# Programming with Partially Specified Aggregates in Java

Federico Bergenti, Luca Chiarabini, Gianfranco Rossi

Dipartimento di Matematica
Università degli Studi di Parma
Parco Area delle Scienze 53/A, 43100 Parma, Italy
federico.bergenti|luca.chiarabini|gianfranco.rossi@unipr.it

**Abstract**

Various forms of data aggregates, e.g., arrays, lists, sets, etc., are usually provided by programming languages, either as primitive entities or as additional features made available by standard libraries. In conventional programming languages these data structures are usually specified by completely and precisely enumerating all their constituent elements. Conversely, in (constraint) logic programming languages it is common to deal with partially specified aggregates where either some elements or some parts of the aggregate are left unknown. In this paper we consider the case where partially specified aggregates can occur in a conventional O-O programming language. Specifically, we consider partially specified lists and sets as provided by the Java library JSetL. The definition of such data structures is strongly based on the notion of logical (or constrained) variable usually provided by languages and libraries to support constraint programming. We show through simple examples using Java and JSetL how partially specified lists and sets, along with a few basic constraints over them, can be conveniently exploited in a number of common programming problems.

## 1 Introduction and Motivation

A *data aggregate* (or, simply, an aggregate) is a collection of several, either atomic or structured, data values. An aggregate can be dealt with as a whole, e.g., by passing it as a parameter to a function, or it can be accessed to select one of its elements, e.g., by an index or a symbolic name. The way elements are organized in the data structure and the way they can be accessed singularly precisely characterize the type of the aggregate.

Programming languages usually provide several kinds of data aggregates. Besides the ubiquitous array, string and record (`struct` in C), quite often programming languages provide other well-known aggregates, such as lists, sets, multi-sets, maps, etc. These structured data types are provided either as *primitive* abstractions or they are programmed by using the data abstraction mechanisms of the language itself and made available to the user 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. Discussing the pros and cons of an approach or the other is outside the scope of this paper. Basically we do not distinguish between primitive and user-defined data types if not necessary.

A data aggregate is designated primarily by explicitly enumerating all its elements. Very few programming languages allow aggregates to be designated also *by property*. Two notable exceptions are SETL and Python that allow an aggregate, in particular, a set, to be defined by stating a property $\varphi$ that must be satisfied by all its elements, e.g. $\{x \mid \varphi\}$. Data aggregates defined by property represent a powerful programming tool and their definition and usage raise various interesting and not trivial problems. In this paper, however, we will not consider such a kind of definition. We will rather restrict our attention to the more widely used designations by enumeration. Hereafter, we use the term aggregates to refer to enumerated aggregates.

In imperative programming languages (like the ones mentioned above) data aggregates need to be always completely specified whenever we want to operate on them. This means that all elements of the aggregate must be provided and they must have known values. In contrast, data aggregates 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 aggregates has been amply demonstrated by many examples in the context of declarative programming languages. In this paper we aim to show that partially specified aggregates can be inserted and conveniently exploited also in a more conventional setting, e.g., an imperative O-O language. We do this by presenting and discussing the facilities provided by a library, called JSetL, that supports partially specified lists and sets, within the O-O language Java. More precisely, JSetL integrates within the Java environment the notions of logical (or constrained) variable, (set) unification and constraints that are typical of constraint logic programming languages. In particular, it allows logical variables to occur in list and set data structures and in operations involving them.

In this paper we show how partially specified aggregates can be exploited in Java programs using JSetL by a number of simple examples. It is important to note that, though our discussion is limited to lists and sets, most of the considerations we put forward in this paper apply unaltered to many other data aggregates. 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 briefly reviews data aggregates, in particular lists and sets, as provided by two common programming languages, namely Python and Java. Section 3 introduces the notion of partially specified aggregate, by introducing the notion of logical variable and then by showing how logical variables can be used within aggregates to represent unknown elements or parts of the aggregate. In Section 4 we start considering operations on data aggregates and we address specifically equality between partially specified aggregates, that amounts to deal with (set) unification. We show various examples using JSetL to demonstrate usefulness of partially specified aggregates and unification over them. In Section 5 we show how simple inequality, membership and integer comparison constraints can be advantageously exploited in conjunction with partially specified lists and sets to solve a number of common programming problems. Section 6 takes into account also set variables, whose domains are sets of sets, and it briefly shows how set variables, along with set constraints and partially specified sets, can be advantageously exploited in a number of cases. Finally, in Section 7 we draw some conclusion.

