Programming in Java with Restricted Intensional Sets

Maximiliano Cristiá[1] and Gianfranco Rossi[2]

[1] Universidad Nacional de Rosario and CIFASIS, Rosario, Argentina
[2] Università degli Studi di Parma, Parma, Italy
cristia@cifasis-conicet.gov.ar gianfranco.rossi@unipr.it

Abstract. Intensional sets are sets given by a property rather than by enumerating their elements, similar to set comprehensions available in specification languages such as B and MiniZinc. In a previous paper [3] we have presented a decision procedure for a first-order logic language which provides Restricted Intensional Sets (RIS), i.e. a sub-class of intensional sets that are guaranteed to denote finite—though unbounded—sets. In this paper we show how RIS can be exploited as a convenient programming tool also in a more conventional setting, namely, an imperative O-O language. We do this by considering a Java library, called JSetL [14], that integrates the notions of logical variable, (set) unification and constraints that are typical of constraint logic programming languages into the Java language. We show how JSetL is naturally extended to accommodate for RIS and RIS constraints, and how this extension can be exploited, on the one hand, to support a more declarative style of programming, and on the other hand, to effectively enhance the expressive power of the constraint language provided by the library.

1 Introduction and motivations

Intensional sets, also called *set comprehensions*, are sets described by providing a condition φ that is necessary and sufficient for an element x to belong to the set itself, i.e., $\{\, x \;:\; \varphi[x] \,\}$, where x is a variable and φ is a first-order formula containing x. Intensional sets are widely recognized as a key feature to describe complex problems. As a matter of fact, various specification (or modeling) languages provide intensional sets as first-class entities. For example, some form of intensional sets are offered by MiniZinc [12], ProB [11] and Alloy [9]. As concerns programming languages, only relatively few of them support intensional sets. To name some, the general-purpose programming languages SETL [15] and Python, and the Constraint Logic Programming (CLP) languages Gödel [8] and $\{log\}$ [7]. Also conventional Prolog systems provide some facilities for intensional set definition in the form of the `setof` built-in predicate.

The processing of intensional sets in most of these proposals is based on the availability of a *set-grouping* mechanism [16] capable of collecting in a set S all the instances of x satisfying the formula φ. Basically, the implementation of set-grouping exploits the following intended semantics of intensional sets:

$$\begin{aligned}\{x : \varphi[x]\} = S &\leftrightarrow \forall x(x \in S \rightarrow \varphi[x]) \wedge \forall x(\varphi[x] \rightarrow x \in S) \\ &\leftrightarrow \forall x(x \in S \rightarrow \varphi[x]) \wedge \neg\exists x(x \notin S \wedge \varphi[x]).\end{aligned} \tag{1}$$

An example of this approach is $\{log\}$ (pronounced 'setlog'), a freely available extended Prolog system that embodies the CLP language with sets CLP($\mathcal{SET}$) [7, 13]. For instance, while processing the formula[3] $A = \{1, 2\} \wedge S = \{X : X \subseteq A\} \wedge \{3\} \notin S$, $\{log\}$ first collects in S all elements X which are subsets of the set A, i.e. 2^A, then it checks the $\notin$ constraint on S. Though set grouping works fine in many cases, it also have a number of more or less evident drawbacks: (*i*) if the intensional set denotes an infinite set, then set-grouping may be endless; (*ii*) if the formula φ contains unbound variables other than X, then set grouping may incur the well-known problems of the general handling of negation in a logic programming language [4]; (*iii*) if the set of values to be collected is not completely determined (e.g., the solver rewrites φ to a simplified equi-satisfiable form, without generating the actual values for x), then set-grouping may cause the computation to be not complete (and possibly not sound).

Example 1. The following formulas can be written in $\{log\}$ using (general) intensional sets, but $\{log\}$ is not able to process them adequately, due to some of the problems with set-grouping listed above:

- $S = \{2, 4, 6\} \wedge S = \{x : x \in D \wedge x \bmod 2 = 0\}$, i.e., S is the set of all even numbers belonging to an *unknown* set D;
- $C = A \cap B \wedge S = \{x : x \in A \wedge x \in B\} \wedge C \neq S$, i.e., S is the set of all elements in both sets A and B (A and B *unknown* sets).

If, on the contrary, the sets involved in the above formulas are completely specified sets, then $\{log\}$ is able to perform set-grouping and hence to compute the correct answers. □

Set-grouping is not always necessary to deal with intensional sets. For example, given the formula $t \in \{x : \varphi\}$, one could check whether it is satisfiable or not by simply checking satisfiability of $\varphi(t)$, i.e., of the instance of φ which is obtained by replacing x by t; clearly, in this case, there is no need to collect *all* values satisfying φ. A very general proposal along these lines is CLP($\{\mathcal{D}\}$) [5], a CLP language offering arbitrarily nested extensional and intensional sets of elements over a generic constraint domain $\mathcal{D}$, along with basic constraints (namely, $\in$, $\notin$, $=$, and $\neq$,) working with intensional sets. The proposed solver is able to eliminate occurrences of intensional sets from the constraints *without* explicit enumeration of the sets. The generality of the intensional set formers supported in this proposal, however, may cause some of the drawbacks mentioned above (namely, problems concerned with intensional sets denoting infinite sets and with the general use of negation). As observed in [5], the presence of undecidable constraints such as $\{x : p(x)\} = \{x : q(x)\}$, where p and q can have an infinite number of solutions, "prevents us from developing a parametric and *complete*

[3] In order to not burden the presentation too much, $\{log\}$ formulas will be written using the usual mathematical notation rather than its concrete syntax.

solver". As a matter of fact, no working implementation of this proposal has been developed.

