Programming with Partially Specified Collections

Gianfranco Rossi

Dipartimento di Matematica e Informatica
Università degli Studi di Parma, Parma, Italy
gianfranco.rossi@unipr.it

Abstract. Enumerated collections (e.g. lists, sets, etc.) provided by programming languages are usually defined by completely and precisely enumerating all their constituent elements. Conversely, in (constraint) logic programming languages it is common to deal with *partially specified collections* where either some elements or some parts of the collection are left unknown. In this paper we claim that partially specified collections (in particular, lists and sets) can be conveniently exploited in a wider setting, even in the context of more conventional languages using a library-based approach. We prove this claim by showing a number of simple examples using Java and the Java library JSetL.

1 Introduction

Programming languages usually provide several kinds of collections, e.g. array, string, lists, sets, multi-sets, maps, etc, either as *primitive* abstractions or programmed by exploiting the data abstraction facilities of the language itself and made available to users as standard libraries. For example, sets are a primitive data abstraction in Pascal, SETL and Python, while they are provided as libraries in C++ and Java.

In this paper we will restrict our attention to *enumerated collections* only, that is collections designated by explicitly enumerating its constituting elements. Hereafter the term *collection* implicitly refers to *enumerated* collections.

In imperative programming languages collections need to be always *completely specified* whenever we operate on them. This means that all elements of a collection must be provided and they must have known values. For example, using the interface `Set` of `java.util`, the set $\{1, 3, 0\}$ can be constructed by executing the statements

```
s.add(0); s.add(3); s.add(1);
```

where `s` is an object denoting the empty set which can be defined as

```
Set s = new HashSet();
```

and `HashSet` is one of the implementations of the interface `Set` provided by `java.util`. Note that the requirement to be completely specified does not prevent a collection from containing variables, but whenever the collection needs to be evaluated all variables possibly occurring in it must be replaced by their actual values.

In contrast, collections in declarative programming languages, such as lists in Prolog, are allowed to contain unknown elements and to be only partially specified, and, nevertheless, they can be accessed and manipulated. For instance, the Prolog predicate `[a,b,c] = [X,Y|R]` compares the (completely specified) list of three elements, `a`, `b`, and `c`, with the partially specified list containing two unknown elements `X` and `Y` and an unknown remainder part `R` (i.e., any list containing at least two elements).

Usefulness of partially specified collections has been amply demonstrated in the context of declarative programming languages. In particular, programming with partially specified lists is a well-assessed technique in standard logic programming languages. Partially specified sets, instead, are supplied by extended logic programming languages with sets, such as [6, 7, 5], and are shown to be a powerful data abstraction. Usefulness of partially specified sets is advocated also in specific application areas such as deductive databases, computational linguistics, and knowledge representation.

Aim of this paper is to show that partially specified collections can be conveniently exploited also in a more conventional setting, namely an imperative O-O programming language. In particular, we consider a *library-based approach* where the new data abstractions are defined by using the language data abstraction mechanisms, without requiring any extension of the host language.

We prove our claim by presenting a number of simple examples written in Java using JSetL [19], a general-purpose library that supports partially specified lists and sets, within the O-O language Java. JSetL integrates the notions of logical (or constrained) variable, (set) unification and constraints that are typical of constraint logic programming languages into the Java language. In particular, it allows logical variables to occur in list and set data structures and provides operations to deal with such objects.

The presented examples aim at showing, in particular, how partially specified lists and sets can be exploited for:

- supporting a *declarative programming style*
- providing data modelling tools (in particular sets and set unification) which prove to be particularly suitable for problems involving non-determinism
- supporting constraint programming.

It is important to note that our main concern here is on providing modelling and programming support. Conversely we are only relatively little interested in the efficiency issue (e.g., efficiency of the constraint solver). Furthermore, our focus is on *programming* rather than on constraint solving or satisfiability checking of formulas involving partially specified collections.

Though our discussion is limited to lists and sets, similar considerations could apply to other kinds of collections as well. Moreover, though we are focusing on Java, the same considerations could be easily exported to other O-O languages (such as C++) where these data abstractions could be supplied by libraries, while they can serve as useful guidelines to devise extensions to programming languages where these new data abstractions could be provided as primitive ones.

