Quaderni del Dipartimento di Matematica e Informatica
Università degli Studi di Parma

Gianfranco Rossi[1], Federico Bergenti[2]

Nondeterministic Programming in Java with JSetL

29 gennaio 2013 n. 510

Il presente lavoro è stato finanziato in parte dal progetto GNCS "Specifiche insiemistiche eseguibili e loro verifica formale".

[1]Università degli Studi di Parma, Parma, Italy, gianfranco.rossi@unipr.it
[2]Università degli Studi di Parma, Parma, Italy, federico.bergenti@unipr.it

Nondeterministic Programming in Java with JSetL

Gianfranco Rossi, Federico Bergenti

Dipartimento di Matematica e Informatica
Università degli Studi di Parma
Parco Area delle Scienze 53/A, 43124 Parma, Italy
{gianfranco.rossi|federico.bergenti}@unipr.it

Abstract

In this paper, we show how nondeterministic programming techniques can be used within the Java programming language. Our proposal is to stay within a library-based approach but taking advantage of the (nondeterministic) constraint solver provided by the library JSetL to give the user the possibility to define its own nondeterministic code as new constraints. We also point out the potential synergy between nondeterminism and sets and between nondeterminism and constraint programming. We support our claims by showing concrete Java implementations of a number of different examples ranging from classical combinatorial problems to list operations, and from graph-based problems to implementing Definite Clause Grammars.

1 Introduction

Nondeterministic programming is a programming paradigm in which one can specify various alternatives (called *choice points*) for program flow. The program computation, therefore, can be described as a tree (namely, a *computation tree*) where each node represents a choice point. Basically, two forms of nondeterminism have been considered in programming languages: *don't know nondeterminism* and *don't care nondeterminism*. For don't know nondeterminism, the choice can be made arbitrarily: *each* path in the computation tree should lead to a correct outcome. For don't care nondeterminism, *some* path in the computation tree should lead to a correct outcome, while others may lead to a failure. In this case, the choice matter, but the correct one is not known at the time the choice is made. These two forms of non-determinism lead to completely different issues and different considerations. Don't care nondeterminism is present, for example, in Dijkstra's guarded command language and in Concurrent ML, in which communications may be synchronized nondeterministically. In this paper we will not address don't care nondeterminism.

Unless otherwise specified, whenever we will say "nondeterminism" we will refer to *don't know nondeterminism*.

The usual method to implement (don't know) nondeterminism is via *backtracking*. If a failure is encountered, the program computation have to revert (i.e., to backtrack) to some *open choice point*, i.e. one which have at least one alternative originally ignored, and try another unexplored path. One complication is that the system must be able to restore old program states by undoing all effects caused by partially executing a branch that eventually failed.

The problem of incorporating constructs to support nondeterminism into programming languages have been discussed at length in the past. Early references to this topic are [5], for a general overview, and [14] for an analysis of the problem in the context of functional programming languages. Logic programming languages, notably Prolog, strongly rely on nondeterminism. Their computational model is inherently nondeterministic (at each computation step, one of the clauses unifying a given goal is selected nondeterministically) and the programmer can exploit and control nondeterminism using the language features when defining its own procedures.

As regards imperative programming, however, only relatively few languages provide primitive constructs to support nondeterminism. An early example is SETL [13], a Pascal-like language endowed with sets, which provides among others a few built-in features to support backtracking (e.g., the `ok` and `fail` primitives). Also Python's `yield` mechanism—and, more generally, the coroutine mechanisms present in various programming languages—can be used as a way to explore the computation tree associated with a nondeterministic program. More recently, the programming language Alma-0 [1, 2] provides a comprehensive collection of primitive constructs to support nondeterministic programming, such as the statements `orelse`, `some`, `forall`, `commit`, for creating choice points and handling backtracking, as well as other related features such as statements as boolean expressions, generalization of equality and a new parameter passing mechanism.

A challenging possibility would be to have facilities to support (general) nondeterministic programming also in a well-assessed object-oriented imperative programming language, such as Java. Extending the language with *primitive* constructs that offer such support is, indeed, quite demanding in general. An alternative solution would be a *library-based* approach in which the new facilities are provided as external libraries, possibly written in the language itself by exploiting the language abstraction mechanisms. This approach has the undeniable advantage of having no impact on the host language and hence of being easier to be accepted by the (usually conservative) programmers. On the other hand, supporting nondeterminism requires to interact with the program control flow, and this is done in general much better from within the language, via primitive constructs, rather than *on the top* of the language itself, as it happens with the library-based approach.

At our knowledge no existing library addresses the problem of enriching an imperative (possibly O-O) language with a comprehensive collection of facilities to

support nondeterministic programming. Actually, libraries that support constraint programming, such as Choco [4], Gecode [8], JaCoP [9], and many others, very often provide mechanisms for nondeterministically compute solutions to Constraint Satisfaction Problems (CSP). While it is undeniable that constraint solving can be used to solve many of the problems usually addressed in nondeterministic programming (think for example to the ubiquitous n-queens problem), it is also true that not all such problems can be modelled as constraint satisfaction problems.

Our proposal is to stay within a library-based approach but taking advantage of the (nondeterministic) constraint solver provided by the library to give the user the possibility to define its own nondeterministic code as new constraints.

We will illustrate this possibility in the next sections with a number of simple examples using JSetL [12]. JSetL is a Java library that endows Java with a number of facilities that are intended to support declarative and constraint programming. In this paper we show how JSetL can be used to support general forms of nondeterministic programming as well. This is obtained by combining different but related facilities: set data structures (along with their relevant operations), constraint solving, logical variables, unification, and user-defined constraints. In particular, JSetL provides a nondeterministic constraint solver, using choice-points and backtracking. It also allows the user to define its own constraints, that can take advantage of the facilities for expressing and handling nondeterminism provided by the solver. Thus, the user can define his/her general nondeterministic procedures as new constraints, letting the constraint solver handle them.

The paper is organized as follows. In Section 2 we show how JSetL can support nondeterminism through the use of general forms of set data structures and set operations which are, by their nature, inherently nondeterministic. Section 3 shows how various problems can be modelled as CSP and solved using JSetL's constraints and constraint solving procedures, exploiting the *labelling* mechanism for nondeterministically assigning domain values to variables involved in the CSP. Section 4 briefly introduces general nondeterministic control structures and the relevant language constructs. In Section 5 we show how different nondeterministic control structures can be implemented in Java using the facilities for defining new constraints provided by JSetL. Section 6 discusses multiple uses of constraint methods, that is the possibility to use the same methods both for testing and computing solutions. Finally, in Section 7 we show a more complete example of application of the facilities offered by JSetL to support nondeterministic control structures: the implementation of Definite Clause Grammars.

2 Sets and Nondeterminism

Nondeterminism is strongly related to the notion of set and set operations. According to [16] "The abstract character of a set makes it an essentially nondeterministic structure ...". "Nondeterministic operations are related to sets in that we think of them as capable of returning any element from some set of possible results.". "the

paradigmatic example of a nondeterministic operation is that of an arbitrary choice among the elements of some sets". Relations between sets and nondeterminism are pointed out also in [11] where nondeterminism is analyzed in the context of set theoretic formal specifications.

Availability of suitable set data abstractions, therefore, represents a first viable solution to the problem of expressing nondeterminism within a programming language.

Let us illustrate this solution with a couple of simple examples written using JSetL. First we need to briefly introduce the main features provided by JSetL to support set data abstractions.