Recently, Cristia and Rossi [3] proposed a narrower form of intensional sets, called *Restricted Intensional Sets* (RIS), and a complete solver to deal with basic set-theoretical operations on them in a way similar to [5] (i.e., not using set-grouping). RIS have similar syntax and semantics to the set comprehensions available in the formal specification languages Z and B, i.e. $\{x : D \mid \Psi \bullet \tau\}$, where D is a set, Ψ is a formula over a first-order theory $\mathcal{X}$, and τ is a term involving x. The semantics of the RIS $\{x : D \mid \Psi \bullet \tau\}$ is "the set of terms $\tau[x]$ such that x is in D and Ψ holds for x". RIS have the restriction that D must be a finite set and Ψ is a quantifier-free constraint in $\mathcal{X}$, for which we assume a complete solver to decide its satisfiability is available. It is important to note that, although RIS are guaranteed to denote finite sets, nonetheless, RIS can be not completely specified. In particular, as the domain can be a variable or a partially specified set, RIS are finite but *unbounded*.

The work in [3] is mainly concerned with the definition of the constraint language and its solver, and the proof of soundness and completeness of the constraint solving procedure. In this paper, our main aim is to explore *programming* with intensional sets. Specifically, we are interested in exploring the potential of using RIS in the more conventional setting of imperative O-O languages. To this purpose, we consider JSetL [14], a Java library that integrates the notions of logical variable, (set) unification and constraints that are typical of constraint logic programming languages into the Java language. First, we show how JSetL is naturally extended to accommodate for RIS. Then, we show with a number of simple examples how this extension can be exploited, on the one hand, to support a more declarative style of programming, and on the other hand, to effectively enhance the expressive power of the constraint language provided by the library. Observe that though we are focusing on Java, the same considerations could be easily ported to other O-O languages (such as C++).

The paper is organized as follows. Sect. 2 introduces RIS informally through some examples. Sect. 3 briefly reviews the JSetL library and Sect. 4 presents the extension of JSetL with RIS. In Sect. 5 we start showing examples using JSetL to demonstrate the usefulness of RIS and RIS constraints to support declarative programming. Sect. 6 shows how RIS can be used to define and manipulate partial functions. In Sect. 7 we consider some extensions to RIS and we present examples showing their usefulness. Finally, in Sect. 8 we draw some conclusion.

2 An Informal Introduction to RIS

In this section we introduce RIS in an informal, intuitive way, through a number of simple examples.

The language that embodies RIS, called $\mathcal{L}_{\mathcal{RIS}}$, is a quantifier-free first-order logic language which provides both RIS and extensional sets, along with basic operations on them, as primitive entities of the language. $\mathcal{L}_{\mathcal{RIS}}$ is *parametric* with respect to an arbitrary theory $\mathcal{X}$, for which we assume a decision procedure

for any admissible $\mathcal{X}$-formula is available. Elements of $\mathcal{L}_{\mathcal{RIS}}$ sets are the objects provided by $\mathcal{X}$, which can be manipulated through the primitive operators that $\mathcal{X}$ offers (at least, $\mathcal{X}$-equality). Hence, RIS in $\mathcal{L}_{\mathcal{RIS}}$ represent *untyped unbounded finite hybrid sets*, i.e. unbounded finite sets whose elements are of any sort. For the sake of convenience, we assume that the language of $\mathcal{X}$, $\mathcal{L}_{\mathcal{X}}$, provides the constant, function and predicate symbols of the theories of the integer numbers and ordered pairs.

$\mathcal{L}_{\mathcal{RIS}}$ provides the following set terms: *a*) the empty set, noted $\emptyset$; *b*) *extensional sets*, noted $\{x \sqcup A\}$, where x, called *element part*, is an $\mathcal{X}$-term, and A, called *set part*, is an $\mathcal{L}_{\mathcal{RIS}}$ set term; and *c*) *restricted intensional sets* (RIS), noted $\{c : D \mid F \bullet P[c]\}$, where c, called *control term*, is an $\mathcal{X}$-term; D, called *domain*, is an $\mathcal{L}_{\mathcal{RIS}}$ set term; F, called *filter*, is an $\mathcal{X}$-formula; and P, called *pattern*, is an $\mathcal{X}$-term containing c. Both extensional sets and RIS can be partially specified because elements and sets can be variables. As a notational convenience, $\{t_1 \sqcup \{t_2 \sqcup \cdots \{t_n \sqcup t\} \cdots\}\}$ (resp., $\{t_1 \sqcup \{t_2 \sqcup \cdots \{t_n \sqcup \emptyset\} \cdots\}\}$) is written as $\{t_1, t_2, \ldots, t_n \sqcup t\}$ (resp., $\{t_1, t_2, \ldots, t_n\}$). When useful, the domain D can be represented also as an interval $[m, n]$, m and n integer constants, which is intended as a shorthand for $\{m, m+1, \ldots, n\}$. $\mathcal{RIS}$-literals are of the form $A = B$, $A \neq B$, $e \in A$ or $e \notin A$, where A and B are set terms and e is an $\mathcal{X}$-term. $\mathcal{RIS}$-formulas are built from $\mathcal{RIS}$-literals using conjunction and disjunction.

