JSetL User's Manual

Version 3.0

Gianfranco Rossi*, Roberto Amadini and Andrea Fois
Dipartimento di Scienze Matematiche, Fisiche e Informatiche
Università di Parma
Parma (Italy)

March 3, 2020

Abstract

This manual describes JSetL 3.0, a Java library that offers a number of facilities to support declarative programming like those usually found in constraint logic programming languages: logical variables, list and set data structures (possibly partially specified), unification, constraint solving over integers and sets, nondeterminism. JSetL is intended to be used as a general-purpose tool, not devoted to any specific application. The manual describes all the features of JSetL and it shows, through simple examples, how to use them.

JSetL has been developed at the Department of Mathematics and Computer Science of the University of Parma (Italy). It is completely written in Java. The full Java code of the JSetL library, along with sample programs and related material, is available at the JSetL WEB page `http://www.clpset.unipr.it/jsetl/`.

*Correspondence to: Gianfranco Rossi, Dipartimento di Scienze Matematiche, Fisiche e Informatiche, Università di Parma, Parco Area delle Scienze, 53/A, I-43124 Parma, Italy. E-mail address: `gianfranco.rossi@unipr.it`

Contents

1 Introduction

JSetL is a Java library that combines the object-oriented programming paradigm of Java with valuable concepts of CLP languages [6], such as logical variables, lists (possibly partially specified), unification, constraint solving, nondeterminism. The library provides also sets and set constraints, like those found in CLP($\mathcal{SET}$) [3] and subsequent extensions [2, 1].

Unification may involve logical variables, as well as list and set objects ("set unification"). Set objects can be defined either *extensionally*, i.e., by enumerating their elements, or *intensionally*, i.e., by providing a property that all elements must satisfy. Set objects can be *partially specified* in that they can contain unbound variables, both as set elements and as part of the set itself.

Constraints concern basic set-theoretical operations (e.g., membership, union, intersection, etc.), as well as equality, inequality and comparison operations over integers. Constraint solving over integers uses the well-known *finite domain* (FD) constraint solving techniques. Constraint solving over sets uses both the efficient finite set (FS) constraint solving techniques of [5], for completely specified sets of integers, and the less efficient but more general and complete constraint solving procedure of CLP($\mathcal{SET}$) [3] for general (possibly partially specified and nested) sets of elements of any type.

Nondeterminism (using choice points and backtracking) is exploited both by specific methods for solution search (namely, the *labeling* methods), as well as by constraints over general sets, e.g., set unification, set union, etc..

Finally, JSetL allows the user to define new constraints and to deal with them as the built-in ones.

How to get JSetL

JSetL has been developed at the Department of Mathematics of the University of Parma (Italy). It is completely written in Java. The full Java code of the JSetL library, along with sample programs and related documents, is available at `http://www.clpset.unipr.it/jsetl/`.

The library is free software; one can redistribute it and/or modify it under the terms of the GNU General Public License.

How to use JSetL

The library is carried out as a Java package, and as such it is subject to the common rules of use defined by the language. The classes of the library must be saved into a folder named `jsetl`. To use JSetL in a program it is necessary to import the library by inserting the statement

```
import jsetl.*;
```

at the beginning of the source file. JSetL must be a sub-folder of the folder in which the classes that import JSetL are saved. Otherwise, the path from root to the library folder must be added to the variable `CLASSPATH`.

Credits

The first implementation of JSetL was carried out by Elisabetta Poleo, during her "Laurea" Thesis in 2002 under the supervision of Gianfranco Rossi. The library code was subsequently fixed by Elio Panegai and Gianfranco Rossi. In 2008 Federico Bergenti joined the JSetL Team until 2014. In the years 2010–2012 Luca Chiarabini worked at the development of programming examples using JSetL. In 2011 Roberto Amadini contributed to the design and

implementation of part of the JSetL constraint solver. At present, Andrea Fois is deeply reviewing the whole implementation of JSetL and developing some new extensions of it.