Basically, sets are created in JSetL as instances of the class `LSet`. Elements of a `LSet` object can be of any type, including other `LSet` objects (i.e., *nested* sets are allowed). Moreover, sets denoted by `LSet` (also referred to as *logical sets*) can be *partially specified*, i.e., they can contain unknown elements, as well as unknown parts of the set. Single unknown elements are represented by unbound *logical variables* (i.e., uninitialized objects of the class `LVar`), whereas unknown parts of the set are represented by unbound logical sets (i.e., uninitialized objects of the class `LSet`). Sample declarations of logical variables and sets in JSetL are shown in Example 2.1.

Example 2.1 *(Logical variables and sets in JSetL)*

- `LVar x = new LVar()` *creates an unbound logical variable* `x`
- `LSet ls1 = new LSet()` *creates an unbound logical set* `ls1`
- `LSet ls2 = LSet.empty().ins(1).ins(2).ins(3)` *creates a logical set* `ls2` *with value* {1,2,3}*; in general,* `s.ins(o)`*, where* `s` *is a logical set and* `o` *is any object, returns a new logical set whose value is obtained by adding* `o` *as an element to* `s`
- `LSet ls3 = ls1.ins(x)` *creates a partially specified logical set* `ls3` *with an unknown element* `x` *and an unknown rest* `ls1` *(also denoted* $\{x \mid ls1\}$ *using an abstract notation).*

□

JSetL provides the basic operations on this kind of sets, such as equality, inequality, membership, cardinality, union, etc., in the form of primitive constraints, similarly to what provided by the Constraint Logic Programming language CLP($\mathcal{SET}$) [6]. In particular, JSetL uses *set unification* [7] to solve equalities over partially specified sets.

Set unification, as well as other basic set-theoretical operations, may involve nondeterminism. For example, the equation $\{x, y\} = \{1, 2\}$, where x and y are unbound logical variables, admits two distinct solutions: $x = 1 \wedge y = 2$ and $x = 2 \wedge y = 1$. In JSetL, these solutions are computed nondeterministically by the constraint solver. Similarly, the predicate $x \in \{1, 2, 3\}$, where x is an unbound logical variable, admits three distinct solutions, $x = 1$, $x = 2$, and $x = 3$, which are computed nondeterministically by the constraint solver.

A JSetL *constraint solver* is an instance of the class `SolverClass`. Basically, it provides methods for adding constraints to its *constraint store* (e.g., the method `add`) and to prove satisfiability/unsatisfiability of a given constraint (methods `check` and `solve`). If `s` is a solver, Γ is the constraint stored in its constraint store (possibly empty), and `c` is a constraint, `s.check(c)` returns `false` if and only if $\Gamma \wedge$ `c` is unsatisfiable.[1]

Hereafter we show two examples where we exploit the nondeterminism embedded in set operations to provide a nondeterministic solution to two simple problems. The examples illustrate also how the non-determinism of the JSetL solver interacts with the usual features of the underlying imperative Java language.

As a first example, let us consider the problem of printing all permutations of a set of integer numbers s. The problem can be modelled as the problem of unifying a (partially specified) set of $n = |s|$ logical variables $\{x_1, \ldots, x_n\}$ with the set s, i.e., $\{x_1, \ldots, x_n\} = s$. Each solution to this problem yields an assignment of (distinct) values to variables $x_1, \ldots, x_n$ that represents a possible permutation of the integers in s.

Example 2.2 *(Permutations) Given a set of integer numbers s, print all permutations of s.*

```
public static void allPermutations(LSet s) {
    int n = s.getSize();          // the cardinality of s
    LSet r = LSet.mkLSet(n);      // r = {x1,x2,...,xn}
    solver.check(r.eq(s));        // r = s
    do {
        r.printElems(' ');
        System.out.println();
    } while (solver.nextSolution());
}
```

The invocation `LSet.mkLSet(n)` *creates a set composed of* `n` *unbound logical variables. This set is unified, through the constraint* `eq`*, with the set of* `n` *integers* `s`*. This is done by invoking the method* `check` *of the current constraint solver* `solver` *(*`solver` *is assumed to be created outside the method* `allPermutations`*). The invocation* `check(c)` *causes a viable assignment of values from* `s` *to variables in* `r` *to be computed. Values in* `r` *are then printed on the standard output by calling the method* `printElems`*.*

Calling the method `nextSolution` *allows checking whether the current constraint admits further solutions and possibly computing the next one. Thus, all possible rearrangements of the values in the given sets are computed and printed, one at a time. For example, if s is* `{1,3,5}`*, the produced output is:*

```
1 3 5
```

[1] The method `solve` differs from `check` in that the former throws an exception if the constraint is unsatisfiable, whereas the latter always returns a boolean value.

```
1 5 3
3 1 5
5 1 3
3 5 1
5 3 1
```

□

The method `nextSolution` exploits the backtracking mechanism embedded in the constraint solver: calling `nextSolution()` forces the computation to go back until the nearest open choice point is encountered. Specifically, in the above example, we exploit nondeterminism provided by set unification, as implemented by the `eq` constraint solving procedure. Solving `r.eq(s)` nondeterministically computes a solution to the set unification problem involving the two sets `r` and `s`. Each computed solution is a valid solution for the problem at hand, i.e., it is a correct permutation. The method `nextSolution` is used here to generate all possible solutions.

As another example, let us consider a simplified version of the well-known *Traveling Salesman Problem* (TSP). For this problem we can easily devise a *set based* solution in which directed graphs are represented as sets whose elements are ordered pairs containing neighboring nodes. This representation has an immediate implementation in terms of JSetL's data structures: a graph can be implemented as a set whose elements are lists of two nodes, and each node is a distinct logical variable.[2] With these assumptions, we can provide a straightforward nondeterministic solution by exploiting partially specified sets [3] and membership constraints on them.

Example 2.3 *(TSP) Given a set of nodes N and a graph G, determine whether there is a path in G starting from a source node, passing exactly once for every other node and returning in the initial node.*

```
public static LList tsp(LSet nodes, LSet edges, char start)
throws Failure {
   int n = nodes.getSize();
   //----------- path = [x0,...,xn-1]
   LList path = LList.mkList(n);
   //----------- start is the first node in path
   LVar first_node = ((LVar)path.get(0));
   solver.add(first_node.eq(start));
   //----------- forall i,j. path[i] != path[j]
   solver.add(path.allDiff());
   //----------- forall i. (path[i],path[i+1]) must be an edge
   for (int i=0; i<n-1; i++) {
      LVar ith_node = ((LVar)path.get(i)));
      LVar ith_next_node = ((LVar)path.get(i+1)));
      LList adj_nodes = (LList.empty().ins(ith_next_node).ins(ith_node));
      solver.add(edges.contains(adj_nodes));
```

[2]Note that arcs can be conveniently implemented using sets—instead of lists—if we have to deal with undirected graphs.

```
    }
    //----------- (path[n-1],path[0]) must be an edge of the graph
    LVar last_node = ((LVar)path.get(n-1));
    LList first_last_nodes = (LList.empty().ins(first_node.ins(last_node)));
    solver.add(edges.contains(first_last_nodes));
    //----------- solve constraints
    solver.solve();
    return path;
}
```

The method `add` *is used to add constraints to the constraint store of* `solver`*. In particular, adding the constraint* `path.allDiff()` *we force all elements in* `path` *(i.e., all nodes of the graph) to be pairwise distinct, while adding the membership constraints* `edges.contains(adj_nodes)` *we require all adjacent nodes in* `path` *to be connected by an edge of the graph. Finally, with the constraint* `edges.contains(first_last_nodes)` *we require that also the last node in* `path` *is connected by an edge to the initial node.*

Satisfiability of the constraints stored in the constraint store is then checked using the method `solve`*. In particular, solving the constraints* `edges.contains(adj_nodes)` *allows the program to nondeterministically generate all possible assignments of values (i.e., nodes connected by an edge) to the adjacent elements in* `path`*. If the selected assignment turns out to be not a valid one, a new assignment is computed (if it exists), via backtracking to the nearest choice point. If, on the contrary, the constraint store is evaluated to* `true`*, the values assigned to variables in* `path` *represent the solution we are looking for and the computation finally terminates.*

For example, with the graph

`{[c,e],[a,c],[c,b],[b,a],[b,e],[d,a],[e,d],[d,b]}`

and source node `a`*, the first computed TSP path is*

`[a,c,e,d,b,a]`.