The paper is organized as follows. Section 2 introduces the notion of partially specified collection, showing how logical variables can be used within collections to represent unknown elements or part of the collection. In Section 3 we start considering operations on partially specified collections, specifically (set) unification. In Section 4 we show how simple inequality, membership and integer comparison constraints over elements of partially specified collections can be used to describe and compute solutions for a number of common programming problems. Section 5 takes into account also set variables, whose domains are collections of sets, and it briefly shows how set variables and set constraints can be advantageously exploited in conjunction with partially specified sets. Sections 6 and 7 are devoted to further remarks and to draw some conclusion.[1]

2 Partially Specified Collections

Declarative programming languages (e.g., functional and logic programming languages) are based on a notion of variable, often called *logical* (or *mathematical variables*, *constrained*) *variable*, that differs from that of imperative programming languages. Logical variables represent unknowns, not memory cells. As such they have no modifiable value stored in them, as ordinary programming language variables have. Conversely, one can associate values to logical variables through relations (or *constraints*) involving logical variables and values from some specific domains.

When the domain of a (logical) variable is restricted to a single value then the variable is said to be *bound* (or *instantiated*) to this value. Otherwise, the variable is *unbound*. An equality relation, in particular, may restrict the domain of a logical variable to a single value. For example, if x is a logical variable ranging over the domain of integers, the equality $x = 3$ forces x to be bound to the value 3. However, the same result could be obtained through other relations, e.g., $x < 4 \wedge x > 2$. The value of a logical variable is immutable. That is it can not be changed, e.g. by an assignment statement like in imperative languages.

Logical variables are found also in conventional programming languages that support *constraint programming* (e.g., Alma-0 [1]), as well as in libraries for constraint programming, such as ILOG [13] and Choco [3] (see also the recent Java Specification Request for a standard Java Constraint Programming API [14]).

In this paper we will refer to what is provided by JSetL [19], a Java library that supports declarative (constraint) programming in an O-O framework. In JSetL a (generic) logical variable is an instance of the class `LVar`. Basically, `LVar` objects can be manipulated through constraints, namely equality (`eq`), inequality (`neq`), membership (`in`) and not membership (`nin`) constraints. Moreover the library provides methods to test whether a variable is bound or not, to get the value of a bound variable (but not to modify it), to get its external name, and so on. Values associated with generic logical variables can be of any type. For

[1] An extended version of this paper appeared in Computer Languages, Systems & Structures, Elsevier, vol. 37/4, 178-192, 2011

some specific domains, however, JSetL offers specializations of the `LVar` data type, which provide further specific constraints. In particular, for the domain of integers, JSetL offers the class `IntLVar`, which extends `LVar` with a number of new methods and constraints specific for integers. In particular, `IntLVar` provides integer comparison constraints such as $<$, $\leq$, etc.

Other important specializations of logical variables are the class `LCollection` and its derived subclasses, `LSet` (for Logical Sets) and `LList` (for Logical Lists). Values associated with `LSet` (`LList`) are objects of the `java.util` class `Set` (`List`). A number of constraints are provided to work with `LSet` (`LList`), which extend those provided by `LVar`. In particular, `LSet` provides equality and inequality constraints that account for the semantic properties of sets (namely, irrelevance of order and duplication of elements); moreover it provides constraints for many of the standard set-theoretical operations, such as union, intersection, set difference, and so on.

Example 1. Logical lists/sets in JSetL.

```
LVar x = new LVar();          // an unbound logical variable
LVar y = new LVar("y",1);     // a bound logical variable
                              // with external name "y" and value 1
LList ll = new LList();       // an unbound logical list
LSet ls_1 = new LSet();       // an unbound logical set
```

A collection where either some elements, or part of the collection itself, are unbound logical variables represents a *partially specified collection*. For example (using a Prolog-like notion), the list `[1,X,Y]`, where `X` and `Y` are unbound logical variables, denotes a partially specified list containing one known element `1` and two unknown elements, denoted `X` and `Y`. As another example, the list `[1,2|Z]`, where `Z` is an unbound logical list (i.e., a logical variable ranging over the domain of lists), designates an *open list* containing two known elements 1 and 2 and an unknown remainder part denoted by `Z`.

A partially specified set (list) can be created in JSetL by adding possibly unknown elements, i.e., unbound `LVar` objects, to either a known or an unknown set (list) using the element insertion operator `ins`.

Example 2. The two partially specified lists `[1,X,Y]` and `[1,2|Z]` can be defined in JSetL as follows:

```
LVar X = new LVar();
LVar Y = new LVar();
LList cl =                                          // the closed p.s. list
    LList.empty().ins(Y).ins(X).ins(1);             // [1,X,Y]
```

and

```
LList Z = new LList();                              // the open p.s. list
LList ol = Z.ins(2).ins(1);                         // [1,2 |Z]
```

JSetL provides a number of facilities, mainly in the form of constraints, to access and manipulate partially specified lists/sets (e.g., unification). Conversely,

most of the non-CLP languages/libraries mentioned above allow (possibly unbound) logical variables to occur in conventional data structures, such as vectors, but they do not provide any specific operation to deal with them as partially specified collections. Even languages/libraries that provide set constraints [11], such as ILOG-Solver [13], Choco [3], and Mozart-Oz [15], do consider only completely specified sets.