An extensional set $\{x \sqcup A\}$ is interpreted as $\{x\} \cup A$. A RIS term is interpreted as follows: if $x_1, \ldots, x_n$ $(n > 0)$ are all the variables occurring in c, then $\{c : D \mid F \bullet P[c]\}$ denotes the set $\{y : \exists x_1 \ldots x_n (c \in D \wedge F \wedge y =_{\mathcal{X}} P[c])\}$. Note that $x_1, \ldots, x_n$ are bound variables whose scope is the RIS itself. Also note that equality between y and the pattern P requires equality of the theory $\mathcal{X}$.

In order to devise a decision procedure for $\mathcal{L}_{\mathcal{RIS}}$, the control term c and the pattern P of a RIS are restricted to be of specific forms. Namely, if x and y are variables ranging on the domain of $\mathcal{X}$, then c can be either x or (x, y), while P can be either c or (c, t) or (t, c), where t is any (uninterpreted/interpreted) $\mathcal{X}$-term, possibly involving the variables in c. As it will be evident from the various examples in the next sections, in spite of these restrictions, $\mathcal{L}_{\mathcal{RIS}}$ is still a very expressive language.

Example 2. The following are $\mathcal{RIS}$-literals involving RIS terms:

- $\{x : [-2, 2] \mid x \bmod 2 = 0 \bullet x\} = \{-2, 0, 2\}$
- $(5, y) \in \{x : D \mid x > 0 \bullet (x, x * x)\}$, where y and D are free variables
- $(5, 0) \notin \{(x, y) : \{P \sqcup R\} \mid y \neq 0 \bullet (x, y)\}$, where P and R are free variables, and $\{P \sqcup R\}$ is a set term denoting the set $\{P\} \cup R$. □

When the pattern is the control term and the filter is *true*, they can be omitted (as in Z and B), although one must be present.

$\mathcal{L}_{\mathcal{RIS}}$ provides a complete constraint solver which is able to decide satisfiability of any $\mathcal{L}_{\mathcal{RIS}}$ formulas. Precisely, the solver reduces any input formula Φ to either *false* (hence, Φ is unsatisfiable), or to an equi-satisfiable disjunction of formulas in a simplified form, called the *solved form*, which is guaranteed to be satisfiable (hence, Φ is satisfiable). If Φ is satisfiable, the answer computed

by the solver constitutes a finite representation of all the concrete (or ground) solutions of the input formula.

Example 3 (Constraint solving with RIS).

- The second formula of Ex. 2 is rewritten by the $\mathcal{L}_{\mathcal{RIS}}$ solver to the solved form formula $y = 25 \wedge D = \{5 \sqcup N_1\}$, where the second equality states that D must contain 5 and something else, denoted N_1.
- The first formula of Ex. 1 can be written using RIS as $S = \{2, 4, 6\} \ \wedge \ S = \{x : D \mid x \bmod 2 = 0\}$; this formula is rewritten by the solver to a solved form formula containing the constraint $D = \{2, 4, 6 \sqcup N_1\} \wedge \{x : N_1 \mid x \bmod 2 = 0\} = \emptyset$, where the second equality states that N_1 cannot contain even numbers (note that this constraint has the obvious solution $N_1 = \emptyset$). □

It is worth noting that the $\mathcal{L}_{\mathcal{RIS}}$ solver is able to correctly solve all formulas of Ex. 1, written using RIS. Moreover, note that the formula $\{x : p(x)\} = \{x : q(x)\}$, which is undecidable when p and q have an infinite number of solutions, can be "approximated" using RIS as $\{x : D_1 \mid p(x)\} = \{x : D_2 \mid q(x)\}$, D_1, D_2 variables, which is a solved form formula admitting at least one solution, namely $D_1 = D_2 = \emptyset$; hence it is simply returned unchanged by the solver.

3 JSetL

JSetL [14] is a Java library that combines the object-oriented programming paradigm of Java with valuable concepts of CLP languages, such as logical variables, lists, unification, constraint solving and non-determinism. The library provides also sets and set constraints like those found in CLP($\mathcal{SET}$). Unification may involve logical variables, as well as list and set objects (i.e., set unification [6]). Set constraints are solved using a complete solver that accounts for partially specified sets (i.e., sets containing unknown elements). Equality, inequality and comparison constraints on integers are dealt with as Finite-Domain Constraints, like in CLP(FD) [2].