□

Note that the proposed solution uses a pure *"generate & test"* approach. Interest in such solution, however, is mainly in its naturalness in expressing nondeterminism rather than in its efficiency. As a matter of fact, nondeterminism is implemented simply by operations on sets (in particular, set membership), and the nondeterministic search is completely embedded in the constraint solver. 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.

Sets also provide a natural way to collect all solutions for problems involving (large) solution spaces. In JSetL such operation can be implemented either by combining calls to `nextSolution` and Java iterative statements or, more abstractly,

by using the library method `setof`. Both techniques allow one to explore the search space of a nondeterministic computation and to collect into a set all the computed solutions for a specified logical variable `x`. Actually this represents a limited form of *intensionally defined sets*. A set s is constructed by collecting all possible values of a variable x for which a condition φ, involving x, turns out to be satisfied, i.e. $s = \{x : \varphi\}$. If φ can be represented as a JSetL constraint, the set collection can be easily performed by using `nextSolution()` or the method `setof` over this constraint.

3 CSP and Nondeterminism

Many of the problems that are naturally dealt with by nondeterministic programs—i.e., those that deal with multiple alternatives—can be viewed as Constraint Satisfaction Problems (CSP).

Formally, a CSP is defined by a set of variables v_1, v_2, ..., v_n, each with a non-empty domain d_i of possible values, and a set of constraints c_1, c_2, ..., c_m. Each constraint c_j involves some subset of the variables and specifies the allowable combinations of values for that subset. A *solution* to a CSP is an assignment of values to some or all of the variables that satisfies all the constraints.

At a very high level the typical solving process for a given CSP can be described as follows:

(i) define all the variables, domains and constraints for the problem,

(ii) check satisfiability of the constraints, and

(iii) search for one or all solutions for the problem.

Actually, the main CSP search techniques interleave various forms of search (step (iii)) with constraint propagation (step (ii)), in which inconsistent values are removed from the domain of the variables through reasoning about the constraints.

According to this approach, nondeterminism is usually confined to the last step (often indicated as the *labelling* phase). Values to be assigned to variables are picked up nondeterministically; if the selected assignment turns out to be not suitable, another alternative is then explored.

Availability of features to support constraint programming, therefore, represents another viable—though partial—solution to the problem of expressing nondeterminism within a programming language.[3] Let us illustrate this technique with a simple example written using JSetL. Most of its facilities to support constraint programming have been already presented in the previous section. Another important feature which is useful at this point is the *labeling* mechanism. Solving the constraint `s.label()`, where `s` is a collection of logical variables, allows the program to

[3]Constraint programming in the context of conventional imperative programming languages, such as Java and C++, is mainly supported via suitable libraries—see, for instance, the JSR-331 effort for defining a standard API for constraint programming in Java [10]).

nondeterministically generate an admissible assignment of values to variables in `s`, starting from the first variable in `s` and the first value in its domain (default labeling strategy in JSetL). This assignment is propagated to all the constraints in the constraint store and if none of them turns out to be unsatisfiable, then an assignment for the next variable in `s` is computed and propagated, and so on. As soon as a constraint in the store turns out to be unsatisfiable, backtracking occurs and a new assignment for the lastly assigned variable is computed. If a viable assignment for all the variables in `s` is finally found, then it represents a solution for the given CSP.

The considered example is the *n-queens problem*, a well-known combinatorial problem. We show how it can be formulated as a CSP, and how viable solutions for it can be obtained using a final nondeterministic labelling phase. The proposed solution is written using JSetL, but very similar formulations can be devised for most of the available constraint programming systems.

Example 3.1 *(n-queens) Try to place n queens on a $n \times n$ chess board such that no two queens attack each other, i.e., no two queens are placed on the same row, the same column or the same diagonal.*

```
public static void main (String[] args)
throws Failure {
   IntLSet columns = new IntLSet(0,n-1);     // columns
   Vector<IntLVar> rows = new Vector<IntLVar>();
   for(int i = 0; i < n; i++)
      rows.add(new IntLVar());               // rows
   //———————- specify domains
   for (int i = 0; i < n; i++)
      solver.add(rows.get(i).dom(0,n-1));
   //———————- add constraints
   for (int i = 0; i < n-1; i++) {
      IntLVar r_i = rows.get(i);
      for (int j = i+1; j < n; j++) {
         IntLVar r_j = rows.get(j);
         solver.add(r_i.neq(r_j));                  // r_i ≠ r_j
         solver.add(r_j.sum(j).sub(r_i).neq(i));    // r_j + j - r_i ≠ i
         solver.add(r_i.sum(j).sub(r_j).neq(i));    // r_i + j - r_j ≠ i
      }
   }
   //———————- check constraints
   solver.solve();
   //———————- generate a solution
   solver.solve(IntLVar.label(rows));
   //———————- print a solution
   for (int i = 0; i < n; i++) {
      System.out.print("\n Queen "+i+" in:  ");
      System.out.print("["+i+","+rows.get(i).toString()+"]");
   }
   return;
}
```

}

Integers in set `columns` *indicate the columns on which the queens can be placed (*`n` *is a* `static final int` *declared in the outer environment). A value v in the i-th logical variable of* `rows` *indicates a queen placed in row i at column v. The resolution of the problem consists in assigning a value among 0 and $n-1$ to each of the n logical variables in* `rows` *satisfying the constraints. The first inequality constraint states that two queens can not be placed on the same column, while the other two inequalities state that two queens can not be placed on the same diagonal.*

Solving the `label` *constraint allows the program to nondeterministically generate an admissible assignment of values (i.e., columns) to all the variables in* `rows`*. Each such assignment represents a solution for the given CSP.*

A possible solution printed by this program is:

```
Queen_0 in:  [0,1]
Queen_1 in:  [1,3]
Queen_2 in:  [2,0]
Queen_3 in:  [3,2]
```

□

The above example clearly shows that if a problem can be formulated as a CSP then we have the opportunity to use the constraint programming features possibly supported by our language to provide some form of nondeterministic solution for the problem at hand.

Unfortunately not all problems whose solutions are naturally formulated as nondeterministic algorithms are also easily modelled as CSP. There are situations in which, in particular, the variable domains are hardly reconducted to those supported by exiting CP solvers, making the programming effort to model them in terms of the existing ones too cumbersome and sometimes quite ad-hoc. On the other hand, the use of sets and set operations to model nondeterministic computations, as shown in the previous section, is not always feasible and/or convenient.

In conclusion, there are cases in which some more general programming tools to express and handle nondeterminism are required. We will face this topic in the next sections.

4 Nondeterministic Control Structures

Dealing with general nondeterministic control structures requires primarily the ability to express and handle *choice points* and *backtracking*.

This implies, first of all, that the notion of program computation is extended to allow distinguishing between computations that terminate with success and computations that terminate with failure. Basically, a computation *fails* whenever it executes, either implicitly or explicitly, a `fail` statement. Conversely, a finite, error-free computation *succeeds* if it does not fail.

