JavaSet: Declarative Programming in Java with Sets

Elisabetta Poleo and Gianfranco Rossi

Dip. di Matematica, Università di Parma, Dip. di Matematica, Via M. D'Azeglio 85/A, 43100 Parma (Italy). `gianfr@prmat.math.unipr.it`

Abstract. In this paper we present a Java library—called JAVASET—that offers a number of facilities to support declarative programming like those usually found in CLP 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 JAVASET and it shows how these features can be exploited to write in Java declarative solutions to a number of simple problems in a CLP style.

1 Introduction

Many different constraint solvers have been designed in the last twenty years, devoted to different constraint domains with different features, and using different implementation techniques. Most of them have been proposed in the context of Constraint Logic Programming (CLP). Among them we can mention Prolog III and IV [5,16], CHIP [10,19], CLP(R) [14], GNU Prolog (formerly clp(FD)) [7,3], CLP(SET) [8] and ECLIPSE [11].

On the other hand, from the beginning of '90ties, researchers have realized that it can be convenient to have constraint solvers embedded in a more conventional programming setting (in particular an object-oriented one), in which one can accommodate the fundamental capabilities of CLP as well as those constructs for programming and software structuring that are typical of conventional programming languages. As a matter of fact, most real-world software development is done using conventional programming languages, in particular, object-oriented languages such as C++ and Java.

Among the proposals that move along these lines one of the best known is that of the ILOG Solver [15,13]. 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. Similar proposals are those of INC++ [12], NeMo+ [18], and JSolver [4], the last one based on the Java language instead of on C++.

In all these proposals the constraint solvers are viewed as *libraries* of the host language, more or less integrated with the language itself. A different approach for allowing constraints in a conventional programming language consists in defining a new programming language, or extending an existing one, in such a way constraints are viewed as "first-class citizens" of the language itself. This is the solution adopted for instance in the languages Alma-0 [2], Singleton [17], and DJ (Declarative Java) [20,21].

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 the facilities offered by the constraint solver, 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.

The work presented in this paper is another proposal following the OO library approach: we endow an OO language, namely Java, with facilities for defining and manipulating constraints, by providing them as a library—called JAVASET. What is peculiar in our proposal, however, is the kind of data abstractions and constraints that the library provides, and the programming style that these facilities support. Specifically, some notable features of JavaSet 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).

These facilities provide a valuable support to *declarative programming*. In particular the constraint solver allows complex (set) expressions to be checked for satisfiability on a specific domain, disregarding the order in which they are encountered and the instantiation of variables occurring in them. Moreover, the use of partially specified set 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, in particular for combinatorial problems.

All the features listed above for JavaSet are present also in the CLP($\mathcal{SET}$) language [8], but embedded in a CLP framework. An attempt to "export" these features outside CLP is represented by the definition of the Singleton language [17], 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. One of the aims of this work is to allow us to compare the approach followed in Singleton with the library based approach followed in JavaSet, 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. Actually, the debate about pros and cons of the two approaches is still largely open.

2 An informal introduction to programming with JavaSet

First of all we show a simple example of a Java program using JavaSet 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.

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 a set `s` of integers. Observe that the proposed implementation does not take care of execution efficiency. Indeed, JavaSet 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(); }  }
```

`x` and `y` 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 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

JAVASET 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.

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). 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 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.

A *list* is a finite (possibly empty) sequence of arbitrary Lvar values (i.e., the *elements* of the list). In JAVASET 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.* 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.

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, $\ldots$, e_n, while $[\,]$ is used to denote the *empty list*. Moreover, $[e_1, e_2, \ldots, e_n \mid R]$, where R is a list, is used to denote a list containing the n elements e_1, e_2, $\ldots$, e_n, plus elements in R. In particular, if R is uninitialized, $[e_1, e_2, \ldots, e_n \mid R]$ represents an "unbound" list, with elements e_1, $\ldots$, e_n and an unknown part R. Similar abstract notation will be introduced also to represent sets (with square brackets replaced by curly brackets).

A *set* is a finite (possibly empty) collection of arbitrary `Lvar` values (i.e., the *elements* of the list). While in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. In JavaSet 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*. Also, a set can be either initialized or uninitialized.

Example 1. `Lvar`*,* `Lst`*, and* `Set` *definitions*