JSetL has been used as one of the first six implementations for the standard Java Constraint Programming API defined in the Java Specification Request JSR-331 [10] (see for instance `http://openrules.com/jsr331/JSR331.UserManual.pdf`). The JSetL library can be downloaded from the JSetL's home page at `http://cmt.math.unipr.it/jsetl.html`.

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. 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. Moreover, a logical variable can have an optional external name (i.e., a string) which can be useful when printing the variable and the possible constraints involving it.

Example 4 (Logical variables).

```
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       □
```

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 $<$ (`lt`), $\leq$ (`le`), 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 5 (Logical lists/sets).

```
LList l = new LList();  // an unbound logical list
LSet r = new LSet("r"); // an unbound logical set with external name "r"
LSet s1 = LSet.empty().ins(2).ins(1); // the set {1,2}
LVar x = new LVar("x");
LSet s2 = r.ins(x);     // the set {x} ∪ r                              □
```

`ins` is the *element insertion* method for `LSet`s: `s.ins(o)` returns the new logical set whose elements are those of the set `s` $\cup$ {`o`}. In particular, the last statement in Ex. 5 creates a partially specified set `s` with an unknown element `x` and an unknown rest `r` (i.e., {`x` | `r`}, using a Prolog-like notation).

A JSetL *constraint solver* is an instance of the class `SolverClass`. Basically, it provides methods for adding constraints to its *constraint store* (e.g., the method `add`) and to prove constraint satisfiability (e.g., the method `solve`). If `solver` is a solver, Γ is the collection of constraints stored in its constraint store (possibly empty), and `c` is a constraint, then `solver.solve(c)` checks whether $\Gamma \wedge$ `c` is satisfiable or not: if it is, then the constraint store is modified so to represent the computed constraint in solved form; otherwise an exception `Failure` is raised.

Example 6 (Constraint solving).

```
LSet s1 = LSet.empty().ins(2).ins(1);  // the set {1,2}
LVar x = new LVar("x"), y = new LVar("y");
LSet s2 = LSet.empty().ins(y).ins(x);  // the set {x,y}
SolverClass solver = new SolverClass();
```

```
solver.add(s1.eq(s2).and(x.neq(1)));   // the constraint s1=s2 ∧ x≠1
solver.solve();
x.output(); y.output();
```

where the method `output` is used to print the value possibly bound to a logical object. Executing this code fragment will output: `x = 2, y = 1`. □

4 Adding RIS to JSetL

In this section we show how RIS are added to JSetL. The new version of the JSetL library can be downloaded from the JSetL's home page. All sample Java programs shown in this and in the next sections have been tested using the new version and are available on-line.

RIS are added to JSetL by defining a new class, named `Ris`. `Ris` extends `LSet`, hence `Ris` objects can be used as logical sets, and all methods of `LSet` are inherited by `Ris`. Basically, a `Ris` object (i.e. an instance of `Ris`) is created by specifying its control term (an object of type `LVar` or `Pair`), its domain (an object of type `LSet`), its filter (an object of type `Constraint`), and its pattern (an object of type `LVar` or `Pair`). The pattern can be omitted if it coincides with the control term.

Example 7 (`Ris` object creation). The first RIS of Ex. 2 is created in JSetL as follows:

```
IntLVar x = new IntLVar();
LSet d = new LSet(new Interval(-2,2));
Constraint f = x.mod(2).eq(0);
Ris ris = new Ris(x,d,f);   // {x:[-2,2] | x mod 2 = 0 • x}
```

where `Interval` and `Constraint` are classes provided by JSetL with their obvious meaning. □

The new methods added by `Ris` to those inherited from `LSet` are basically constraint methods and general utility methods working on `Ris` objects.

Constraint methods provided by `Ris` implement the constraints $=$ (method `eq`) and $\neq$ (method `neq`). Moreover, the constraint methods `in` and `nin` of classes `LVar` and `Pair` are extended so to accept `Ris` objects as their arguments.

Example 8 (RIS constraints). If `ris` is the `Ris` object created in Ex. 7, then the following are possible RIS constraints posted on `ris`:

```
LSet s = LSet.empty().ins(2).ins(0).ins(-2);
solver.add(ris.eq(s));    // {x:[-2,2] | x mod 2 = 0 • x} = {-2,0,2}
LVar y = new LVar(1);
solver.add(y.nin(ris));   // 1 nin {x:[-2,2] | x mod 2 = 0 • x}
```

□

The class `Ris` provides also a number of general utility methods, such as `isBound()`, which returns true iff the domain of the RIS is bound to some value, and `expand` which returns the `LSet` object representing the extensional set denoted by the RIS (i.e., it performs set grouping) or it raises an exception if the domain of the RIS is not a completely specified set.

Example 9 (RIS constraints). If `ris` is the `Ris` object created in Ex. 7, then the corresponding extensional set `S` is computed and printed as follows:

```
LSet S = new LSet;
S = ris.expand().setName("S");
S.output();
```

whose execution yields `S = {2,0,-2}`. □

In JSetL the language of RIS and the language of the parameter theory $\mathcal{X}$ are completely amalgamated. Thus, it is possible to use constraints of the latter together with constraints of the former, as well as to share logical variables of both. The following in an example that uses this feature to prove a general property about sets.