In response to a failure, the computation backtracks to the last open choice point. Choice points may be created by the programmer using suitable language constructs, such as the following `orelse` statement (borrowed from [1]):

$$\texttt{either } S_1 \texttt{ orelse } S_2 \texttt{ ... orelse } S_n \texttt{ end}$$

where $S_1 \ldots S_n$ are statements, which expresses a nondeterministic choice among n statements. More precisely, the computation of the `orelse` goes as follows: statement S_1 is executed first; if, at some point of the computation (possibly beyond the end of the `orelse` statement) a failure occurs, then backtracking takes place and the computation resumes with S_2 in the state it had when entering S_1; if a new failure occurs, then the computation backtracks and it resumes with S_3, and so on; if a failure occurs after executing S_n and no other open choice points do exist, then the computation definitively fails.

Let us briefly illustrate how to deal with general nondeterministic control structures with a couple of simple examples written using a C-like pseudo-language endowed with the `orelse` statement and a few other facilities to support nondeterministic programming. In the next section we will show how the same control structures can be implemented in Java using JSetL.[4]

As a first simple example, let us consider the problem of computing and printing all possible sublists of a list l. The problem can be solved by defining a nondeterministic function that returns any of the possible different sublists of l. All sublists are then obtained by forcing the computation to backtrack until all solutions have been generated.[5] An implementation of this algorithm written in pseudo-code using the `orelse` construct is shown in Example 4.1.

Example 4.1 *(Printing all sublists—in pseudo-code) Let l, s, and p be (generic) lists. The following function nondeterministically generates the sublists of the given list l.*

```
sublist(l):
   either
      p = any prefix of l;
      return p;
   orelse
      s = sublist(l without its first element);
      return s;
   end;
```

[4] A more systematic discussion of nondeterministic control structures is outside the scope of this paper. A presentation of a minimal set of such structures, including the basic concept of *continuations*, and their usage to express a variety of search algorithms can be found, for instance, in [15].

[5] Of course, the problem could be solved also by defining a deterministic procedure using conventional features of a conventional language, such as arrays and nested loops on the indexes of the array. The resulting code, however, is likely to be more complex than that using nondeterminism (although, in this simple example, the solution is in any case not too complicated).

where the prefix of a list $[e_1, \ldots, e_n]$ *is any list* $[e_1, \ldots, e_m]$ *with* $m \leq n$, $m, n \geq 0$ *(prefixes of a list* l *can be nondeterministically generated by a function defined in a similar way to* `sublist`*).*

To compute and print all sublists of l *we need a way to force the computation to backtrack to* `sublist` *and try another open alternative created by the* `orelse` *statement, until at least one such alternative does exist. We assume our pseudo-language is endowed with the following* `all_solutions` *construct (akin to the* `FORALL` *statement of [1] and the* `Exploreall` *of [15]):*

`all_solutions` S_1 `...`S_n `end`

where S_1, ..., S_n *are statements, whose meaning is: execute* S_1, ..., S_n*; if at the end there are open choice points then fail; else continue (note that in this way, if the last open choice point is within* S_i *then all statements following* S_i *are executed again).*

Using this construct, our problem can be solved as follows:

```
all_solutions
     r = sublist(l);
     print(r);
end
```

After printing the first value for r*, the computation backtracks to* `sublist`*, which tries another open* `orelse` *alternative and computes a new value for* r*, if it exists; then* `print(r)` *is executed again.* □

Note that in this case the domain of discourse is that of lists. Trying to encode it in terms of the usual constraint domains, e.g. that of the integer numbers, though feasible in principle (one could, for example, deal with the indexes of the elements and restate the problem as a CSP over integers), may lead to rather involved programs in practice.

As another more articulated example let us consider the well-known problem of finding a path between two nodes in a directed graph. As shown in Example 2.3, a directed graph can be represented as a set of ordered pairs $[n_1, n_2]$, where n_1, n_2 are nodes, and each pair represents an arc of the graph. In this case, however, using sets and operations on sets as done in Example 2.3 is not enough to represent a nondeterministic solution for this problem. In fact, in the case of the TSP problem we know exactly the number of nodes to be visited (i.e., all nodes of the graph), so the problem can be simply stated as a collection of (nondeterministic) membership constraints over these nodes. Conversely, in the path finding problem we do not know a priori the number of nodes to be visited and we must decide at each step if we can stop or we must continue visiting the graph. That is, we need to nondeterministically choose between two possible actions and this requires more general nondeterministic control structures.

A nondeterministic solution for the path finding problem can be formulated in pseudo-code, using the `orelse` construct, as shown in Example 4.2.

Example 4.2 *(Finding a path—in pseudo-code) Let G be a directed graph and s and d two nodes of G. The problem consists in determining whether there is a path in G from s (the source node) to d (the destination node).*

```
path(G,s,d):
   either
      test([s, d] is an arc in G);
      return true;
   orelse
      test(there exists a node t in G such that
      [s, t] is an arc in G and path(G,t,d));
      return true;
   end
   return false;
```

Here we assume to use the `test(e)` *construct, where e is a boolean expression, with the following operational meaning: if e evaluates to* `true` *then continue; else fail.*[6]*. Moreover, we assume that the boolean expression "there exists a node t in G such that ..." nondeterministically binds the variable t to any node in G which is directly connected to s whenever it returns* `true` *(it can be easily implemented by a nondeterministic function using the* `orelse` *construct in a similar way to* `sublist`*).*

If, at some point in the computation, the path that has been built till that moment turns out to not lead to the destination node, i.e., the last recursive call to `path(G,t,d)` *definitively fails, then the computation backtracks to the last choice point with at least one open alternative and the computation continues from that alternative (in practice, a different path is taken into account).* □

As mentioned in Section 1, very few programming languages support the above mentioned nondeterministic constructs as *primitive* features. On the other hand, providing such features as part of a library (i.e., implementing them in some high-level language) may turn out to be not a trivial task. Our proposal is to stay within a library-based approach but taking advantage of the (nondeterministic) constraint solver provided by the library to give the user the possibility to define its own nondeterministic code as new constraints.

We will illustrate this possibility in the next sections with a number of simple examples using JSetL.

5 Implementing Nondeterministic Control Structures in JSetL

JSetL is endowed with a nondeterministic constraint solver. Nondeterminism is used to implement various set operations (e.g. set unification), as well as to perform labelling. Availability of built-in nondeterministic constraints, however, is not sufficient to obtain the general kind of nondeterminism we would like to have. Let us

[6] It is the same as the "boolean expressions as statement" feature of Alma-0 [1]

consider for example the following program fragment, where we assume that `x` and `y` are unbound logical variables, `r` is an unbound logical set, and `s1`, `s2` are logical sets which are bound to {`1`} and {`x`}, respectively:

```
Solver.solve(r.inters(s1,s2));
if (r.isEmpty()) Solver.add(y.eq(0));
else Solver.add(y.ge(1));
...
Solver.add(y.neq(0));
Solver.solve();
```

The constraint `r.inters(s1,s2)` admits two distinct solutions, namely, $\mathtt{x} \neq 1 \wedge \mathtt{r} = \{\}$ and $\mathtt{x} = 1 \wedge \mathtt{r} = \{1\}$. The constraint solver is able to compute them using nondeterminism. Assuming the first solution is computed first, the `if` statement adds the constraint $\mathtt{y} = 0$ to the constraint store. Therefore, the invocation to the solver in the last statement detects a failure. The second solution for the constraint `inters` is then taken into account, but the condition of the `if` statement is no longer evaluated. The constraint solver will examine the constraint store again, with the new value for `r`, but still with the constraint $\mathtt{y} = 0$ stored in it. Hence, it fails again.