## 2 Completely Specified Aggregates

In this section we briefly review data aggregates as provided by existing imperative programming languages. For the sake of simplicity, examples are drawn from two specific programming languages, namely Python and Java.

Python provides various aggregate types as primitive (called *iterables*): sequences, strings, lists, dictionaries, set and files. Conversely, Java provides only arrays, strings and struct as primitive, while other aggregate types are provided as part of its standard libraries, in particular, the package `java.util`. This package provides the definitions and implementations of a number of common aggregate types such as sets, maps, collections and lists.

As mentioned in Section 1, we will focus on (enumerated) lists and sets only, though similar considerations could apply to other kinds of aggregates as well. Lists and sets differ mainly in that order and repetition of elements are relevant in lists while they do not matter in sets.

Enumerated lists (sets) can be designated in many different ways, such as:

- by a *list (set) literal value* which enumerates all elements of the list (set), e.g., {`1,3,0`}
- by successive *element insertion* operations: in this case, the list (set) is built, element by element, starting from an existing list (set), in particular, from the empty list (set)
- by passing any aggregate object (e.g., a sequence, a list, etc.), to a *list (set) constructor.*

**Example 2.1** *In Python the set* {1,3,0} *can be denoted, e.g., by:*[1] (*i*) *the set literal* {`1,3,0`} *(newer releases only);* (*ii*) *the expression*

```
set().add(0).add(3).add(1)
```

*where* `set()` *denotes the empty set and the method* `add(e)` *adds the specified element* `e` *to the set;* (*iii*) *the expression*

```
set( (3, 1, 0, 3, 0) ),
```

*where* `(3, 1, 0, 3, 0)` *is a sequence literal and* `set` *is the set constructor.*

*Similarly, using the interface* `Set` *of* `java.util`*, the set* {1,3,0} *can be constructed, e.g., by:* (*i*) *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`*;* (*ii*) *the declaration*

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

[1] In order to distinguish the abstract notion of a set from its concrete designations (i.e., in a sense, semantics from syntax), we will use teletype font for the latter.

*where `valuesVector` is a `Vector` object containing all elements of the set `s` and `Vector` is yet another class provided by `java.util`.* □

Note that, as usual in mathematics, order and repetition of elements in a set do not matter. Thus, the set literal {`1,1,2`} and {`2,1`}, both of which are legal in Python, designate exactly the same set, namely the set whose members are the integers `1` and `2`. Similarly, the two Java groups of statements using `java.util`, `s.add(2); s.add(1); s.add(1);` and `s.add(1); s.add(2);` denote the same set.

Elements of a list (set) can be either of arbitrary types, possibly not homogeneous, like in Python, or they must be all of the same type (the *base type*), possibly limited to simple types, like sets in Pascal. `java.util` offers both ways for sets (and lists, as well): `Set` allows elements of the set to be of any type derivable from `Object`, i.e., any non-primitive type, and `Set<T>` where `T` is any non-primitive type, forces elements of the set to be all of the same type `T` or its subtypes. When list (set) elements are allowed to be themselves of structured types, the list (set) can contain other lists (sets) as its elements, i.e., we can have *nested lists (sets)*. For example, in Python, {`'Tom',frozenset(`{`1,5`}`)`} designates a set containing an atomic element of type `string` and a nested set of two (atomic) elements of type `integer`.[2]

A number of basic operations are also provided by the language/library to deal with aggregates. For example, the interface `Set` of `java.util` provides (at least) methods to add/remove elements to/from a set, to check if a value is contained in a set (set membership), to compare two sets for equality and inclusion (subset), to check if a set is empty, and to get the number of elements in a set (set cardinality). Almost the same set of operations are provided in Python.

In all the considered proposals , e.g., Python and `java.util`, however, data aggregates must be always *completely specified* in order to be manipulated.

**Definition 2.2** *An aggregate whose elements are all explicitly enumerated and have a known value is said a* completely specified *aggregate.*

Note that this requirement does not prevent the aggregate to contain variables, e.g., {`x,y`} where `x` and `y` are variables of arbitrary types, but whenever the aggregate is evaluated all variables possibly occurring in it

[2]Python requires all members of a set to be *hashable*. For this reason, {`1,5`} has to be assumed to be a *frozenset*, that is an hashable object, whereas (mutable) sets are not.

must be replaced by their actual values. Analogously, one can use variables in place of entire aggregates in aggregate operations, e.g. `{1,2} + s` where `s` is variable of type `set`, but whenever the expression is evaluated variables are necessarily replaced by their values, i.e., by completely specified aggregates.