3 (Set) Unification

One of the basic operation to access and manipulate partially specified collections in logic programming languages is unification. JSetL provides both standard unification, over `LList` objects, and set unification, over `LSet` objects. Both forms of unification are implemented by the equality method `eq`. The meaning of `o1.eq(o2)` is the unification between the objects `o1` and `o2`. `o1` is either a simple logical variable (i.e., an instance of `LVar`) or a possibly partially specified data collection (i.e., an instance of `LList` or `LSet`). `o2` is either an object of the same type as `o1` or an admissible value for it (i.e., a `Set` object, an `Integer` object, and so on). The following is an example of unification over lists in JSetL.

Example 3. Check whether list `l3` is the concatenation of lists `l1` and `l2`. The problem can be modelled as a (list) unification problem. If the first list `l1` is the empty list, the other two lists must be unifiable (and vice versa). Otherwise, the given lists must satisfy the constraint (using a Prolog-like abstract notation) $\texttt{l1} = [\texttt{x} \mid \texttt{l1new}] \ \wedge \ \texttt{l3} = [\texttt{x} \mid \texttt{l3new}]$, where `l3new` is the concatenation of the "shorter" list `l1new` and the given list `l2`, and equality = represents unification.

```
public static boolean concat(LList l1, LList l2, LList l3) {
    if ( solver.check(l1.eq(Lst.empty()).and(l2.eq(l3))) )
        return true;
    else {
        LVar x = new LVar();
        LList l1new = new LList();
        LList l3new = new LList();
        return
            solver.check(l1.eq(l1new.ins(x)).and(l3.eq(l3new.ins(x))))
            && concat(l1new,l2,l3new);
    }
}
```

The first call to the method `check` tests satisfiability of the constraint $\texttt{l1} = [] \ \wedge \ \texttt{l2} = \texttt{l3}$. `solver` is an object of the class `SolverClass` which is assumed to be created outside the method `concat`. Solving an equality calls into play unification: if the two lists are unifiable, then `check` returns `true`, otherwise it returns `false`. If the constraint $\texttt{l1} = [] \ \wedge \ \texttt{l2} = \texttt{l3}$ is `false`, then the constraint $\texttt{l1} = [\texttt{x} \mid \texttt{l1new}] \ \wedge \ \texttt{l3} = [\texttt{x} \mid \texttt{l3new}]$ is checked and the result is combined with the result of the recursive call to `concat`.

This example shows how the use of a partially specified collection and unification allows us to devise a truly declarative solution to the given problem. Another advantage of using unification—in place of assignment and standard programming variables—is that the same methods can be used both to assign values to variables and to test known values. In particular, unification over partially specified lists can be used both to access single elements of a list and to construct the list itself. For example, the method `concat` can be used either to test if a list `l3` is the concatenation of two given lists `l1` and `l2` or to get one list given the other two. No assumption is made about which are input and which are output parameters. As an example, if `l3` is the JSetL `LList` for `['d','a','t','a','.','t','x','t']`, `l2` is the JSetL `LList` for `['.','t','x','t']`, and `l1` is an unbound `LList`, then the invocation `concat(l1,l2,l3)` binds `l1` to the LList `['d','a','t','a']`.

Set unification differs from standard unification in that the former must account for the properties of sets, namely that order and repetition in a set do not matter. Thus, for example, the two set unification problems, $\{a\} = \{a, a\}$ and $\{a, b\} = \{b, a\}$ have a solution, whereas they would have no solution using standard unification. A general survey of the problem of unification in presence of sets, across different set representations and different admissible classes of set terms, can be found in [8]. The following is a simple example using partially specified sets and set unification.

Example 4. (Coloring of a map) Given a map of n regions and a set of m colors find an assignment of colors to regions such that neighboring regions have different colors. To solve this problem we assume to represent the map as a set whose elements are themselves sets containing two neighboring regions and to represent each region as a distinct unbound logical variable. Hence, the map is represented by a partially specified set. For example,

$$\{\{r_1, r_2\}, \{r_2, r_3\}, \{r_3, r_4\}, \{r_4, r_1\}\}$$

where r_1, r_2, r_3, r_4 are unbound logical variables, is a map of four regions. With these assumptions, the coloring problem can be modelled as the problem of unifying the set representing the map with the set of all admissible unordered pairs of colors that can be constructed from the given m colors. For example, given the map considered above and the set of colors $\{1, 2, 3\}$, the problem to be solved turns out to be (using the usual Prolog-like abstract notation):