Basically the problem is caused by the fact that nondeterminism in JSetL is confined to constraint solving: backtracking allows the computation to go back to the nearest open choice point *within* the constraint solver, but it does not allow to re-execute user program code (the `if` statement in the above example).

The solution that we propose to circumvent these difficulties is based on the possibilities offered by JSetL to introduce new user-defined constraints. Those methods that require to use nondeterminism can be defined as new JSetL constraints. Within these methods the user can exploit facilities for creating and handling choice-points. When solving these constraints the solver can explore the different alternatives using backtracking.

5.1 User-defined Constraints

User-defined constraints in JSetL are defined as part of a user class that extends the abstract class `NewConstraintsClass`. For example,

```
public class MyOps extends NewConstraintsClass {
// public and private methods implementing user-defined constraints
}
```

is intended to define a collection of new constraints implementing user defined operations.

The actual implementation of user-defined constraints requires some programming conventions to be respected, as shown in the following example.

Example 5.1 *(Implementing new constraints) Define a class* `MyOps` *which offers two new constraints* `c1(o1,o2)` *and* `c2(o3)`*, where* `o1`*,* `o2`*,* `o3` *are objects of type* `t1`*,* `t2`*, and* `t3`*, respectively.*

```
public class MyOps extends NewConstraintsClass {
   public MyOps(SolverClass currentSolver) {
       super(currentSolver);
   }

   public Constraint c1(t1 o1, t2 o2) {
           return new Constraint("c1", o1, o2);
   }
   public Constraint c2(t3 o3) {
           return new Constraint("c2", o3);
   }

   protected void user_code(Constraint c)
   throws Failure, NotDefConstraintException {
       if (c.getName().equals("c1")) c1(c);
       else if(c.getName().equals("c2")) c2(c);
       else throw new NotDefConstraintException();
   }

   private void c1(Constraint c) {
       t1 x = (t1)c.getArg(1);
       t2 y = (t2)c.getArg(2);
       //implementation of constraint c1 over objects x and y
   }

   private void c2(Constraint c) {
       t3 x = (t3)c.getArg(1);
       //implementation of constraint c2 over object x
   }
 }
```

The one-argument constructor of the class `MyOps` *initializes the field* `Solver` *of the super class* `NewConstraintsClass` *with (a reference to) the solver currently in use by the user program.*

The other public methods simply construct and return new objects of class `Constraint`*. This class implements the atomic constraint data type. All built-in constraint methods implemented by JSetL (e.g.,* `eq`*,* `neq`*,* `in`*, etc.) return an object of class* `Constraint`*, whereas the method* `add` *takes an object of this class as its parameter. Each different constraint is identified by a string name (e.g.,* `"c1"`*), which can be specified as a parameter of the constraint constructor.*

The method `user_code`*, which is defined as abstract in* `NewConstraintsClass`*, implements a "dispatcher" that associates each constraint name with the corresponding user-defined constraint method. It will be called by the solver during constraint solving.*

Finally, the private methods, such as `c1` *and* `c2`*, provide the implementation of the new constraints. These methods must, first of all, retrieve the constraint arguments, whose number and type depend on the constraint itself. We will show*

possible implementations of such methods (using nondeterminism) in Examples 5.4, 5.5, and 7.2. □

Once objects of the class containing user-defined constraints have been created, one can use these constraints as the built-in ones: user-defined constraints can be added to the constraint store using the method `add` and solved using the `SolverClass` methods for constraint solving. For example, the statements

```
MyOps myOps = new MyOps(solver);
solver.solve(myOps.c1(o1,o2));
```

where `MyOps` is the class defined in Example 5.1, create an object of type `MyOps`, called `myOps`, and then create the constraint `"c1"` over two objects `o1` and `o2` by calling `myOps.c1(o1,o2)`. This constraint is added to the constraint store of the constraint solver `solver` and immediately solved by calling the `solve` method. Solving constraint `"c1"` will cause the solver to call the concrete implementation of the method `user_code` provided by `myOps`, and consequently to execute its method `c1`.

5.2 Nondeterminism in User-defined Constraints

Let us start with a trivial sample problem: nondeterministically assign a value among 1, 2, and 3 to an integer variable x, and print it. Using pseudo-code and the `orelse` construct, we can write a solution for this problem as follows:

```
either
   x = 1;
orelse
   x = 2;
orelse
   x = 3;
end;
print(x);
```

In JSetL this problem can be solved defining a new constraint that implements the `orelse` construct—we will call it `ndAssign`. Defining nondeterministic constraints in JSetL, however, requires additional arrangements to be taken into account.

First of all note that methods defining user-defined constraints must necessarily return an object of type `Constraint`. Thus, any result possibly computed by the method must be returned through parameters. The use of unbound logical variables (i.e., objects of class `LVar`) as arguments provides a simple solution to this problem. The result of `ndAssign` will be the value bound to the logical variable `x` passed as its argument.

More generally, the use of *logical objects*, i.e., logical variables as well as logical sets and lists, is fundamental in JSetL when dealing with nondeterminism. As a matter of fact, if an object is involved in a nondeterministic computation we need the ability to restore the status it had before the last choice point whenever the

computation must backtrack and try a different alternative. In JSetL this ability is obtained by allowing the solver to save and restore the global status of all *logical objects* involved in the computation. Note that logical objects in JSetL are characterized by the fact that their value, if any, can not be changed through side-effects (e.g. by an assignment statement). Thus, saving and restoring the status of logical objects is a relatively simple task for the solver.

Hence, we will always use logical objects, in particular logical variables, for all those objects that are involved in nondeterministic computations, such as the variable `x` in `ndAssign`. Since values of logical objects can not be modified by using the usual imperative statements (e.g., the assignment) we will always use constraints to manage them. In particular, the equality constraint l`.eq(`v`)` is used to *unify* a logical variable l with a value v. If l is unbound, this simply amounts to bind x to v. Once assigned, the value v is no longer modifiable.

A possible implementation of `ndAssign` using JSetL is shown in Example 5.2.

Example 5.2 *(Defining nondeterministic constraints) Define a constraint* `ndAssign(x)` *that nondeterministically binds* `x` *to either* `1`*,* `2`*, or* `3`*.*

```
public class NdTest extends NewConstraintsClass {
   ...
   public Constraint ndAssign(LVar x) {
      return new Constraint("ndAssign",x);
   }
   ...
   private void ndAssign(Constraint c) {
      LVar x = (LVar)c.getArg(1);
      switch(c.getAlternative()) {
         case 0:
            Solver.addChoicePoint(c);
            Solver.add(x.eq(1));
            break;
         case 1:
            Solver.addChoicePoint(c);
            Solver.add(x.eq(2));
            break;
         case 2:
            Solver.add(x.eq(3));
      }
   }
}
```

The method `ndAssign` *is provided as part of a new class, named* `NdTest`*, that extends* `NewConstraintsClass` *(see Example 5.1 for completing the code of class* `NdTest`*).* `ndAssign` *implements the nondeterministic construct* `orelse` *by using the methods* `getAlternative` *and* `addChoicePoint`*. The invocation* `c.getAlternative()` *returns an integer associated with the constraint* `c` *that can be used to count the nondeterministic alternatives within this constraint. Its initial value is* 0*. Each*

time the constraint `c` *is re-considered due to backtracking, the value returned by* `c.getAlternative()` *is automatically incremented by* 1.