Since its beginning, several students, under the supervision of Gianfranco Rossi, have helped us to improve the library, adding new functionalities and enhancing the existing ones: (in chronological order) Nadia Toledi, Delia Di Giorgio, Roberto Amadini, Daniele Pandini, Michele Giacomo Filippi, Luca Pedrelli, Alberto Dalla Valle, Lucia Guglielmetti, Fabio Biselli, Andrea Longo, Riccardo Zangrandi, Michele Giacobini, Federica Belli, Lorenzo Furini, Alessio Bertolotti, Davide Allevi, Gianluca Lutero, Lorenzo De Santis, Andrea Guerra, Andrea Fois, Michael Cobianchi, Marco Ghezzi, Francesco Vetere, Giulia Magnani.

2 JSetL class hierarchy

In JSetL, every logical object (e.g., a logical variable, a logical set, etc.) is an instance of the class `LObject`. This class provides general utility methods that are common to every logical object, such as methods to check whether an object is bound or not, to set its external name, and so on.

`LObject` has two subclasses, corresponding to two different kinds of logic objects:

- `LVar`, which represents (general) *logical variables* whose values can be objects of any type. `LVar` provides three subclasses that may be used by the user to post type-specific constraints:
  - `BoolLVar`, representing *boolean logical variables*, i.e., logical variables whose values are either `true` or `false`; this class provides methods for posting constraints over boolean variables, such as logical conjunction, disjunction, implication, negation, etc.
  - `IntLVar`, representing *integer logical variables*, i.e., logic variables whose values are integers; this class provides additional methods for posting constraints over integers, such as multiplication, subtraction, module, etc.
  - `SetLVar`, representing *set logical variables*, i.e., logic variables whose values are bounded sets of integers; this class provides additional methods for posting constraints over sets, such as equality, union, intersection, etc.
- `LCollection`, which represents (general) unbounded collections of values of any kind (even non-homogenous in type). The user never creates instances of `LCollection`; instead he/she can use one of its subclasses.
  - `LList`, representing *logical lists*, i.e., logical collections whose values are lists of values of any kind, including other (possibly unbound) logical objects (in particular, other logical lists). `LList` provides also a subclass:
    - `LPair`, representing *logical pairs*, i.e., logical objects whose values are pairs of values of any kind, including other (possibly unbound) logical objects.
  - `LSet`, representing *logical sets*, i.e., logical collections whose values are sets of values of any kind, including other (possibly unbound) logical objects (in particular, other logical sets). This class provides methods for posting constraints over set variables, such as equality, membership, union, intersection, etc. `LSet` provides also a number of subclasses that allow the user to specify constraints on specific kinds of sets (namely, sets of integer numbers, intensional sets, binary relations, maps).

    - `IntLSet`, representing *integer logical sets*, i.e., logic sets whose elements are either integer values or (possibly unbound) integer logical variables (i.e., instances of `IntLVar`).
    - `LRel`, representing *binary relations*, i.e., logical sets whose elements are (logical) pairs of elements of any kind; this class provides additional methods for posting constraints over binary relations, such as domain, range, composition, etc. `LRel` provides also a subclass:
        - `LMap`, representing *maps* (or *partial functions*), i.e., binary relations where all pair first components are all distinct from each other.
    - `Ris`, representing *Restricted Intensional Sets*, i.e., logical sets whose elements (of any kind) are specified by providing a property that they must satisfy in order to belong to the set.

All exceptions defined in the library are located in the package `jsetl.exception`. Those exceptions are listed below.

- `Fail`: unchecked exception raised when a constraint is found to be unsatisfiable, to start the backtracking procedure.
- `Failure`: checked exception raised when a constraint is found to be unsatisfiable and there are no more open choice points, meaning that no solution can be found.
- `InitLObjectException`: unchecked exception raised when a logical object should be unbound but it is found to be bound to some value.
- `NotDefConstraintException`: unchecked exception raised when the solver tries to solve a not defined constraint (neither built-in nor user-defined).
- `NotInitLObjectException`: unchecked exception raised when a variable that should be bound to some value is found to be unbound.
- `NotPFunException`: unchecked exception raised when the created `LMap` is not a partial function.
- `NotValidDomainException`: unchecked exception raised when a not valid `IntLVar` or `SetLVar` domain is found.
- `UnsupportedHeuristicException`: unchecked exception raised when a labeling heuristic is not supported by the method that is asked to use it.

All parameters of all methods (and constraints) in the library should not be `null`, unless they are overriding `Object` methods that allow `null` values. If a parameter is `null`, an exception `NullPointerException` is raised; if a collection or array of values is passed as parameter, none of the values can be `null`, otherwise an exception `NullPointerException` is raised. For more info on nullable and non-null parameters and return values see the Javadoc documentation.