Example 10. Check the property of set intersection as stated by the second formula of Ex. 1.

```
LSet A = new LSet(), B = new LSet(), C = new LSet();
solver.add(C.inters(A,B));      // the constraint C=A ∩ B
LSet D = new LSet();
LVar X = new LVar();
Ris S = new Ris(X,A,X.in(B));   // the RIS {X:A | X ∈ B}
solver.add(S.neq(C));           // the constraint {X:A | X ∈ B} ≠ C
```

Calling `solver.solve()` causes an exception `Failure` to be thrown (i.e., the formula is found to be *false*). □

Observe that constraints in JSetL are predicates, not functions; hence we write, for instance, `C.inters(A,B)`, instead of `C.eq(A.inters(B))`, to denote the predicate $inters(A, B, C)$ which is true iff $C = A \cap B$.

5 Declarative programming with RIS

Intensional set definition represents a powerful tool for supporting a declarative programming style, as pointed out for instance in [7].

A first interesting application of RIS to support declarative programming is to represent *Restricted Universal Quantifiers* (RUQ). The formula $\forall x \in D : F[x]$ can be easily represented by using a RIS as follows: $D = \{x : D \mid F[x]\}$. When D is a known set, solving this formula amounts to check whether $F[x]$ holds for all x in D. For instance, $D = \{1, 2, 3\} \wedge D = \{x : D \mid x > 0\}$ is true, whereas $D = \{1, -2, 3\} \wedge D = \{x : D \mid x > 0\}$ is false.

Since the $\mathcal{L}_{\mathcal{RIS}}$ solver can decide satisfiability of such formulas, then we have a decision procedure for a fragment of first-order logic with quantifiers.

The following are Java programs that solve simple—though not trivial—problems using JSetL with RIS. Basically, their solution is expressed declaratively as a formula using RUQ.

Example 11. Compute and print the minimum of a set of integers `S`.

```
public static LVar minValue(LSet S)} throws Failure {
   IntLVar x = new IntLVar();
   IntLVar y = new IntLVar();
   Ris ris = new Ris(x,S,y.le(x));
   solver.add(y.in(S).and(S.eq(ris)));
   solver.solve();
   return y; }
```

The method `minValue` posts the constraint $y \in S \wedge S = \{x : S \mid y \leq x\}$. The solver, non-deterministically binds a value from S to y and then it checks if the property $y \leq x$ is true for all elements x in S. If this is not the case, the solver backtracks and tries a different choice for y. A possible call to this method is:

```
int[] sampleSetElems = {8,4,6,2,10,5};
LVar min = minValue(new LSet(sampleSetElems)).setName("min");
min.output();
```

and the printed answer is `min = 2`. □

It is important to observe that operations on RIS are dealt with as real constraints. This implies, among other things, that the order in which constraints are posted to the solver is irrelevant.

More generally, the use of constraints allows to compute with only partial specified aggregates [1]. For example, the set passed to the method `minValue` can be $\{8, 4, 6, x\}$, where x is an uninitialized logical variable, or even it can contain an unknown part, e.g. (using the abstract notation), $\{8, 4 \sqcup R\}$, where R is an uninitialized `LSet` object. In the first case, i.e., with `S` equal to $\{8, 4, 6, x\}$, the solver is able to non-deterministically generate three distinct answers, one with $x = 4, \texttt{min} = 4$, another with $x \leq 4, \texttt{min} = x$, and another with $x \geq 4, \texttt{min} = 4$.

Another example that shows the use of RIS to define in a declarative way a universal quantification is the following simple instance of the well-known map coloring problem.

Example 12. Given a cartographic map of n regions $r_1, \ldots, r_n$ and a set $\{c_1, \ldots, c_m\}$ of colors find an assignment of colors to the regions such that no two neighboring regions have the same color.

Each region can be represented as a distinct logical variable and a map as a set of unordered pairs (hence, sets) of variables representing neighboring regions. An assignment of colors to regions is represented by an assignment of values (i.e., the colors) to the logical variables representing the different regions.

```
public static void coloring(LSet regions, LSet map, LSet colors)
throws Failure {
   solver.add(regions.subset(colors));
   LSet p = new LSet();
   Ris ris = new Ris(p, map, p.size(2));
   solver.add(map.eq(ris));
   solver.solve(); }
```

The method `coloring` posts the constraint $regions \subseteq colors \wedge map = \{p : map \mid |p| = 2\}$. The first conjunct exploits the `subset` constraint to nondeterministically assign a value to all variables in `regions`. The second equality requires that all pairs of regions in the map have cardinality equal to 2, i.e., all pairs have distinct components. If `coloring` is called with `regions` = {`r1,r2,r3`}, `r1`, `r2`, `r3` uninitialized logical variables, `map` = {{`r1,r2`},{`r2,r3`}}, and `colors` = {`"red", "blue"`}, the invocation terminates with success, and `r1`, `r2`, `r3` are bound to `"red"`, `"blue"`, and `"red"`, respectively (actually, also the other solution which binds `r1`, `r2`, `r3` to `"blue"`, `"red"`, and `"blue"`, respectively, can be computed through backtracking). □

This solution uses a pure "generate & test" approach; hence it quickly becomes very inefficient as soon as the map becomes more and more complex. However, it may represent a first "prototype" whose implementation can be subsequently refined (e.g., by modeling the coloring problem in terms of Finite Domain (FD) constraints and using the more efficient FD solver provided by JSetL), without having to change its usage. On the other hand, this solution allows to exploit the flexible use of constraints and partially specified sets. As a matter of fact, the same program can be used to solve related problems; e.g., given a map and a partially specified set of colors, find which constraints the unknown colors must obey in order to obtain an admissible coloring of the map.