The invocation `Solver.addChoicePoint(c)` *adds a choice point to the alternative stack of the current solver. This allows the solver to backtrack and re-consider the constraint* `c` *if a failure occurs subsequently.* □

In order to use `ndAssign` we must first create an instance of the class `NdTest`, and then solve the new constraint by invoking either `solve` or `check` of the current solver. A possible usage of `ndAssign` is shown in Example 5.3.

Example 5.3 *(Using nondeterministic constraints) Nondeterministically select a value among* `1`*,* `2`*, and* `3`*, and print it.*

```
LVar n = new LVar();
NdTest ndTest = new NdTest(solver);
solver.check(ndTest.ndAssign(n));
System.out.println(n);
```

Executing this fragment of code will print the first value for `n` *(e.g.,* `1`*).*[7] *To obtain all possible values for* `n`*, one could use the method* `nextSolution` *as follows:*

```
solver.check(ndTest.ndAssign(n));
do{
   System.out.println(n);
} while(solver.nextSolution());
```

□

As another example let us consider the implementation of the nondeterministic function `sublist` shown in Section 4.

Note that, generally speaking, functions are replaced by relations whenever they are defined as constraints. Specifically, a function $f : d_1 \to d_2$ can be implemented by a constraint $c(x_1, x_2)$, with x_1 and x_2 ranging over the domains d_1 and d_2, respectively. This constraint defines a binary relation R over the domains d_1 and d_2: $c(x_1, x_2)$ holds iff $\langle x_1, x_2 \rangle \in R \subseteq d_1 \times d_2$.

In the example at hand, the function `sublist` of Example 4.1 will be implemented in JSetL by a new constraint `sublist(l,sl)` which states that `sl` must be a sublist of `l`. Moreover, as noted before, `l` and `sl` are conveniently represented as logical lists (rather than, for example, as instances of `Vector` or `List`). In fact, values bound to `l` and `sl` are subject to possible changes due to backtracking during the computation of `sublist`. Being logical objects, their values will be accessed only through constraints, in particular equality constraints.

[7] The same result could be obtained in JSetL by using the built-in set constraint `in`: the invocation `solver.check(n.in(s))`, where `s` is a set containing `1`, `2`, and `3`, will nondeterministically bind `n` to values in `s`.

Example 5.4 *(*`sublist` *in JSetL) Define a constraint* `sublist(l,sl)` *which is true if* `sl` *is a sublist of the list* `l`*.*

```
public Constraint sublist(LList l,LList sl) {
    return new Constraint("sublist",l,sl);
private void sublist(Constraint c) throws Failure {
    LList l    = (LList)c.getArg(1);
    LList sl   = (LList)c.getArg(2);
    switch(c.getAlternative())
    {
    case 0: Solver.addChoicePoint(c);
            Solver.add(prefix(l,sl));   // sl is any prefix of l
            break;
    case 1: LVar  x = new LVar();
            LList r = new LList();      // r is l without its
            Solver.add(l.eq(r.ins(x))); // first element
            Solver.add(sublist(r,sl));  // sl is any non-empty
            Solver.add(sl.neq(LList.empty()));  //sublist of r
    }
    return;
}
```

The method `prefix` *can be defined in JSetL in the same way shown for* `sublist`*. The rest* `r` *of the list* `l` *is obtained through unification between* `l` *and the partially specified list* `r.ins(x)` *(i.e,* `[x | r]` *using Prolog notation for lists), while the fact that the sublist of* `r` *must be non-empty is guaranteed by the JSetL constraint* `sl.neq(LList.empty())`*.* □

Finally, let us consider a little bit more complex example: the implementation of the nondeterministic procedure `path` shown in Example 4.2. To make the solution more realistic, we add to `path` a further parameter `p` that represents the path from `s` to `d`. `path` will be implemented in JSetL as a new constraint, named `"path"`, in a class that extends `NewConstraintsClass` as shown in Section 5.1. The concrete Java code of the method that implements the new constraint `"path"` is shown in Example 5.5.

Example 5.5 *(*`path` *in JSetL)*

```
public Constraint path(LSet G,LVar s,LVar d,LList p) {
   return new Constraint("path",G,s,d,p);
}
private void path(Constraint c) {
   LSet G = (LSet)c.getArg(1);          // the graph
   LVar s = (LVar)c.getArg(2);          // the source node
   LVar d = (LVar)c.getArg(3);          // the destination node
   LList p = (LList)c.getArg(4);        // the computed path
   switch(c.getAlternative())
   {
```

```
      case 0:
        Solver.addChoicePoint(c);
        LList finalArc = LList.empty().ins(d).ins(s);
        Solver.add(G.contains(finalArc));
        Solver.add(p.eq(LList.empty().ins(finalArc)));
        break;
      case 1:
        LVar t = new LVar();
        LList intermediateArc = LList.empty().ins(t).ins(s));
        LList t_dPath = new LList();
        Solver.add(G.contains(intermediateArc));
        Solver.add(path(G,t,d,t_dPath));
        Solver.add(p.eq(t_dPath.ins(intermediateArc)));
    }
    return;
  }
```

As an example, if `G` *is the graph*

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

`s` *has value* `'a'` *and* `d` *has value* `'d'`*, then solving the constraint* `path(G,s,d,p)` *nondeterministically assigns to* `p` *the three different paths:*

```
[['a','b'],['b','d']]
[['a','c'],['c','d']]
[['a','c'],['c','b'],['b','d']].
```

□

6 Multiple Uses of Constraint Methods

The use of relations in place of functions, along with the use in their implementation of a number of specific features provided by JSetL, such as logical variables and unification, have another important consequence on the usability of user-defined constraint methods.

Let us consider a function $y = f(x)$ and a possible call to it, $z = f(a)$. In JSetL one can define a constraint $c_f(x, y)$ which represents the relation $R_f = \{\langle x, y\rangle : y = f(x)\}$ and then solve the constraint $c_f(a, z)$. Solving this constraint actually amounts to check whether $\langle a, z\rangle \in R_f$, for some z. While calling $f(x)$ to compute y implies assuming x to be the input parameter and y the output, solving $c_f(x, y)$ does not make any assumption on the "direction" of their parameters. Thus one can compute y out of a given x, or, vice versa, x out of a given y, or one can test whether the relation among two given values x and y holds, or one can compute any of the pair $\langle x, y\rangle$ belonging to R_f. Hence, the same method can be used to implement different, though related, functions.

This general use of user-defined constraints in JSetL is made possible thanks to the availability of a number of different facilities to be used in the constraint implementation. Specifically,

- the use of logical variables as parameters

- the use of unification in place of equality and assignment
- the use of nondeterminism to compute multiple solutions for the same constraint.

Note that the fact that a logical variable acts as input or as an output parameter depends on the fact it is bound or not when the method is called. In particular, unbound variables can be easily used to obtain output parameters (see Example 5.2). Moreover, if the value bound to a variable is a partially specified aggregate, e.g. a logical list, then it can act simultaneously as input and as output, i.e. as an input-output parameter.

The following is an example of the possible different usages of the constraint `sublist(l1,l2)` shown in Example 5.4.

Example 6.1 *(Uses of* `sublist`*)*

- *Compute* `l2` *out of* `l1`*. Let* `l1` *be bound to the list* `[1,2,3]` *and* `l2` *be unbound. The invocation*

  `solver.add(listOps.sublist(l1,l2))`,

  where `listOps` *is an instance of the class containing* `sublist` *that extends* `NewConstraintsClass`*, will generate nondeterministically the following values for* `l2`*:*

  `[]`, `[1]`, `[1,2]`, `[1,2,3]`, `[2]`, `[2,3]`, `[3]`.