$$\{\{r_1, r_2\}, \{r_2, r_3\}, \{r_3, r_4\}, \{r_4, r_1\} \mid R\} = \{\{1, 2\}, \{2, 3\}, \{1, 3\}\}$$

where R represents the set of viable color pairs that possibly have not been used in the computed solution. One of the many possible solutions to this problem is:

$$r_1 = 1, r_2 = 2, r_3 = 1, r_4 = 3, R = \{\{2, 3\}\}.$$

Using JSetL, the proposed modelling of the coloring problem can be easily implemented in Java by the following method:

```
public static void coloring(LSet map, LSet colorPairs) {
  LSet R = new LSet();
  solver.check(colorPairs.eq(R.insAll(map)));
}
```

where `R.insAll(map))` is the set containing all elements of the set `map` and an unknown part `R`, and `colorPairs` is the set of all admissible unordered pairs of colors. This set can be constructed "by hand" or it can be computed starting from the set of colors by exploiting set unification in a way similar to the one shown in Example 5.

The coloring example shows that the use of partially specified sets and set unification allows us to devise very concise and declarative—though possibly inefficient—solutions.[2]

A key feature of unification over partially specified sets is the *non-determinism* embedded in the set unification algorithm. As a matter of fact, a set unification problem may have more than one (independent) solution (non-uniqueness of mgu's). For example, $\{X, Y\} = \{a, b\}$, where X and Y are unbound variables, has *two* solutions, namely (i) $X = a, Y = b$ and (ii) $X = b, Y = a$. The set unification algorithm can compute all these solutions, one after the other, through backtracking. The following is an example that shows how to exploit this feature within a Java program.

Example 5. (Permutations) Print all permutations of the integer numbers from 1 to n ($n \geq 0$). The problem can be modelled as the problem of unifying a (partially specified) set of n logical variables $\{X_1, \ldots, X_n\}$ with the set of the integer numbers from 1 to n, i.e., $\{X_1, \ldots, X_n\} = \{1, \ldots, n\}$. 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 between 1 and n. The following method `allPermutations` implements this solution in Java using JSetL.

```
public static void allPermutations(int n) {
    IntLSet I = new IntLSet(1,n);      // I = {1,2,...,n}
    LSet S = LSet.mkLSet(n);           // S = {X1,X2,...,Xn}
    solver.add(S.eq(I));               // {X1,X2,...,Xn} = {1,2,...,n}
    solver.check();
    do {
        S.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 `I`. The method `add` allows a constraint to be added to the *constraint store* of the specified solver. Constraints stored in the constraint store can be checked for satisfiability by calling the constraint solving procedure of the solver, e.g. by invoking the method `check()`. Calling the method `nextSolution()` allows to check whether the constraint in the constraint store of the current solver admits further solutions and possibly to compute the next one. Finally, the invocation

[2] An alternative, more efficient, solution of the coloring problem using constraints will be hinted at in the next section.

`s.printElems(' ')` prints all elements of the set `s` on the standard output separated by the specified character (blank in this case).

This example also illustrates the interplay between Java statements and the non-determinism mechanism embedded in the JSetL constraint solving procedure. In particular, using the method `nextSolution()` within an iterative statement such as the `do_while` construct provides a simple way to compute all possible solutions for a given problem. As a matter of fact, `nextSolution()` exploits the backtracking mechanism embedded in the constraint solver: calling `nextSolution()` forces the computation to go back until the nearest open choice point. In this example the choice points are created during execution of the set unification algorithm called into play by solving the equality constraint between the two sets `I` and `S`.

Remark 1. The set unification problem has been proved to be a *NP-complete problem*. This may lead, in general, to an exponential growth in the complexity of the satisfaction procedure used in JSetL. However, the full power of set unification is not always required. As an example, in the problem of Example 5 we can observe that one of the two sets is completely known (the set `I`), while the other (the set `S`) is a closed set of exactly n unbound variables. Thus we can replace the equality constraint $\mathtt{S} = \mathtt{I}$ that implements general set unification with the conjunction of constraints $\mathtt{1} \in \mathtt{S} \wedge \mathtt{2} \in \mathtt{S} \wedge \ldots \wedge n \in \mathtt{S}$.[3] The new version of the method `allPermutations` using n membership constraints turns out to be executed much more efficiently than the previous one.

4 Constrained Partially Specified Collections

Logical variables occurring in partially specified collections can be suitably constrained. This allows us, in general, to characterize more precisely the collection of data structures denoted by a partially specified collection. For example, the partially specified list $[X, Y]$, where X and Y are unbound logical variables, with the additional constraint $X > Y$, can be used to designate any list of two elements with the first element greater than the second. The following is a simple example showing how inequality and membership constraints can be advantageously exploited in conjunction with partially specified lists.

Example 6. (All pairs) Compute the set of all pairs `[x,y]` such that both `x` and `y` belong to a given set `s` and $\mathtt{x} \neq \mathtt{y}$.