The next program shows another example where RIS are used to check whether a property holds for all elements of a set, but using a different kind of equality constraint.

Example 13. Check whether `n` is a prime number or not:[4]

```
public static Boolean isPrime(int n)} {
   if (n <= 0) return false;
   IntLVar N = new IntLVar(n);
   IntLVar x = new IntLVar();
   LSet D = new LSet(new Interval(2,n/2));
   Ris ris = new Ris(x,D,N.mod(x).eq(0));
   solver.add(ris.eq(LSet.empty()));
   return solver.check(); }
```

The method `isPrime` posts the constraint $\{x : [2, N \; div \; 2] \mid N \; mod \; x = 0\} = \emptyset$. The equality between the RIS and the empty set ensures that there is no x in the interval $[2, N \; div \; 2]$ such that $N \; mod \; x = 0$ holds. If, for instance, `n` is 101, then the call to `isPrime` returns `true`. □

6 Using RIS to define partial functions

Another important application of RIS is to define *(partial) functions* by giving their domains and the expressions that define them. In general, a RIS of the

[4] `s.check()` differs from `s.solve()` in that the latter raises an exception if the constraint in the constraint store of `s` is unsatisfiable, whereas the former returns a boolean value indicating whether the constraint is satisfiable or not.

form $\{x : D \mid F \bullet (x, f(x))\}$, where f is any function symbol belonging to the language $\mathcal{L}_{\mathcal{X}}$, defines a partial function. Such a RIS contains ordered pairs whose first components belong to D which cannot have duplicates (because it is a set). Then, since no two pairs share the same first component, the RIS is a function.

Given that RIS are sets, then, in $\mathcal{L}_{\mathcal{RIS}}$, functions are sets of ordered pairs. Therefore, through standard set operators, functions can be evaluated, compared and point-wise composed; and by means of constraint solving, the inverse of a function can also be computed. The following examples illustrate these ideas in the context of Java with JSetL.

Example 14. The square of an integer n.

```
IntLVar x = new IntLVar();
LSet D = new LSet();
Ris Sqr = new Ris(x,D,Constraint.TRUE,new Pair(x,x.mul(x));
```

The RIS `Sqr` defines the set of all pairs $(x, x*x)$, where x belongs to an (unknown) set D. This function can be "evaluated" in a point n, and the result sent to the standard output, by executing the following code:

```
IntLVar y = new IntLVar("y");
solver.solve(new Pair(n,y).in(Sqr));
y.output();
```

that is, `y` is the image of `n` through function `Sqr`. If, for instance, `n` has value 5 (e.g., `int n = 5`), then the printed result is `y = 25`. □

Note that `Sqr` is a set of ordered pairs as its pattern is an ordered pair. Besides, `Sqr` is a partial function because each of its first components never appears twice, since they belong to the set D. Moreover, note that the RIS domain, D, is left underspecified as a variable.

The same RIS of Ex. 14 can be used also to calculate the inverse of the square function, that is the square root of a given number. To obtain this, it is enough to post and solve the constraint

```
solver.solve(new Pair(y,n).in(Sqr));
```

If, for instance, `n` has value 25, the computed result is `y = unknown - Domain: {-5,5}`, stating that the possible values for `y` are `-5` and `5`.

The interesting aspect of using RIS for defining functions is that RIS are sets and sets are data. Thus, we have a simple way to deal with functions as data. In particular, since `Ris` objects can be passed as arguments to a function, we can use RIS to write generic functions that take other functions as their arguments. The following is an example illustrating this technique.

Example 15. The following method takes as its arguments an array of integers `A` and a function `f(x)` and updates `A` by applying `f` to all its elements.

```
public static void mapList(int[] A,Ris f) throws Failure {
   for(int i=0; i<A.length; i++) {
       IntLVar y = new IntLVar();
       solver.solve(new Pair(A[i],y).in(f));
       A[i] = y.getValue();
   } }
```

If, for instance, the array passed to `mapList` is {3,5,7} and `f` is the RIS `Sqr` of Ex. 14, then the modified array is {9,25,49}. □

7 Extended RIS

To guarantee that the constraint solver is indeed a decision procedure a number of restrictions are imposed on the form of RIS in [3]. Specifically: (*i*) the control term and the pattern of a RIS are restricted to be of specific forms—roughly speaking, variables and pairs, see Sect. 2; (*ii*) the filter of a RIS cannot contain "local" variables, i.e. existentially quantified variables declared inside the RIS; (*iii*) recursively defined RIS such as $X = \{x : D \mid \Psi[X] \bullet \tau\}$ are not allowed.[5]

Although compliance with these restrictions is important from a theoretical point of view, in practice there are many cases in which they can be (partially) relaxed without compromising the correct behavior of programs using RIS.