- *Compute* `l1` *out of* `l2`*. Let* `l2` *be bound to the list* `[1,2]` *and* `l1` *be unbound. The same invocation as above will generate nondeterministically the following (infinite) sequence of values for* `l1`*:*

  `[1,2|r_1]` *(which means a list containing* `1`*,* `2` *and something else),*

  `[x_1,1,2|r_1]`,

  `[x_1,x_2,1,2|r_1]`,

  and so on, where `r_1` *and* `x_1`*,* `x_2`*, . . . , are unbound internal logical objects of type* `LList` *and* `LVar`*, respectively.*

- *Compute all the sublists of* `l1` *with exactly two elements. Let* `l1` *be bound to the list* `[1,2,3]` *and* `l2` *be bound to the partially specified list* `[x,y]`*, where* `x` *and* `y` *are unbound logical variables. The same invocation as above will generate nondeterministically the following values for* `x` *and* `y`*:*

  `x = 1` *and* `y = 2` *(i.e., the list* `[1,2]`*),*

  `x = 2` *and* `y = 3` *(i.e., the list* `[2,3]`*).*

□

As another example, consider the procedure `path` shown in Example 5.5, along with the sample graph `G` considered at the end of the example, and the constraint

```
path(G,s,d,p),
```

where `s`, `d` and `p` are logical variables that represent the source node, the destination node, and the path from `s` to `d`, respectively. If `s` is bound to `'a'`, while `d` is left unbound, then solving this constraint allows us to solve a different problem from that of Example 5.5, namely, finding all nodes in `G` that are reachable from the node `'a'`. Thus, solving `path(G,s,d,p)` will nondeterministically bind `d` to the values `'b'`, `'c'`, and `'d'`.

7 Implementing DCGs

In this section we show a more complete example of application of the facilities offered by JSetL to support nondeterminism: the implementation of Definite Clause Grammars.

A Definite Clause Grammar (DCG) is a way to represent a context-free grammar as a set of first-order logic formulae in the form of definite clauses. As such, DCGs are closely related to Logic Programming, and tools for dealing with DCGs are usually provided by current Prolog systems. Given the DCG representation of a grammar one can immediately obtain a parser for the language it describes by viewing the DCG as a set of Prolog clauses and using the Prolog interpreter to execute them.

In this section we show how DCGs can be conveniently used also in the context of more conventional languages, such as Java, provided the language is equipped with a few features that are fundamental to support DCGs processing, namely (logical) lists and nondeterminism. We prove this claim by showing how DCGs can be encoded and processed in Java using JSetL.

Consider the following simple grammar of constant arithmetic expressions

$$\langle expr \rangle ::= \langle num \rangle | \langle num \rangle + \langle expr \rangle | \langle num \rangle - \langle expr \rangle$$

Assume that input to be parsed is represented as a list of numerals and symbols. For example, `[8, +, 2, -, 7]` is a valid $\langle expr \rangle$.

This grammar may be encoded in terms of first-order logic formulae in clausal form in the following way: create one predicate for each non-terminal in the grammar and define each predicate using one clause for each alternative form of the corresponding non-terminal. Each predicate takes two arguments, the first being the list representation of the input stream, and the second being instantiated to the list of input elements that remain after a complete syntactic structure has been found. As an example, the above grammar can be written as a DCG as follows (using a pure Prolog notation).

Example 7.1 *(A DCG for $\langle expr \rangle$)*

```
expr(L, Remain) :-
   num(L, Remain).
expr(L, Remain) :-
```

```
   num(L, L1), L1 = [+|L2], expr(L2, Remain).
expr(L, Remain) :-
   num(L, L1), L1 = [-|L2], expr(L2, Remain).
num(L, Remain) :-
   L = [D|Remain], number(D).
```

where $[o|L]$ *represents the list with first element* o *and rest the list* L*, and the predicate* `number(D)` *is true if* `D` *is a numeric constant.*[8] □

This grammar representation constitutes an executable Prolog program that can be immediately used as a top-down parser for the denoted language. Using this program we can prove that, for example,

```
expr([1, +, 2, -, 3], [])
```

is true (i.e., `1+2-3` is a valid arithmetic expression), while

```
expr([1, +, 2, -], [])
```

is false.

Note that clauses for the same predicate are selected nondeterministically. The Prolog interpreter first attempts to use the first clause to process the input stream list; if it fails then it backtracks and tries to use the second one, and then, if it fails again, the third one and so on, until the computation terminates successfully. For instance, having to prove `expr([1, +, 2, -, 3], [])` the first clause fails since `Remain` is instantiated to `[+, 2, -, 3]` by the call to `num(L, Remain)`, which is trivially different from `[]`. Then the second clause will be used instead.

The DCG shown in Example 7.1, that we have written as a Prolog program, can be implemented with a relatively small effort as a JSetL program as well. Each predicate corresponding to a non-terminal in the grammar is implemented as a new JSetL constraint, that is a method of a class extending the class `NewConstraintsClass`. These methods exploit the nondeterministic features provided by JSetL to support the nondeterministic choice among different clauses for the same predicate. List data structures are implemented using JSetL logical lists, that is objects of the class `LList`. In particular, partially specified lists with an unknown rest (i.e., $[o \,|\, l]$, l unbound) can be constructed using the method `ins` and accessed through unification. The complete JSetL implementation of the DCG shown above is given in Example 7.2.

Example 7.2 *(Implementing the DCG for* $\langle expr \rangle$ *in JSetL)*

```
public class ExprParser extends NewConstraintsClass {
      public ExprParser(SolverClass CurrentSolver) {
         super(CurrentSolver);
      }
```

[8] Special syntax exists in current Prolog systems that allows BNF-like specification of DCGs. For instance, the second clause of Example 7.1 can be written in Prolog (also) as `expr --> num, [+], expr`. The Prolog interpreter automatically translates this special form to the pure clausal form used in Example 7.1.

```
public Constraint expr(LList L, LList Remain) {
   return new Constraint("expr", L, Remain);
}
public Constraint num(LList L, LList Remain) {
   return new Constraint("num", L, Remain);
}
public Constraint number(LVar n) {
   return new Constraint("number", n);
}

protected void user_code(Constraint c)
throws NotDefConstraintException {
  if (c.getName().equals("expr"))
     expr(c);
  else if (c.getName().equals("num"))
     num(c);
  else if (c.getName().equals("number"))
     number(c);
  else {
     throw new NotDefConstraintException();
  }
  return;
}

private void expr(Constraint c) {
  LList L = (LList)c.getArg(1);
  LList Remain = (LList)c.getArg(2);
  switch (c.getAlternative()) {
    // expr(L, Remain) :- num(L, Remain).
    case 0: {
      Solver.addChoicePoint(c);
      Solver.add(num(L, Remain));
      break;
    }
    // expr(L, Remain) :- num(L, L1), L1 = [+|L2], expr(L2, Remain).
    case 1: {
      Solver.addChoicePoint(c);
      LList L1 = new LList();
      LList L2 = new LList();
      Solver.add(num(L, L1));
      Solver.add(L1.eq(L2.ins1('+')));
      Solver.add(expr(L2, Remain));
      break;
    }
    // expr(L, Remain) :- num(L, L1), L1 = [-|L2], expr(L2, Remain).
    case 2: {
      LList L1 = new LList();
      LList L2 = new LList();
      Solver.add(num(L, L1));
```

```
            Solver.add(L1.eq(L2.ins1('-')));
            Solver.add(expr(L2, Remain));
          }
        }
        return;
      }

      private void num(Constraint c) {
        LList L = (LList)c.getArg(1);
        LList Remain = (LList)c.getArg(2);
        LVar D = new LVar();
        Solver.add(L.eq(Remain.ins1(D)));
        Solver.add(number(D));
        return;
      }

      private void number(Constraint c) {
        LVar n = (LVar)c.getArg(1);
        if (n.getValue() instanceof Integer ||
            n.getValue() instanceof Double)
              return;
        else
              c.fail();
      }
}
```