In all languages/platforms considered so far, not only lists (sets) must be completely specified but also any list (set) operation requires that all its arguments must be known when evaluated. For example, in the Python expression `{1,2} + s`, where `s` is a variable of type `set`, `s` must be replaced by its value when evaluated. Actually, this is nothing different from what is usually required for arithmetic expressions in conventional languages: all arguments of the expression must have a value when evaluated. This limitation is removed in declarative and constraint programming languages where the notions of expression, variable and evaluation are replaced by those of relation (constraint), logical variable and satisfiability, respectively. We will move to this different context in the next section.

All data aggregates considered so far are completely specified aggregates, in particular, completely specified lists and sets. Note that completely specified aggregates are necessarily finite.

## 3 Partially Specified Aggregates

### 3.1 Logical Variables

Declarative programming languages (e.g., functional and logic programming languages) are based on a notion of variable, often called *logical variable*, that differs from that of imperative programming languages.[3] Logical variables represent unknowns, not memory cells. As such they have no modifiable value stored in them, as ordinary programming languages variables have. Conversely, one can associate values to logical variables through relations (or *constraints*), involving logical variables and values from some specific domains.

The case of singleton domains is particularly interesting.

**Definition 3.1** *When the domain of a variable is restricted to a single value (i.e., the size of the domain is 1), we say that the variable is* bound *(or*

[3]Logical variables are also called *mathematical variables*, *constrained variables*, *unknowns* (e.g., in [1]), *parameter variables* (e.g., in [21]) or, sometimes, simply *variables*, depending on the context and the authors. We have chosen to refer to them as logical variables not to say that they are necessarily related to logics but because this term seems to be sufficiently general though precise.

instantiated *with this value). Otherwise, the variable is* unbound*. With a little abuse of terminology, we will say that the value associated with a bound variable $x$ is* the value of $x$*.*

The equality relation, in particular, allows a precise value to be associated to a logical variable. 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 can 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. Thus declarative languages are characterized by the absence of assignment statements.

Logical variables are found also in conventional languages that support, at some extent, *constraint programming*. For example, the experimental language Alma-0 [1] provides logical variables (called *unknowns*), in addition to conventional programming variables. Logical variables can be manipulated only through constraints. Also libraries that support constraints in the context of O-O languages, such as ILOG [16], Choco [4], JaCoP [17], and Koalog [15], provide some form of logical variables, usually by exploiting the mechanisms for abstract data type definition offered by the language. The recent Java Specification Request for a standard Java Constraint Programming API, being developed under the Java Community Process rules [19], requires that the API provides logical variables (called *constrained* variables, and defined as instances of the class `Var`), to be used as one of the main components of the Constraint Satisfaction Problem definition.

In this paper we will refer to what provided by JSetL, a Java library that supports declarative (constraint) programming in an O-O framework.

A (generic) logical variable can be introduced in JSetL as an instance of the class `LVar`. Basically, `LVar` objects can be manipulated through four different kinds of constraints: equality (`eq`), inequality (`neq`), membership (`in`) and not membership (`nin`) constraints. Membership constraints, in particular, allow to state that the variable can take its value from a given collection of possible values (its domain). Logical variables can be either bound or unbound. When the collection of possible values for a logical variable reduces to a singleton this value becomes the value of the variable and the variable is bound. A precise value for a variable can be specified also when the variable is created. Furthermore the library provides methods to test whether a variable is bound or not, and to get the value of a bound variable (but not to modify it).

**Example 3.2** *Logical variables in JSetL.*

```
LVar x = new LVar();        // an unbound logical variable
LVar y = new LVar(1);       // a bound logical variable
                            // with value 1
```

*Note that the second declaration is equivalent to define an unbound variable* `y` *and to state that the constraint* `y.eq(1)` *must hold for this variable.* □

Values associated with generic logical variables can be of any type. For 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 classes 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 3.3** *Logical lists/sets in JSetL.*

```
LList lr = new LList();    // an unbound logical list
LSet ls_1 = new LSet();    // an unbound logical set
LSet ls_2 = new LSet(s);   // a bound logical set
                           // (where s is the Set object of Ex.2.1)
```

□

## 3.2 Logical Variables in Data Aggregates

When logical variables can occur in data aggregates we have the opportunity to represent partially specified aggregates.