7.1 General patterns

As noted in [3], the necessary and sufficient condition for patterns to guarantee correctness and completeness of the constraint solving procedure is that patterns be bijective functions. All the admissible patterns of $\mathcal{L}_{\mathcal{RIS}}$ are bijective patterns. Besides these, however, other terms can be bijective patterns. For example, $x + n$, n constant, is also a bijective pattern, though it is not allowed in $\mathcal{L}_{\mathcal{RIS}}$. Conversely, $x * x$ is not bijective as x and $-x$ have $x * x$ as image (note that $(x, x * x)$ is indeed a bijective pattern allowed in $\mathcal{L}_{\mathcal{RIS}}$).

Unfortunately, the property for a term to be a bijective pattern cannot be easily syntactically assessed. Thus in [3] we adopted a more restrictive definition of admissible pattern. From a more practical point of view, as in JSetL, we can admit also more general patterns. In particular, we allow patterns to be any *integer logical expression* involving the RIS control variable. An integer logical expression in JSetL is created by using methods such as `sum`, `mul`, etc., applied to `IntLVar` objects, and returning `IntLVar` objects (e.g., `x.mul(x)`, where `x` is an uninitialized `IntLVar`).

If the expression used in the RIS pattern defines a bijective function (e.g., `x.sum(2)`) then dealing with the RIS is anyway safe; otherwise, it is not safe in general, but it may work correctly in many cases.

Example 16. Compute the set of squares of all even number in [1,10].

```
IntLVar x = new IntLVar();      //the RIS {x:[1,10] | x mod 2=0 • x*x}
LSet D = new LSet(new Interval(1,10));
Ris ris = new Ris(x,D,x.mod(2).eq(0),x.mul(x));
LSet Sqrs = ris.expand();
```

Executing this code will correctly bind `Sqrs` to {4,16,36,64,100}. □

[5] Note that, on the contrary, a formula such as $X = \{D[X] \mid \Psi \bullet \tau\}$ is an admissible constraint, and it is suitably handled by the $\mathcal{L}_{\mathcal{RIS}}$ decision procedure.

Allowing more general forms of patterns (and, possibly, control terms) is planned as a future extension for RIS in general, and for the implementation of RIS in JSetL, as well.

7.2 RIS with local variables

Allowing local variables in RIS raises major problems when the formula representing the RIS filter has to be negated during RIS constraint solving (basically, negation of the RIS filter is necessary to assure that an element that does not satisfy the filter does not belong to the RIS itself). In fact, this would require that the solver is able to deal with quite complex universally quantified formulas, which is usually not the case (surely, it is not the case for the JSetL solver). Thus, to avoid such problems a priori, the RIS filter cannot contain local variables.

However, as already observed for RIS patterns, in practice there are cases in which we can relax some restrictions on RIS without losing the ability to correctly deal with such more general RIS constraints.

Thus, in JSetL, we allow the user to specify that some (logical) variables in the RIS filter are indeed local variables. This is achieved—as a temporary solution—by using a special syntax for the external name of the logical variable (namely, the name must start with an underscore character).

Example 17. If R is a set of ordered pairs and D is a set, then the subset of R where all the first components belong to D can be defined as follows:

```
LSet D = new LSet();                  //the RIS
LSet R = new LSet();                  //{x:D | ∃y((x,y) ∈ R • (x,y))}
LVar x = new LVar();
LVar y = new LVar("_y");
Pair P = new Pair(x,y)
Ris ris = new Ris(x,D,P.in(R),P);
```

If we try to solve the constraint

```
solver.solve(new Pair(1,2).in(ris).and(new Pair(3,4).in(ris)))
```

i.e., $(1,2) \in \texttt{ris} \wedge (3,4) \in \texttt{ris}$, then the solver terminates with success, binding (as its first solution) `D` to $\{1, 3 \sqcup N_1\}$ and `R` to $\{(1,2), (3,4) \sqcup N_2\}$, N_1, N_2 new fresh variables. □

In the above example, `y` is a local variable. If `y` is not declared as local, then the same call to `solver.solve` will terminate with failure, since `y` is dealt with as an external variable and the first constraint, $(1,2) \in \texttt{ris}$, binds `y` to 2 so that the second constraint $(3,4) \in \texttt{ris}$ fails.

Anyway, it can be observed that many uses of local variables can be avoided by a proper use of the control term and pattern of a RIS. For example, the extended RIS of Ex. 17 can be expressed with a RIS without local variables as follows: $\{(x,y) : R \mid x \in D\}$. Hence, the planned extension of the admissible control terms and patterns for RIS can also be useful to alleviate the problem of local variables in RIS filters.

7.3 Recursive RIS

The class `Ris` extends the class `LSet`; hence it is possible, at least in principle, to use `Ris` objects inside the RIS filter formula in place of `LSet` objects. This allows, among other things, to define *recursive restricted intensional sets* (RRIS).