If, for example, the expression to be parsed is $5 + 3 - 2$*, which is represented by a logical list* `tokenList` *with value* `[5,+,3,-,2]`*, and* `sampleParser` *is an instance of the class* `ExprParser`*, then the invocation*

```
solver.check(sampleParser.expr(tokenList,LList.empty()))
```

will return `true`*, while, if* `tokenList` *has value* `[5,+,3,-]`*, the same invocation to* `sampleParser.expr` *will return* `false`*.* □

Actions to be performed when a non-terminal has been successfully reduced (e.g., to evaluate the parsed expression or to generate the corresponding target code) can be easily added to a DCG by adding new arguments to predicates defining non-terminals and new atoms at the end of the body of the corresponding clauses. Accordingly, the JSetL implementation of a DCG can be easily extended by adding new arguments and suitable statements to the user-defined constraints implementing the non-terminals.

For example, the DCG of Example 7.2 can be extended so as to return the JSetL constraint corresponding to the parsed arithmetic expression. Basically, this is obtained by adding an extra argument `z` of type `IntLVar` to each constraint of the DCG and a further argument `cc` of type `Constraint` to the constraint `expr`. Moreover, the constraint implementation is modified so as to construct in `cc` a constraint of the form `z.eq(`C_e`)`, where C_e is the JSetL constraint corresponding

to the input expression e. For instance, the switch alternative dealing with the `+` operator of the constraint `expr` is modified as follows:

```
case 1: {
  Solver.addChoicePoint(c);
  LList L1 = new LList();
  LList L2 = new LList();
  IntLVar z1 = new IntLVar();
  Solver.add(num(z1, L, L1));
  Solver.add(L1.eq(L2.ins1('+')));
  IntLVar z2 = new IntLVar();
  Solver.add(expr(z2, L2, Remain));
  Solver.add(z.eq(z1.sum(z2)));
  cc.add(z.eq(z1.sum(z2)));
  break;
}
```

If `sampleParser` and `tokenList` are those of Example 7.2, and `z` and `cc` are `IntLVar` and `Constraint` objects, respectively, then the invocation

```
solver.check(sampleParser.expr(tokenList,LList.empty(),z,cc))
```

will bind `cc` to the JSetL constraint:

```
z.eq(x_1.sum(x_2)).and(x_1.eq(5)).
and(x_2.eq(x_3.sub(2))).and(x_3.eq(3))
```

Finally, note that the simple expression grammar considered in the above examples forces right associativity in expression evaluation, which is rather unusual. Obtaining left associativity would require left-recursive grammar rules. However, the top-down execution strategy of Prolog, and the equivalent strategy used to solve nondeterministic constraints in JSetL, naturally yield a recursive descent parser, which is well known to be not adequate to handle left-recursive grammars: as a matter of fact it will go into an infinite loop on them. Therefore, when handling DCG in Prolog, as well as in JSetL, one should rearrange the grammar so as to avoid left-recursion. This in turn may result, in general, in slightly more complicated DCGs.

8 Conclusions and future work

In this paper we have shown, through a number of simple examples, that nondeterministic programming is feasible and it may be conveniently exploited also within conventional O-O languages such as Java. We have obtained this by combining a number of different features offered by the Java library JSetL: set data abstractions, nondeterministic constraint solving, logical variables, unification, user-defined constraints. In particular, general nondeterministic procedures can be defined in JSetL as new user-defined constraints, taking advantage of the facilities for expressing and handling nondeterminism provided by the solver.

As a future work we plan to identify the minimal extensions to be made to the JSetL's solver to make it capable of supporting, with the same approach outlined in this paper, other nondeterministic control structures such as those described for instance in [1] and [15].

Acknowledgments

This work has been partially supported by the G.N.C.S. project "Specifiche insiemistiche eseguibili e loro verifica formale". Special thanks to Luca Chiarabini who took part and stimulated the preliminary discussions on this work, and to Andrea Longo who contributed to the development of the JSetL-based implementation of DCGs.

References

[1] (1998) K.R. Apt, J. Brunekreef, V. Partington, A. Schaerf (1999) Alma-0: An Imperative Language that Supports Declarative Programming. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 20, No. 5, Sept. 1998, 1014–1066.

[2] K.R. Apt, A. Schaerf (1999) The Alma Project, or How First-Order Logic Can Help Us in Imperative Programming. In E.-R. Olderog, B. Steffen, Eds., *Correct System Design*, *LNCS*, v. 1710, 89–113. Springer-Verlag, 1999.

[3] F. Bergenti, L. Chiarabini, and G. Rossi (2011) Programming with Partially Specified Aggregates in Java. *Computer Languages, Systems & Structures*, Elsevier, 37(4), 178–192.

[4] Choco. `http://www.choco-constraints.net`.

[5] J. Cohen (1979) Non-Deterministic Algorithms. Computing Surveys, Vol. 11, No. 2, 79–94, June 1979.

[6] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi (2000) Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931.

[7] A. Dovier, E. Pontelli, and G. Rossi (2006) Set unification. *Theory and Practice of Logic Programming*, 6:645–701.

[8] Gecode. `http://www.gecode.org/`.

[9] JaCoP. `http://jacop.osolpro.com/`.

[10] (2010) JSR-331, Java Constraint Programming API (Early Draft). Java Community Process. `http://www.jcp.org`.

[11] S.H. Mirian-HosseinaAbadi, M.R. Mousavi (2002) Making Nondeterminism Explicit in Z. Proceedings of the Iranian Computer Society Annual Conference (CSICC 02), Tehran, Iran, February.

[12] G. Rossi, E. Panegai, and E. Poleo (2007) JSetL: A Java Library for Supporting Declarative Programming in Java. *Software-Practice & Experience*, 37:115-149.

[13] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg (1986) *Programming with sets, an introduction to SETL*. Springer-Verlag.

[14] H. Sondergaard, P. Sestoft (1992) Non-Determinism in Functional Languages. The Computer Journal, 35(5), October 1992, 514-523.

[15] P. Van Hentenryck, L. Michel (2006) Nondeterministic Control for Hybrid Search. Constraints, Vol. 11, No. 4, December 2006, 353– 373

[16] M. Walicki, S. Meldal (1993) Sets and Nondeterminism. Presented at ICLP'93 Post-Conference Workshop on Logic Programming with Sets (`http://people.math.unipr.it/gianfranco.rossi/sets/body_workshop.html`), Budapest, June 1993.

Stampato in proprio, a cura degli autori, presso il Dipartimento di Matematica e Informatica dell'Università degli Studi di Parma, Parco Area delle Scienze, 53/A, 43124 Parma, adempiuti gli obblighi ai sensi della Legge n. 160 del 15.04.2004 "Norme relative al deposito legale dei documenti di interesse culturale destinate all'uso pubblico" (G.U. n. 98 del 27 aprile 2004) e del Regolamento di attuazione emanato con D.P.R. n. 252 del 3 maggio 2006 (G.U. n. 191 del 18 agosto 2006) entrato in vigore il 2 settembre 2006 [precedente normativa abrogata: Legge n. 374 del 2.2.1939, modificata in D.L. n. 660 del 31 agosto 1945].

Esemplare fuori commercio per il deposito legale agli effetti della legge 15 aprile 2004, n. 160.