      {
      Lst l1 = (Lst)s.arg1;
      Lst l2 = (Lst)s.arg2;
      Lst l3 =(Lst)s.arg3;
      switch(s.caseControl)
          {
          case 0:
              add_ChoicePoint(s);
              add(l1.eq(Lst.empty));
              add(l2.eq(l3));
              return;
          case 1:
              Lvar x = new Lvar();
              Lst l1new = new Lst();
              Lst l3new = new Lst();
              add(l1.eq(l1new.ins1(x)));
                                    // l1 = [x | l1new]
              add(l3.eq(l3new.ins1(x)));
                                    // l3 = [x | l3new]
              add(concat(l1new,l2,l3new));
                                    // concat(l1new,l2,l3new)
              return;
          }
      }
  }
```

The first definition of the concat method creates a new instance of the class StoreElem which is used to store the new constraint within the constraint store. The instance contains all parameters for the concat method, along with an integer $n$ which will be used by the solver to uniquely identify the new constraint. The user_code method is used to associate each internal code with the corresponding method that implements the user-defined constraint (namely, the second definition of the concat method in this example). The control expression of the **switch** statement is the caseControl attribute of the constraint store s associated with concat (default value: 0). Each case block, but the last one, creates a choice point and adds it to the stack of the alternatives by executing the statement add_ChoicePoint(s); then the remaining code of the case block adds the constraints necessary to compute one of the possible solutions.