**Definition 3.4** *An aggregate where either some elements have an unknown value or part of the aggregate itself is unknown is said a* partially specified aggregate*.*

Unknown aggregate elements can be represented by unbound logical variables. For example, using Prolog notion, the list `[1,X,Y]`, where `X` and `Y` are unbound variables, denotes a list containing one known element `1` and two unknown elements, denoted `X` and `Y`. Similarly, unknown parts of an aggregate can be represented by unbound logical variables ranging over the domain of aggregates. Using such variables, e.g., in an aggregate constructor, allows us to build open (partially specified) aggregates.

**Definition 3.5** *An aggregate whose elements are only partially enumerated (i.e., part of the aggregate is left unspecified) is said an* open aggregate. *Conversely, when all elements are given, either with or without known values, the aggregate is said a* closed aggregate.

For example, still using Prolog notation, the term `[1,2|Z]`, where `Z` is an unbound list variable, designates an open list containing two known elements 1 and 2 and an unknown remainder part denoted by `Z`. Conversely, the list considered above, `[1,X,Y]`, is a closed list.

Note that, closed aggregates, even if partially specified, have always bounded cardinality, whereas cardinality of open aggregates has only known lower limit. For example, cardinality of the set {`1,X,Y`} can vary from 1 to 3, depending on the values possibly bound to `X` and `Y`. In contrast, cardinality of the set {`1|R`}, where `R` is an unbound set variable, is equal to or greater than 1.

JSetL allows the programmer to create partially specified aggregates, either closed or open.

**Example 3.6** *For example, the The two partially specified lists considered above, i.e., `[1,X,Y]` and `[1,2|Z]`, can be defined in JSetL as follows:*

```
LVar X = new LVar();
LVar Y = new LVar();
LList cl =
   LList.empty().ins(Y).ins(X).ins(1);      // [1,X,Y]
```

*and*

```
LList Z = new LList();
LList ol = Z.ins(2).ins(1);                 // [1,2 |Z]
```

*where `ins` is the element insertion operator for lists in JSetL, and `Z` is a list (logical) variable, i.e., an instance of the class `LList`.* □

Usefulness of partially specified lists is widely accepted in the context of (constraint) logic programming languages. As an extension of this idea, partially specified sets are supplied by extended logic programming languages with sets, such as [18] and [6, 7], and shown to be a powerful data abstraction. Usefulness of (open) partially specified sets is advocated also in specific application areas such as deductive databases, computational linguistics, and knowledge representation. In [11], for instance, CLP($\mathcal{FD}$) is extended to deal with incomplete knowledge on domains to cope with the difficulties connected with required a priori knowledge in FD-like reasoning.

Basic operations on data aggregates include primarily the ability to compare two aggregates for equality. When considering the more general case of partially specified aggregates, i.e. aggregates involving unbound logical variables, this operation amounts to solve a *unification problem*. We will treat in more details this important issue in the next section, where we will show various examples using JSetL to demonstrate usefulness of partially specified aggregates and unification over them.

## 4 (Set) Unification

Roughly speaking, the unification problem is the problem of comparing two objects possibly containing unbound logical variables. Solving the unification problem consists in computing (or simply testing the existence of) an assignment of values to the unbound variables occurring in the given objects which makes them identical. Standard unification assumes that no (semantic) properties hold for the objects to be compared, whereas semantic (or extended) unification assumes that the objects to be compared have some properties (e.g., commutativity) that must be accounted for when establishing if the objects are identical or not, i.e., they must be identical modulo the considered properties. In particular, if the objects to be compared are sets, the (semantic) unification problem becomes a *set unification* problem (see, e.g., [8]).

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 aggregate (i.e., an instance of `LList` or `LSet`). `o2` is either an object of the same type of `o1` or an admissible value for it (i.e., a `Set` object, an `Integer` object, and so on).

We consider list unification first and then set unification.

## 4.1 List Unification

The following is a very simple example of unification over lists in JSetL.