```
public static Set<?> allPairs(Set<?> s) {
    LVar x = new LVar();
    LVar y = new LVar();
    LList pair = new LList.empty().ins(y).ins(x);  // a pair [x,y]
    solver.add(x.in(s).and(y.in(s)).and(x.neq(y)));
    return solver.setOf(pair);
}
```

[3] Note that `S` has fixed cardinality, which prevents it from containing other values other than $1..n$.

The first four statements serve to specify the form of the pairs we are looking for: partially specified lists of the form `[x,y]`, with $\texttt{x} \in \texttt{s} \wedge \texttt{y} \in \texttt{s} \wedge \texttt{x} \neq \texttt{y}$. The last statement calls the method `setOf` of the current solver which collects in a Java set all the pairs satisfying this specification and returns it as its result.

This example shows an interesting interaction between partially specified collections (namely, JSetL logical lists), containing logical variables, and completely specified collections (namely, sets and lists of `java.util`). The latter are used to contain known values, while the former are used to describe the general form of values to be computed.

The method `setOf` implements a (limited) form of *intensionally defined sets*. A set s is defined as the set of 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 is represented as a JSetL constraint, the set collection can be easily performed by using `setOf()` over this constraint. As a matter of fact, the set s computed in the above example is the set $\{[\texttt{x},\texttt{y}] : \texttt{x} \in \texttt{s} \wedge \texttt{y} \in \texttt{s} \wedge \texttt{x} \neq \texttt{y}\}$.

The same effect can be obtained more explicitly by combining iteration and `nextSolution()`, similarly to Example 5. Specifically, the call to the method `setOf` in Example 6 can be replaced by the following code fragment:

```
Set pairSet = new HashSet();
solver.check();
do {
    pairSet.add(pair.getValue());
} while (solver.nextSolution());
```

In this solution each computed pair is explicitly added to the set of all solutions s by using the (destructive) `add` method of Java sets. Collecting all solutions in a data structure such as s—instead of simply printing them as in Example 5—allows the program to subsequently work on them using operations provided by that data structure.

In the next example we make use of a list of constrained integer logical variables to represent the solution we are looking for. Assigning possible values to these variables will allow us to obtain all the admissible solutions for the given problem (actually, only one solution in the considered example).

Example 7. (Sort) Sort a collection of n distinct integer numbers in ascending order. The problem can be modelled as a finite domain constraint satisfaction problem in the following way. Let s be the `Set` object representing the collection to be ordered, and $lOrd$ be a list of n integer logical variables X_i, $i = 1 \ldots n$. Variables in $lOrd$ are constrained so that each X_i has domain s and the following conjunction holds: $\bigwedge_{i=1}^{n-1} X_i < X_{i+1}$. Then a solution to this problem is computed by finding the instance of $lOrd$ (i.e., an assignment of values to all its variables) that satisfies the given constraint. This modelling of the sorting problem is easily implemented in Java using JSetL by the following method:

```
public static List sortList(Set s) {
    int n = s.size();
```

```
    IntLList lOrd = IntLList.mkLList(n);      // lOrd = [X1,X2,...,Xn]
    Iterator it = lOrd.iterator();            // it = Iterator over LList
    while(it.hasNext())                       // ∀i ∈ 1..n, Xi.in(s)
        solver.add((((IntLvar)it.next()).in(s)));
    for(int i=0; i<n-1; i++)                  // ∀i ∈ 1..n, Xi.lt(Xi+1)
        solver.add(((IntLvar)lOrd.get(i)).lt((IntLvar)lOrd.get(i+1)));
    solver.check();
    return lOrd.getValue();
}
```

The invocation `IntLList.mkLList(n)` creates a partially specified list composed of `n` integer logical variables (`IntLList` is a subclass of `LList` where list elements are restricted to be integers—either constants or logical variables of type `IntLVar`). The `while` statement allows membership constraints to be added to the *constraint store* of the constraint solver `solver` for each variable of the list `ordList`. The next `for` statement adds the `lt` constraints that force values for the variables in `ordList` to be assigned respecting the desired ordering relation (namely, $<$). These constraints, along with the membership constraints posted before, are checked for satisfiability through the method `check()`. Finally, the Java list representing the value of the logical list `ordList` is obtained by invoking the method `getValue` and returned as the final result of the method `sortList`. As an example, if `s` is the set $\{5, 2, 4\}$, the constraint to be solved is

$$X_1 \in \{5,2,4\} \wedge X_2 \in \{5,2,4\} \wedge X_3 \in \{5,2,4\} \wedge X_1 < X_2 \wedge X_2 < X_3.$$

Efficiency of the proposed solutions strongly depends in general on how efficiently the constraint solver can handle the involved constraints. For instance, if the constraints generated in Example 7 are solved by using a simple generate & test approach, computational complexity of the proposed solution is in $\mathcal{O}(n!)$ and it is clearly unacceptable even for extremely small n. Conversely, if the solver can exploit the efficient techniques used to solve finite domain (FD) constraints (see, e.g., [10] for an overview), the computational behavior of the proposed solution is in $\mathcal{O}(2^n)$ (the assignments of values to variables that the solver has to test constitute paths of a *binomial tree*), and this is more acceptable, at least for small values of n. This behavior is obtained in JSetL by simply replacing the membership constraint `in` by a constraint `dom` in the first `while` statement[4], and adding the posting of the `label` constraint over the variables in `lOrd`, i.e., `solver.add(lOrd.label())`, before calling the `check` method. With the new solution, values will be assigned one by one to the unbound variables by the `label` constraint and propagation will shrink the domains of the involved variables consequently, until all the domains get reduced to singletons.

Sometimes it may be convenient to reformulate the problem solution in order to exploit the more efficient constraint solving techniques made available by the constraint solver. For example, the simple but very inefficient solution

[4] The semantics of `x.dom(s)` is the same as that of the membership constraint `x.in(s)`, but solving the former simply updates the domain of `x` to `s`, whereas the latter (non-deterministically) assigns all values in `s` to `x`.

to the coloring problem of Example 4 can be replaced by a slightly more complex but much more efficient solution that exploits inequality constraints and FD constraint solving. The only differences w.r.t. the previous version are that colors are represented by integers and the set of all regions (actually, a partially specified set containing one logical variable for each region in the map) is passed explicitly to the method `coloring`. The idea is to state that the domain of each variable representing a region is the set of colors, and then to post the constraints that specify that the two regions in each pair of the map must be distinct.

However, it is important to note that, generally speaking, execution efficiency is not a primary requirement when using the kind of partially specified collections we are considering in this paper. Actually, easiness of problem modelling, as well as easiness of program development and understanding, are definitively more important features in this context.

5 Set Constraints

In Section 4 we have considered constraints over the *elements* of a (partially specified) collection. However, logical variables can range over the domain of collections as well (e.g., sets), and constraints can apply to *whole collections*. Various proposals exist in the literature that take into account this kind of constraints (see [12] for a survey). In particular, *set constraints* [11] are based on set variables whose domain is a finite collection of sets of integers, usually specified as a *set interval* $[l, u]$ representing the lattice of sets induced by the subset partial ordering relation $\subseteq$ having sets l and u as the greatest lower bound and the least upper bound, respectively.

Implementations of set constraints solvers are provided, e.g., by ECLiPSe [4], B-Prolog [20], Mozart-Oz [15], and Choco [3]. All these proposals consider only completely specified sets, whose elements are usually restricted to integer values.

However, as shown in previous sections, it may be useful in many occasions to allow also partially specified collections to be taken into account. As a matter of fact, this could significatively enhance the problem modelling capabilities. For example, if we want to say that s is a set containing at least the element 1 but not containing 0 we could simply state this as

$S = \{1 \mid R\} \wedge 0 \notin S$

that is, by using an open partially specified set and a non-membership (set) constraint over it.

Though partially specified, it may nevertheless be convenient to perform various operations on the (whole) collections. The case of equality over partially specified lists and sets has already been discussed in Section 3, where it is reconduct to list/set unification. Other operations, such as membership, inclusion, union, intersection, cardinality, and so on, could be conveniently applied to partially specified collections as well. For example, in the above example, we could want to say also that the intersection between the set S and another set T must be empty, i.e., $S \cap T = \emptyset$.

The ability to process such operations on partially specified collections require to deal with them as constraints and to have a constraint solver that is able to properly account for possibly unbound variable occurring within the collections. Such a solver could be able, in particular, to detect unsatisfiability of the given constraint even if the involved collections are only partially known, possibly leading in this way to significatively cut the search space. With reference to the above example, if we further require that the set T contains the set $\{1\}$, that is:

$S = \{1 \mid R\} \wedge 0 \notin S \wedge S \cap T = \emptyset \wedge T \subseteq \{1\}$

the solver could be able to immediately detect that the constraint is unsatisfiable.

A complete solver for these kind of constraints, specifically constraints over partially specified sets, is provided by {log} [6, 7, 5]. A Java implementation of this solver is also made available in the context of the Java language by the JSetL library. Obviously solving this kind of constraints turns out to be in general much more inefficient than solving set constraints over only completely specified sets. Moreover, as it happens with set unification, the solution of many of these constraints may involve non-determinism.

The following are a few examples of the resolution of simple constraints over either open or closed partially specified sets in JSetL.

Example 8. Partially specified set constraints.