```
Lvar x = new Lvar();            // uninitialized l. var.
Lvar y = new Lvar("y",'a');     // initialized l. var. (value 'a'); ext'l name "y"
Lvar t = new Lvar(x);           // uninitialized l. var.;  same as variable x
Lst l = new Lst("l");           // uninitialized list; 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}); ext'l name "s"
```

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 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).

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) (*iii*) `l.ext1(x)` (head element removal)
(*ii*) `l.insn(`e`)` (tail element insertion) (*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]

[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.

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 non-deterministic 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., $\texttt{s} \cup \{val(e)\}$).

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.

Using the insertion methods it is also 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 2. *Set/List element insertion and removal*

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

Basic set-theoretical operations, as well as equalities and disequalities, are dealt with as *constraints* in JavaSet. 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.

The approach adopted for constraint solving in JAVASET is the one developed for CLP($\mathcal{SET}$)[8]. Logically, the constraint store is a conjunction of atomic formulae built using basic set-theoretic operators, along with equality and disequality. Satisfiability is checked in a set-theoretic domain, using a suitable constraint solver which tries to reduce any conjunction of constraints to a simplified form—the *solved form*—which can be easily tested for satisfiability. 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 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.

An *atomic constraint* in JAVASET is an expression of one of the forms:

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

where e_1 is either a `Lvar`, a `Lst` or a `Set` expression, e_2 and e_3 can be `Lvar`, `Lst` or `Set` expressions, a primitive type value or any class object provided of an `equal` method. *op* is one of a collection of predefined operators that implement basic operations on sets, such as: equality, membership, (strict) inclusion, union, disjunction, intersection, set difference, and, for most of them, also their negative counterparts. In particular, set equality turns out to be dealt with as a *set unification* problem [9]. A *constraint* is the conjunction of two or more atomic constraints v_1, v_2,... ,v_n:

$$v_1.\mathbf{and}(v_2)\ \ldots\ .\mathbf{and}(v_n)$$

Example 3. *Set 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 ∧ x ≠ z
```

A constraint C can be added to the constraint store by calling the `add` method of the `Solver` class

`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 [8]). 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. *Constraint solving*
Let `s` *be the set* `{x,y,z}`*, where* `x`*,* `y`*, and* `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();
```

`x.output()` *prints the (external) name of the variable* `x` *followed by its value (if any; otherwise, followed by* `'_'`*). Hence, the output generated by this code fragment is:* `x = 2`.

The value for `x` is computed through backtracking; as a matter of fact, the first value for `x` computed by solving `r.eq(s)` is (likely to be) `1`, which however does not satisfy the other constraint `x.neq(1)`. 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` methods 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, the choice points left open by the previous call to the solver are still open and they are explored by the new invocation.

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

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). Let us see how the solver works on a simple example involving also negative constraints in the computed result.

Example 5 *Programming with constraints.*
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)); }
```

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

```
in_difference(v,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 = _
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.

A convenient way to introduce more than one constraint at a time is by using the `forall` method. Let `x` be an uninitialized variable, S a set expression which is evaluated to an initialized 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: $\forall x((x \in S) \rightarrow C)$ (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, JAVASET 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 a list of uninitialized logical variables.

Example 6 *Using the* `forall` *method.*
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();
      Lst Y = new Lst(Lst.empty.ins1(x2).ins1(x1));
      Lvar x = new Lvar();
      Solver.forall(x,s,Y,x.eq(Lst.empty.ins1(x2).ins1(x1)));
      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 6 shows also how a statement, namely the call `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 [17].

A computation in JAVASET can be nondeterministic, though nondeterminism in JAVASET 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 JAVASET 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 7 *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));
      Solver.setof(r);
      return r;  }
```

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

Finally we show the application of JAVASET to a more complex problem, the well-known combinatorial problem of the coloring of a map.

Example 8 *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();
      Solver.add(regions.eq(colors));
      Solver.forall(x,colors,(Set.empty.ins(x)).nin(map));
      Solver.solve();
      return; }
```

The solution uses a pure "generate & test" approach. The `regions` $=$ `colors` *constraint allows to find a valuable assignment of colors to regions. Invocation of the* `forall` *method allows 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"`.

5 Defining new constraints

Nondeterminism in JAVASET is confined to constraint solving. One consequence of this is that the value of a logical variable computed in a nondeterministic way (hence, within the constraint solver), is no longer "sensible" to backtracking once it is used outside constraint solving. 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 == 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.

The problem is caused by the fact 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, JAVASET 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. JAVASET provides a class, called NewConstraints, which is devoted to contain all definitions of the new constraints possibly introduced by the user (actually this task would be strongly simplified by the use of a suitable preprocessor that allows most of these details to be hidden to the user).

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. The solution can be easily generalized to other cases. First of all the user has to add the following method to the class NewConstraints:

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

where n is an integer selected by the user and greater than 100, that will be used by the solver to identify the new constraint. This method returns an instance of StoreElem, that is a constraint: hence, it associates the method concat with a new constraint, internally identified by the number n. Then the user has to add a new case block to the user_code method of class NewConstraints as follows:

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

Finally it is necessary to define a method concat that takes as its input the store element s and implements the actual constraint handling procedure for the new constraint. To exploit nondeterminism within this method, one has to adhere to some programming conventions. Let the definition of the new method to be based in general on k different nondeterministic alternatives. Then the user must provide a **switch** statement with k case blocks (numbered from 0 to $k-1$), one for each nondeterministic alternative as follows:

```
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 control expression of the **switch** statement is the `caseControl` attribute of the constraint `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.

Execution of the statement

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

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

Note that `concat` can be used both to check if a given concatenation of lists holds and to build any of the three lists, starting from any of the other two (like in the usual well-known definition of the `append` predicate in Prolog).

6 Conclusions and future work

We have presented the main features of the JavaSet 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.

JavaSet is fully implemented in Java and can be obtained—as a `.jar` file (172KB)—from the authors.

As a future work the constraint solving capabilities of JavaSet could be strongly enhanced by enlarging the constraint domain from that of sets to that of *finite domains*. Following [6], this enhancement could be obtained by integrating an existing constraint solver for finite domains, possibly written in Java, with the JavaSet constraint solver over sets. As shown in [6] 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 JavaSet facilities.

Acknowledgments The work is partially supported by MIUR project: *Automatic Aggregate—and number—Reasoning for Computing.*

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. K.R. Apt and A. Schaerf. Programming in Alma-0, or Imperative and Declarative Programming Reconciled. In *Frontiers of Combining Systems 2*, D. M. Gabbay and M. de Rijke (editors), Research Studies Press Ltd, 1-16, 2000.
3. P.Codognet and D.Diaz. Compiling constraints in CLP(FD). *Journal of Logic Programming*, 27(3), 185-226, 1996.
4. A.Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.
5. A.Colmerauer. An introduction to Prolog III. *C-ACM*, 33(7), 69-90, 1990.
6. A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *12th Int'l Workshop on Functional and (constraint) Logic Programming*, Valencia, June 2003.
7. D.Diaz and P.Codognet. A minimal extension of the Wam for CLP(FD). In *Proc. of the 10th Int'l Conference on Logic Programming*, 1993.
8. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.
9. 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 `http://www.cs.nmsu.edu/TechReports`).
10. 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.
11. ECLiPSe, User Manual. Tech. Rept., Imperial College, London. August 1999. Available at `http://www.icparc.ic.ac.uk/eclipse`.
12. E.Hyyonen, S.DePascale, and A.Lehtola. Interval constraint satisfaction tool INC++. In *Proc. of the 5th ICTAI*, IEEE Press, 1993.
13. ILOG Optimisation Suite - White Paper. Available at `http://www.ilog.com/products/optimisation/tech/optimisation/whitepaper.pdf`.
14. J.Jaffar, S.Michaylov, P.J.Stuckey, and R.H.C.Yap. The CLP($\mathcal{R}$) language and system. *ACM TOPLAS 14(3)*, 339-395, 1992.
15. Jean-Francois Puget, and Michel Leconte. Beyond the Glass Box: Constraints as Objects. In *Proc. of the 1995 Int'l Symposium on Logic Programming*, MIT press, pp. 513-527.
16. F.Benhamou et al. Le manuel de Prolog IV, PrologIA, June 1996.
17. G.Rossi. Set-based Nondeterministic Declarative Programming in SINGLETON. In *11th Int.l Workshop on Functional and (constraint) Logic Programming*, Grado (IT), June 2002.
18. 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.
19. P.Van Hentenryck, H.Simonis, and M.Dincbas. Constraint satisfaction using constraint logic programming. *Artificial Intelligence 58*, 113-159, 1992.
20. Neng-Fa Zhou. DJ: Declarative Java, Version 0.5, User's manual. Kyushu Institute of Tecnology, 1999. Available at `http://www.cad.mse.kyutech.ac.jp/people/zhou/dj.htlm`.
21. Neng-Fa Zhou. Building Java Applets by using DJ - a Java Based Constraint Language. Available at `http://www.sci.brooklyn.cuny.edu/ zhou`.