**Example 4.1** *Check whether a list* `l` *contains at least two elements. The problem can be modelled as a unification problem: unify he given list* `l` *with a (partially specified) list representing any list containing at least two elements, i.e.,* $[\mathtt{x}, \mathtt{y} \mid \mathtt{r}]$*, where* `x`*,* `y`*, and* `r` *are unbound logical variables. This is implemented in Java using JSetL by the following method:*

```
public static boolean atLeastTwo(LList l) {
    LVar x = new LVar();
    LVar y = new LVar();
    LList r  = new LList();
    return solver.check(l.eq(r.ins(y).ins(x)));
}
```

*The method requires to check whether the equality* $\mathtt{l} = [\mathtt{x}, \mathtt{y} \mid \mathtt{r}]$ *is satisfied or not. This is done by invoking the method* `check` *of the current constraint solver* `solver`*.* `solver` *is an object of the class* `SolverClass` *which is assumed to be created outside the method* `atLeastTwo`*. Solving the equality calls into play unification: if the two lists are unifiable, then* `check` *returns* true*, otherwise it returns* false*.* □

One advantage of using logical variables and unification, in place of standard programming variables and assignment, 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.

Let us consider the well-known problem of concatenating two lists. We exploit JSetL to implement a method that can be used either to test if a list is the concatenation of two given lists, or to get one list given the other two.

**Example 4.2** *Check whether list* `l3` *is the concatenation of lists* `l1` *and* `l2`*. Also this problem can be modelled as a (list) unification problem. If the first list* `l1` *is the empty list, the other two lists must be equal (and vice versa). Otherwise, the problem is solved by requiring that the given lists satisfy the constraint,* $\mathtt{l1} = [\mathtt{x} \mid \mathtt{l1new}] \ \wedge\ \mathtt{l3} = [\mathtt{x} \mid \mathtt{l3new}]$*, where* `l3new` *is the concatenation of the "shorter" list* `l1new` *and* `l2`*. The following method* `concat` *implements this solution in Java using JSetL.*

```
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* `check` *tests satisfiability of the constraint* `l1` $= [] \wedge$ `l2` $=$ `l3`*. If this is not true, then the constraint* `l1` $= [$`x` $\mid$ `l1new`$] \wedge$ `l3` $= [$`x` $\mid$ `l3new`$]$ *is checked and the result is combined with the result of the recursive call to* `concat`*.* □

The method `concat` can be used not only to compute `l3` out of `l1` and `l2` or to test whether `l3` is the concatenation of `l1` and `l2`, but also to compute either `l1` or `l2` out of the other two lists. No assumption is made about which are input and which are output parameters. As an example, the following code shows the definition of a method, called `prefix`, that uses `concat` to split a list into two sub-lists in order to check whether a list `l1` is a prefix of a list `l2`.

```
public static boolean prefix(LList l1, LList l2) {
    LList l = new LList();
    return Solver.check(concat(l1,l,l2));
}
```

## 4.2 Set Unification

Another advantage of using unification over partially specified aggregates, specifically over sets, is the possibility to exploit the non-determinism embedded in set unification.

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 set unification algorithm employed in JSetL is basically the one proposed in [7], that extends standard unification by embodying the properties of the set constructor operator (namely, permutativity and absorption). This algorithm is inherently non-deterministic. 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 take advantage of set unification over partially specified sets in JSetL.

**Example 4.3** (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 = {X_1,X_2,...,X_n}
    Solver.add(S.eq(I));          // {X_1,X_2,...,X_n} = {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` *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. Precisely,* $S$`.add(`$C$`)` *adds the constraint* $C$ *to the* constraint store *of the solver* $S$*. 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* `S.printElems(' ')` *prints all elements* *of the set* `S` *on the standard output separated by the specified character (blank in this case). For example, if n is 3, the produced output is:*