```
LVar x = new LVar();
IntLVar n = new IntLVar();
LSet r = new LSet();
LSet s1 = LSet.empty().ins(1);                  // s1 = {1}
LSet s2 = LSet.empty().ins(2).ins(x);           // s2 = {x,2}
LSet s3 = r.ins(1).ins(x);                      // s3 = {x,1 | r}
```

- Solving the constraint

  ```
  solver.solve(r.inters(s1,s2));                // r = s1 ∩ s2
  ```

  we get the two distinct answers: $x \neq 1 \wedge r = \{\}$ and $x = 1 \wedge r = \{1\}$.
- Solving the constraint

  ```
  solver.solve(s2.size(n));                     // n = |s2|
  ```

  we get the two distinct answers: $x \neq 2 \wedge n = 2$ and $x = 2 \wedge n = 1$.
- Solving the constraint

  ```
  solver.solve(s3.ncontains(0));                // 0 ∉ s3
  ```

  we get: $x \neq 0 \wedge 0 \notin r$; if we would consider in addition the constraint $1 \notin s3$ then we would get an immediate failure.

The ability to deal with collections whose elements can be of any type (not only integers) allows one to account also for collections containing other collections as their elements, i.e., *nested collections*. Moreover, the inner collections can be partially specified collections or even unbound logical lists/sets. For example, the object `ss`, which is created by the following two Java statements using JSetL:

```
LSet r = new LSet();
```

```
LSet ss = new LSet().ins(r);
```
represents any set containing at least another (nested) set `r`. As a more concrete example, if we want to model a system composed of at least k pieces, each of which has a name $\mathtt{n}_i$, a quantity $\mathtt{q}_i$ and a unit cost $\mathtt{c}_i$, and if we assume that the name and cost of each piece are known, whereas its quantity is unknown, we can represent such a system by the following partially specified set of lists (actually triples):

$\{[\mathtt{"n}_1\mathtt{"},\mathtt{c}_1,\mathtt{q}_1],\ldots,[\mathtt{"n}_k\mathtt{"},\mathtt{c}_k,\mathtt{q}_k] \mid \mathtt{r}\}$

where $\mathtt{q}_1$, ..., $\mathtt{q}_k$ are unbound integer logical variables and `r` is an unbound logical set variable. Then we can post constraints either on the integer logical variables $\mathtt{q}_i$ (e.g., $\mathtt{q}_i$`.in(1,10)`), or on the set variable `r` or on the set representing the whole system (e.g., `r.contains(["o",c,q])`, `c`, `q` unbound logical variables).

6 Discussion and related work

Logic variables are a key feature of languages/libraries to support *constraint programming*. However, in the context of conventional programming languages, logical variables are rarely exploited to support further possibilities, such as:

- in conjunction with unification, to allow the same code to be used for different purposes, e.g., to test a given condition or to compute those values for which the condition holds (*procedure invertibility*);
- to allow *partially specified collections* to be represented and handled, possibly by using suitable constraints.

As concerns procedure invertibility, one exception is the language Alma-0 [1] which explicitly considers it by providing a generalized test expression (which becomes an assignment in the case one side is un unbound variable and the other side is an expression with a known value) and suitable parameter passing mechanisms.

As concerns partially specified collections, languages/libraries that support constraint programming in the context of conventional programming usually allow logic variables to occur in compound data structures. For instance, Alma-0 allows the user to define arrays of (possibly unbound) logical variables, and it allows the programmer to deal with them by a few global constraints such as the `ALL_DIFFERENT` constraint. However, no one of these proposals does really consider these structures as partially specified data and provide operations to deal with them accordingly.

Of course, partially specified collections can be defined in the formal languages provided by generic proof assistants such as Coq [2] and Isabelle [16] and manipulated as such by using their primitive facilities. However, these tools are hardly exploitable from within conventional O-O languages such as Java, in the manner we have shown in this paper.

As concerns efficiency, we have no precise empirical assessment with respect to Prolog systems where unification and backtracking are primitive mechanisms

(actually, as already mentioned, performance is not our main concern). However, unification, constraint solving and (embedded) backtracking can, at least in principle, be implemented within the library as efficiently as they are implemented in Prolog systems. With this respect, there is no price to pay for integrating concepts from logical programming into Java. Conversely, user-defined non-deterministic procedures, that can be defined in JSetL by exploiting the user-defined constraint facility [18], are likely to be much more inefficient in JSetL than in Prolog.

7 Conclusions

In this paper we have presented the potential offered by the presence of partially specified collections, in particular, lists and sets, in a conventional O-O programming language. Usefulness of partially specified collections in declarative programming languages is widely accepted. We have shown that introducing this kind of data abstractions in a more conventional programming setting is feasible and offers several potential advantages. To support our claim we have presented a number of simple examples written in Java using JSetL, a general-purpose library that provides general forms of data collections.

Although we have considered only lists and sets, there are a number of other kinds of data collections, akin to lists and sets, which are often provided by programming languages and libraries and which could be allowed to be partially specified. The solutions and techniques described in this paper could be adapted to these other data collections as well. In particular, a formal characterization of (possibly partially specified) multisets, i.e., unordered collections where element repetitions are allowed, and the definition of suitable solving procedures for equality and inequality constraints over them can be found in [9].

A weak point of our library-based approach is that the notation, necessary to respect the syntax of Java, is much heavier than what one would like to see. A preprocessor is under development at present to remedy, at least partially, this problem. No real extension to the Java syntax is required but a smart use of comments. Using the preprocessor, it will be possible for instance to create a set by using a sort of set literal, such as `LSet s1 = new LSet(/*{1,2,3}*/)`.

As a future work, we plan to compare the proposed approach with *lazy data structures* as those provided, e.g., by the Scala language [17], that seem to have interesting similarities with our partially specified structures. However, a strict requirement of our work is to use a library-based approach. This approach has the undeniable advantage of being easier to develop and more likely to be accepted by the (rather conservative) programmers. On the other hand, integration between the considered data abstractions and the rest of the language is weaker in our proposal than in an extended language, such as Alma-0 or Scala, where the desired abstractions are added as first-class citizen of the language.

References

1. Apt, K.R., Schaerf, A. Programming in Alma-0, or Imperative and Declarative Programming Reconciled. In D. M. Gabbay and M. de Rijke, Eds., *Frontiers of Combining Systems 2*, 1–16, Research Studies Press Ltd (2000)
2. Bertot, Y., Castéran, P. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. Springer, 469 pages (2004)
3. Choco. `http://www.emn.fr/z-info/choco-solver/`.
4. ECLiPSe *The ECLiPSe Constraint Logic Programming System*. London. `http://eclipseclp.org/`.
5. Dal Palù, A., Dovier, A., Pontelli, E., and Rossi, G. Integrating Finite Domain Constraints and CLP with Sets. In Miller, D., Ed., *Fifth ACM-SIGPLAN Int'l Conf. on Principles and Practice of Declarative Programming*, ACM Press, 219–229 (2003)
6. Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G. {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44 (1996)
7. Dovier, A., Piazza, C., Pontelli, E., and Rossi, G. Sets and Constraint Logic Programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931 (2000)
8. Dovier, A., Pontelli, E., and Rossi, G. Set unification. *Theory and Practice of Logic Programming*, 6:645–701 (2006)
9. Dovier, A., Piazza, C., and Rossi G. A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM Transactions on Computational Logic*, 9(3):1–30 (2008)
10. Henz, M., Müller, T. An Overview of Finite Domain Constraint Programming *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies*, Singapore (2000)
11. Gervet, C. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244, (1997)
12. Gervet, C. Constraints over Structured Domains. In Rossi, F., van Beek, P., and Walsh, T., Eds., *Handbook of Constraint Programming*, Elsevier (2006)
13. ILOG *ILOG Solver 6.0 Reference Manual* (2003)
14. JSR-331, Java Constraint Programming API (Early Draft). Java Community Process. `http://www.jcp.org` (2010)
15. The Mozart Consortium. The Mozart programming system. `http://www.mozart-oz.org` (2006)
16. Nipkow, T., Paulson, L., and Wenzel, M. (2002) *Isabelle/HOL. A Proof Assistant for Higher-Order Logic.* LNCS 2283, Springer Verlag, 218 pages.
17. Odersky, M., Spoon, L., and Venners, B. *Programming in Scala: a comprehensive step-by-step guide.* Artima Press (2008)
18. Rossi, G., Bergenti, F. Nondeterministic Programming in Java with JSetL. In Cantone, D., Nicolosi Asmundo, M., eds, CILC 2013: Italian Conference on Computational Logic, CEUR Workshop Proceedings, Vol. 1068, 211-226 (2013)
19. Rossi, G., Panegai, E., and Poleo, E. JSetL: A Java Library for Supporting Declarative Programming in Java. *Software-Practice & Experience*, 37:115-149 (2007)
20. Zhou, N-F. B-Prolog `http://www.probp.com/`.