3 Logical variables: the class `LVar`

Logical variables represent *unknowns*. 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 *constraints*, involving logical variables and values from some specific domains.

When the domain of a variable is restricted to a single value, we say that the variable is *bound* to (or *instantiated* with) this value. Otherwise, the variable is *unbound*. With a little abuse of terminology, we say that the value associated with a bound variable x is *the value of* x.

The equality constraint, 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.

A logical variable also has an *external name*. The external name is a string value which can be useful when printing the variable and the possible constraints involving it.

In JSetL, a logical variable is an instance of the class `LVar`. This class provides constructors for creating logical variables and a number of simple methods for testing and manipulating logical variables, as well as basic constraints over logical variables (namely, equality and inequality, membership and not membership).

3.1 Constructors

`LVar()`
`LVar(String extName)`
: Construct an unbound logical variable, with default external name (resp., with external name `extName`).

`LVar(Object o)`
`LVar(String extName, Object o)`
: Construct a (bound) logical variable, with default external name (resp., with external name `extName`), and value `o`. `o` can be of any type, but a `String` for the one parameter constructor (in fact, in such case, the `LVar(String extName)` constructor is called). Same as to create an unbound logical variable `x` and to post and solve the constraint `x.eq(o)`.

`LVar(LVar lv)`
`LVar(String extName, LVar lv)`
: Construct a logical variable, with default external name (resp., with external name `extName`), equivalent to the logical variable `lv`. Same as to create an unbound logical variable `x` and to post and solve the constraint `x.eq(lv)`.

Remarks

- If `extName` is omitted, a default external name `"N`n`"` is assigned to the variable automatically in which n is a progressive generated number.
- Two logical variables x and y are *equivalent* if they have been (successfully) unified, e.g., by x.`eq`(y). Equivalent variables are dealt with as they were the same variable. If x and y are unbound, and x is unified to y, both x and y remain unbound; if later on, x is bound to some value o, then y becomes bound to the same value o.

Example 1

- *Create an unbound logical variable* `x` *(with default external name):*

```
LVar x = new LVar();
```

- *Create an unbound logical variable* `y` *with external name* `"y"`*:*

```
LVar y = new LVar("y");
```

- *Create a logical variable* `z`*, with external name* `"z"`*, bound to the integer value* `2`*:*

```
LVar z = new LVar("z", 2);
```

- *Create a logical variable* `v` *equivalent to the logical variable* `x`*:*

```
LVar v = new LVar(x);
```

3.2 General utility methods

The class `LVar` provides a number of utility methods that allow the user to inspect a logical variable (e.g., to test if it is bound), to get and set its external name, to print its value, and so on.

`LVar` **clone**`()`
: Returns a copy of this logical variable.

`boolean` **equals**`(Object o)`
: Returns `true` if this logical variable is equal to the object `o`. If `o` is a logical variable, `x` and `o` are equal if either they are the same object (either bound or unbound), or they are equivalent logical variables (either bound or unbound), or they are distinct logical variables but bound to equal values. If `o` is not a logical variable, `x` and `o` are equal if `x` is bound to a value equal to `o`.

`String` **getName**`()`
: Returns the external name associated with this logical variable.

`Object` **getValue**`()`
: Returns the value of this logical variable if it is bound, `null` otherwise.

`boolean` **isBound**`()`
: Returns `true` if this logical variable is bound, `false` otherwise.

`boolean` **isGround**`()`
: Returns `true` if this logical variable is ground, `false` otherwise. A logical variable is ground if it is bound to a non-logical object (regardless of whether this object contains logical objects or not) or if it is bound to a ground logical object (in particular, a ground logical list or set).

`void` **output**`()`
: Prints `'_'` followed by the external name of this logical variable, followed by `'='`, followed by the value of this logical variable if it is bound, or by `"unknown"` if it is unbound (e.g., `_x = 3` or `_y = unknown`).

`LVar` **setName**`(String extName)`
: Sets the external name of this logical variable to `extName` and returns this logical variable.

`LVar` **setValue**`(Object o)`
: If this logical variable is unbound, binds it to the value `o` (same as to post and solve the constraint `this.eq(o)`) and returns this logical variable. If this logical variable is already bound, raises an exception `InitLObjectException`.