```
1 2 3
1 3 2
2 1 3
3 1 2
2 3 1
3 2 1
```
□

Various forms of set unification have been used in various application areas, such as (see [8]): deductive databases, AI and its various sub-fields (e.g., Automated Deduction and Natural Language Processing), program analysis and security, resource allocation problems, combinatorial problems in general. A few simple examples, proving usefulness of partially specified sets and set unification, can be found in [8]. The following is one such example, written using JSetL.

**Example 4.4** (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 this 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 4.3.* □

Other examples, as well as an alternative solution of the coloring problem, will be presented after having introduced other constraints in the next section.

### 4.3 Efficiency Considerations

As concern execution efficiency of the proposed solutions it must be noted that, while standard unification (e.g., list unification) can be performed in linear time, the set unification problem has been proved to be a *NP-complete problem* (see [20]). This may lead, in general, to an exponential growth in the complexity of the satisfaction procedure used in JSetL.

However, complexity of the set unification operation, as well as of other operations on data aggregates, depends on which forms of aggregates we have to dealt with, e.g., completely or partially specified, flat or nested, with or without an unknown parts. For instance, as observed in [8], while the decision problem dealing with nested sets involving unbound variables is NP-complete, the set equivalence test of completely specified flat sets, such as $\{a, b, c\}$ and $\{b, c, a\}$, is much easier (see [8] for a more precise analysis). As a matter of fact, various works have been proposed to study, in particular, the simple cases of matching (where variables are allowed to occur in only one of the two set terms which are compared) and unification of *bound simple* set terms, i.e., terms of the form $\{s_1, \ldots, s_n\}$, where each $s_i$ is either a constant or a variable [2, 14].

In order to master the complexity of the constraint solving process, the constraint solver used in JSetL is structured into levels, as described in [3]. In particular, simpler unification problems, i.e., those involving set objects of simpler forms, are dealt with at lower levels, while more complex problems are passed unchanged to upper levels. In general, some classes of problems can be solved at lower levels without incurring in the cost of higher levels. Moreover, when the solver detects inconsistencies at lower levels, the full power of set unification is not required at all, resulting in a more efficient resolution.

As an example, consider the problem of printing all permutations of the numbers from 1 to $n$ presented in Example 4.3. Instead of exploiting the full power of general set unification between (possibly open) partially specified sets, we can observe that one of the two sets is completely known (the set `I` of integers between 1 and $n$), while the other (the set `S`) is a closed set of exactly $n$ variables. Thus we can replace the equality constraint `S` $=$ `I` with the conjunction of constraints $\mathtt{1} \in \mathtt{S} \wedge \mathtt{2} \in \mathtt{S} \wedge \ldots \wedge n \in \mathtt{S}$ which can be added to the constraint store, e.g. by the statement:

```
while(iteratorOverI.hasNext())
       solver.add(S.contains(iteratorOverI.next()));
```

where `S.contains(x)` allows to generate the constraint `x` $\in$ `S` (see next section).

This replacement can be done "by hand" by the programmer, or it can be one of the possible automatic optimizations embedded in the set unification algorithm mentioned above. An improved implementation of this algorithm could recognize special unification cases and deal with them in the most efficient way. The new version of the method `allPermutations` using $n$ membership constraints is executed much more efficiently than the previous version using the more general set equality constraint.

## 5 Inequality and membership constraints

Besides equality, a number of other constraints can be posted that involve logical variables possibly occurring in partially specified aggregates.

In this section, we show how simple inequality, membership and integer comparison constraints can be advantageously exploited in conjunction with partially specified lists and sets to solve a number of common programming problems using Java and JSetL.

**Example 5.1** (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 $ordList$ be a list of $n$ logical variables $X_i$,*

*$i = 1 \ldots n$, where each variable $X_i$ can take as its value one of the integers in $s$. The sorting problem can be expressed by the following constraint*

$$\bigwedge_{i=1}^{n-1} X_i < X_{i+1}$$

*along with the constraints that state that each $X_i$ has domain $s$. This solution is easily implemented in Java using JSetL by the following method:*

```
public static void sortList(Set s) {
    int n = s.size();
    LList lOrd = LList.
    mkIntLList(n);                    // lOrd = {X1,X2,...,Xn}
    Iterator it = lOrd.iterator();    // it = Iterator over LList
    while(it.hasNext())               // forall i ∈ 1..n, Xi.in(s)
        Solver.add((((IntLvar)it.next()).in(s)));
    for(int i=0; i<n-1; i++)          // forall 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 first* `for` *statement allows membership constraints to be added to the* constraint store *of the constraint solver* `solver` *for each variable of the list* `ordList`*. The second* `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 on how efficiently the constraint solver can handle the involved constraints. For instance, if the constraints generated in Example 5.1 are solved by using a simple generate & test approach, computational complexity of the proposed solution is in $\mathcal{O}(n^n)$ and it is clearly unacceptable even for extremely small $n$ (as a matter of fact, the solver should nondeterministically generate all possible assignments of $n$ values to $n$ variables and for each of them test if it is admissible or not).

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 variables to values that the solver has to test constitute paths of a *binomial tree*) and this is much 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, i.e.,

```
Solver.add((((IntLvar)it.next()).dom(s)));
```

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 the logical variable `x` using the specified set `s`, whereas solving the latter (nondeterministically) assigns all possible values to `x`. With the new solution, values will be assigned to unbound variables (*labeling* phase) and propagation will take place only when the solver is finally invoked to check satisfiability of the constraint store.

We would like to remark that, in this case, the implementation of a simple heuristic, e.g., always considering first the minimum element in the domain of each variable during the labeling phase, would speed-up the complexity to polynomial time. The development and testing of appropriate research strategies is one of the possible areas of future research.