Of course, the presence of recursive definitions may compromise the finiteness of RIS and hence the decidability of the formulas involving them. Therefore RRIS are prohibited in the base language of RIS, $\mathcal{L}_{\mathcal{RIS}}$. In practice, however, their availability can considerably enhance the expressive power of the language and hence RRIS are allowed in the extended language supported by JSetL. Programmers are responsible of guaranteeing termination.

The following is an example using a RRIS. Since RRIS has not yet been fully developed and tested in JSetL we will write programs dealing with them by using only the abstract notation.

Example 18 (Factorial of a number n).

$$\begin{aligned} &Fact = \{(0,1) \sqcup Fact_1\} \wedge \\ &Fact_1 = \{x : D \\ &\qquad \mid \exists y, z(x > 0 \wedge (x-1, z) \in Fact \wedge y = x * z \bullet (x, y))\} \end{aligned} \tag{2}$$

Note that the domain, D, is left underspecified, and recursion is simply expressed as a set membership predicate over the same set being defined: $(x-1, z) \in Fact$ means that z is the factorial of $x-1$. If we conjoin for example the constraint $(5, f) \in Fact$ then the solver will return $f = 120$. As usual in declarative programming, there is no real distinction between inputs and outputs. Therefore if we conjoin to formula (2) the constraint $(n, 120) \in Fact$ then the solver will return $n = 5$. □

RRIS are not yet fully supported in JSetL, but their inclusion, which has already been successfully experimented in $\{log\}$, should be feasible without major problems.

8 Conclusion and future work

In this paper we have presented an extension of the Java library JSetL to support RIS, and we have shown, through a number of simple examples, the usefulness of RIS as a powerful data and control abstraction. Although efficiency is not our main concern, the implementation of RIS in $\{log\}$ has proven to be efficient enough as to compete with mainstream tools [3]. Hence, we expect similar results of the implementation of RIS in the JSetL library. As future work, it can be interesting: (a) on the more theoretical side, trying to extend the language of RIS for which we are able to provide a correct and complete solver, e.g. by enlarging the collection of set and relation operators dealing with RIS (for now limited to equality and membership); (b) on the more practical side, trying to incorporate in the implementation of RIS within the JSetL library all the extensions mentioned in Sect. 7, which although falling outside of the decision procedure, turn out to be of great interest in practice.

Acknowledgments We wish to thank Andrea Guerra and Andrea Fois for their contribution to the implementation of RIS in JSetL. This work has been partially supported by GNCS "Gruppo Nazionale per il Calcolo Scientifico".

References

1. Bergenti, F., Chiarabini, L., Rossi, G.: Programming with Partially Specified Aggregates in Java, Computer Languages, Systems & Structures, 37(4), 178–192 (2011).
2. Codognet, P., Diaz, D.: Compiling constraints in CLP(FD). Journal of Logic Programming, 27(3):185–226 (1996).
3. Cristiá, M., Rossi, G.: A Decision Procedure for Restricted Intensional Sets. In de Moura, L. (Ed.) Automated Deduction 26th Int'l Conf. on Automated Deduction (CADE 26). LNCS, v. 10395, 185-201, Springer (2017).
4. Dovier, A., Pontelli, E., Rossi, G.: Constructive Negation and Constraint Logic Programming with Sets. New Generation Computing, 19(3):209–255, 2001.
5. Dovier, A., Pontelli, E., Rossi, G.: Intensional sets in CLP. In: Palamidessi, C. (ed.) Proc. 19th Int'l Conf. on Logic Programming, LNCS, v. 2916, 284–299. Springer (2003).
6. Dovier, A., Pontelli, E., Rossi, G.: Set unification. Theory and Practice of Logic Programming 6(6), 645–701 (2006).
7. Dovier, A., Piazza, C., Pontelli, E., Rossi, G.: Sets and constraint logic programming. ACM Trans. Program. Lang. Syst. 22(5), 861–931 (2000).
8. Hill, P.M., Lloyd, J.W.: The Gödel programming language. The MIT Press (1994).
9. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2006).
10. JSR-331: *JSR-331 Java Constraint Programming API*, `https://jsr331.org/`.
11. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Keijiro, A., Gnesi, S., Mandrioli, D. (eds.) FME. LNCS, v. 2805, 855–874. Springer (2003).
12. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard CP modelling language. In: Bessiere, C. (ed.) CP 2007. LNCS, v. 4741, 529–543. Springer (2007).
13. Rossi, G.: {*log*} (2008), `http://people.dmi.unipr.it/gianfranco.rossi/setlog.Home.html`.
14. Rossi, G., Panegai, E., and Poleo, E.: JSetL: A Java Library for Supporting Declarative Programming in Java. Software-Practice & Experience, 37:115-149 (2006).
15. Schwartz, J. T., Dewar, R. B. K., Dubinsky, E., Schonberg, E.: Programming with Sets: an Introduction to SETL. Springer-Verlag (1986).
16. Shmueli, O., Naqvi, S.: Set Grouping and Layering in Horn Clause Programs. In J-L. Lassez (ed.), Proc. Fourth Int'l Conf. on Logic Programming, 152–177. The MIT Press (1987).