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 to the coloring problem of Example 4.4 can be replaced by a slightly more complex but much more efficient solution that exploits inequality constraints and FD constraint solving. The following method `coloring` implements the new solution.

**Example 5.2** (Coloring of a map - FD version) *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. The new solution is implemented as follows:*

```
public static void coloring(LSet regions, LSet map,
    IntLSet colors) {
    Iterator regIt = regions.iterator();
    // LVar's in regions have domain = colors
    while(regIt.hasNext())
        solver.add(((IntLvar)regIt.next()).dom(colors));
    Iterator mapIt = map.iterator();
    // Pairs in map have 1st element ≠ 2nd element
    while(mapIt.hasNext()) {
        LSet aux = (LSet)mapIt.next();
        LVar r1 = (LVar)(aux.get(0));
        LVar r2 = (LVar)(aux.get(1));
        solver.add(r1.neq(r2));
    }
    solver.check();
    return;
}
```

*The first* `while` *statement adds the domain constraints for the problem variables (namely, the variables in* `regions`*), whereas the second* `while` *statement adds constraints between neighboring regions. Invocation of the solver activates the labeling process over variables in* `regions` *and the inequality constraints allow the effect of assigning a value to a variable to be propagated to the domains of the other variables.*

*As a sample invocation of* `coloring`*, if* `map` *and* `colors` *are the set of neighboring regions and the set of colors considered in Example 4.4, then one of the possible computed solutions is:*

```
r1 = 1, r2 = 2, r3 = 1, r4 = 2
```

□

Observe that, if one of the elements in the set `colors` of Example 5.2 is left unspecified, say color `3`, by using an unbound logical variable in its place, say `X`, the problem can still be solved by invoking the same method `coloring`, but the computed answers will involve inequality constraints, such as, for instance:

```
r1 = 1, r2 = 2, r3 = 1, r4 = _X, _X neq 1
```

It is important to notice however that, generally speaking, execution efficiency is not a primary requirement when using the kind of partially specified aggregates 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.

## 6 Set Variables and Set Constraints

Logical variables can range over different domains, among which integer numbers, boolean values, floating-point numbers, sets, each of which is characterized by its own collection of operations and constraints. In particular, when the domain is that of (finite) sets, problem solutions can be modelled in terms of *set (logical) variables*, whose domains are sets of sets (e.g., of integers), and *set constraints*, which implement the most common set-theoretical operations, such as inclusion, union, intersection, and so on. See [13] for a survey on constraints over structured domains.

JSetL provides set variables and the related operations through the class `LSet`, and its subclass `IntLSet` for sets of integers. For example, `LSet s = new LSet()` creates an unbound set variable `s`.

The availability of set variables can be advantageously exploited in conjunction with partially specified sets in a number of cases.

We have already seen that open sets can be constructed by using an unbound set variable to denote the remaining (unknown) part of the set. Furthermore, set constraints can be posted over the unspecified part of the set as well, e.g., to state that it must not contain a certain value. For example, the following Java statement using JSetL

```
LSet s = r.ins(1),
```

where `r` is an unbound set variable, constructs an open set containing `1` plus something else (`r`), while the statement

```
solver.add(r.ncontains(0)).
```

adds the constraint $\mathtt{0} \notin \mathtt{r}$ to the constraint store of the current constraint solver.

Partially specified sets using set variables allow one also to express other set operations simply as set unification problems. For example, using the Prolog-like notation, and assuming $s$ is a set variable and $o$, $o_1$, ..., $o_n$ are objects of any type:

- The set membership operation

$$o \in s$$

  can be implemented as:

$$s = \{o \mid N\},$$

where $N$ is a new unbound set variable.[4]

- The set inclusion operation
$$\{o_1, \ldots, o_n\} \subseteq s$$
can be implemented as:
$$s = \{o_1, \ldots, o_n \mid N\}$$
($N$ as above).

- The following operations involving set cardinality (`size`)
$$\begin{gathered}\texttt{size}(S, N) \wedge N = k\\ \texttt{size}(S, N) \wedge N \leq k\\ \texttt{size}(S, N) \wedge N \geq k\end{gathered}$$
($k$ a non-negative integer constant) can be implemented, respectively, as:
$$\begin{gathered}S = \{x_1, ..., x_n\}, \texttt{allDifferent}(S)\\ S = \{x_1, ..., x_n\}\\ S = \{x_1, ..., x_n \mid N\}, \texttt{allDifferent}(S)\end{gathered}$$
where $x_1, \ldots, x_n$ are unbound logical variables and `allDifferent(`$S$`)` is a constraint stating that all variables of the set $S$ must take different values from each other.

However, while defining other set operations using partially specified sets and set unification is feasible in principle (see, e.g., [5]), efficiency and effectiveness considerations usually lead concrete constraint solvers to provide operations on sets, such as inclusion, union, cardinality and so on, as built-in operations using ad-hoc more efficient implementations.

Finally, note that if nested sets are allowed, set variables can be used also as elements to build partially specified sets of 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);
```

represent any set containing at least another (nested) set `r`.

[4]Note that the same result does not hold for lists.

## 7 Conclusion and Future Work

In this paper we have presented the potential offered by the presence of partially specified aggregates, in particular, lists and sets, in a conventional O-O programming language. Usefulness of partially specified aggregates 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 aggregates.

Although we have considered only lists and sets, there are a number of other kinds of data aggregates, 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 aggregates 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]. Partially specified multisets and a few basic multiset constraints have been implemented within JSetL and will be provided as part of its future releases. Problems connected with the definitions and usage of other partially specified aggregates need to be further investigated.

For the near future, we plan to investigate the potential advantages of using partially specified aggregates in conjunction with nondeterminism, by exploiting the facility offered by JSetL to support nondeterministic user-defined constraint solving [22]. Furthermore, on a more practical side, we plan to concentrate on the development of more efficient (set) constraint solvers and to test their impact on the practical usability of the Java programs using JSetL with partially specified aggregates.

## References

[1] Apt, K.R., Schaerf, A. (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.

[2] Arni, N., Greco, S., and Saccà, D. (1996) Matching of Bounded Set Terms in the Logic Language LDL++. *J. of Logic Programming*, 27(1):73–87.

[3] Bergenti, F., Dal Palù, A., and Rossi, G. (2009) Integrating Finite Domains and Set Constraints into a Set-based Constraint Language. *Fundamenta Informaticae*, 96(3):227–252.

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

[5] Dovier, A. and Rossi, G. (1993) Embedding Extensional Finite Sets in CLP. In *Proc. Int'l Logic Programming Symp., ILPS'93*, 540–556. The MIT Press.

[6] Dovier, A., Omodeo, E. G., Pontelli, E., and Rossi, G. (1996) {log}: A Language for Programming in Logic with Finite Sets. *Journal of Logic Programming*, 28(1):1–44.

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

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

[9] Dovier, A., Piazza, C., and Rossi G. (2008) A uniform approach to constraint-solving for lists, multisets, compact lists, and sets. *ACM Transactions on Computational Logic*, 9(3):1–30.

[10] Henz, M. and Mller, T. (2000) An Overview of Finite Domain Constraint Programming *Proceedings of the Fifth Conference of the Association of Asia-Pacific Operational Research Societies*. Singapore.

[11] Gavanelli M., Lamma E., Mello P., and Milano, M. (2005) Dealing with incomplete knowledge on CLP(FD) variable domains. *ACM Transactions on Programming Languages and Systems*, 27(2):236–263.

[12] Gervet, C. (1997) Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *Constraints*, 1(3):191–244.

[13] Gervet, C. (2006) Constraints over Structured Domains. In Rossi, F., van Beek, P., and Walsh, T., Eds., *Handbook of Constraint Programming*, Elsevier.

[14] Greco S. (1996) Optimal Unification of Bound Simple Set Terms. In *Proc. of Conf. on Information and Knowledge Management*, 326–336, ACM Press.

[15] Koalog. `http://www.koalog.com/php/index.php`.

[16] ILOG (2003) *ILOG Solver 6.0 Reference Manual.*

[17] JaCoP. `http://jacop.osolpro.com/index.php`.

[18] Jayaraman, B. (1992) Implementation of Subset-Equational Programs. *J. of Logic Programming*, 12(4):299–324.

[19] (2010) JSR-331, Java Constraint Programming API (Early Draft). Java Community Process. `http://www.jcp.org`.

[20] Kapur, D. and Narendran, P. (1986) NP-completeness of the set unification and matching problems. In Siekmann, J. H., Ed., *Proc. 8th Int'l Conf. on Automated Deduction*, *LNCS*, v. 230, 489–495. Springer-Verlag.

[21] Leler., Wm. (1988) *Constraint programming languages: their specification and generation.* Addison Wesley.

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