`String` **toString**`()`

Returns the string corresponding to the value of this logical variable if it is bound; otherwise, returns the string `"_extName"`, where `extName` is the external name of this logical variable.

Example 2 *(refer to declarations of Example 1)*

- *Test* `equals`*:*

```
x.equals(x);      // -->  true
z.equals(2);      // -->  true
x.equals(y);      // -->  false
x.equals(v);      // -->  true
```

- *Output bound/unbound logical variables:*

```
x.output();       // -->  _N1 = unknown
y.output();       // -->  _y = unknown
z.output();       // -->  _z = 2
```

- *Convert to string:*

```
x.toString();       // -->  _N1
y.toString();       // -->  _y
z.toString();       // -->  2
```

3.3 Constraint methods

Constraints (see Sect. 14) can be posted over logical variables. In particular, the class `LVar` provides methods for generating equality, inequality, membership and not membership constraints.

`Constraint` **eq**`(Object o)`

Returns the atomic constraint `this` = `o`, that *unifies* this logical variable with the object `o`. In particular, if `o` is an object other than a logical variable, and this variable is unbound, this variable becomes bound to `o` after the constraint has been solved.

`Constraint` **neq**`(Object o)`

Returns the atomic constraint `this` $\neq$ `o`, that requires this logical variable to be different from the object `o`.

`Constraint` **in**`(LSet ls)`
`Constraint` **in**`(Set<?> s)`

Return the atomic constraint `this` $\in$ `ls` (resp., `this` $\in$ `s`), that requires this logical variable to be a member of the logical set `ls` (resp., of the Java generic set `s`). When solved, this constraint will nondeterministically unify this logical variable with each value in `ls` (resp., in `s`). The constraint will succeed if at least one unification succeeds.

`Constraint` **nin**`(LSet ls)`
`Constraint` **nin**`(Set<?> s)`

Return the atomic constraint `this` $\notin$ `ls` (resp., `this` $\notin$ `s`), that requires this logical variable not to be a member of the logical set `ls` (resp., of of the Java generic set `s`).

4 Logical lists: the class `LList`

A *logical list* (or, simply, a *l-list*) l is a special kind of logical object whose value is a pair $\langle elems,\ tail \rangle$, where *elems* (the *list value*) is a list of n objects of arbitrary types ($n \geq 0$), and *tail* is either an empty or an unbound l-list, representing the rest of l. When *elems* is empty (i.e., $n = 0$), l is the *empty l-list* and we use [ ] to denote it. When *elems* is not empty (i.e., $n > 0$): if *tail* is the empty l-list, we say that l represents a *closed* list and we use the (abstract) notation $[e_0, \ldots, e_n]$; if *tail* is an unbound l-list r, we say that l represents an *open* list and we use the (abstract) notation $[e_0, \ldots, e_n \mid r]$ to denote it.

A l-list that contains unbound logical objects (i.e., either variables or lists or sets) represents a *partially specified list*. Unbound logical objects can occur in a l-list l both as elements and as the tail of the l-list itself (i.e., l is an open l-list). In the last case, l represents an unbounded collection of (possibly unknown) values.

In JSetL, a logical list is defined as an instance of the class `LList`, which extends the class `LCollection`. In particular, the list value (i.e., the *elems* part of a l-list) is an instance of the class `ArrayList` which implements the `java.util.List` interface.

The class `LList` provides methods to create new l-lists, possibly starting from existing ones, to deal with l-lists as logical objects and to deal with values possibly bound to l-lists. Moreover, like logical variables, l-lists can be used to post constraints that implement basic list operations.

4.1 Constructors

`LList()`
`LList(String extName)`
: Construct an unbound l-list, with default external name (resp., with external name `extName`).

`LList(List<?> l)`
`LList(String extName, List<?> l)`
: Construct a (bound) l-list, with default external name (resp., with external name `extName`), and value `l`. Same as to create an unbound l-list `x` and to post and solve the constraint `x.eq(l)`.

`LList(LList ll)`
`LList(String extName, LList ll)`
: Construct a l-list, with default external name (resp., with external name `extName`), equivalent to the l-list `ll`. Same as to create an unbound l-list `x` and to post and solve the constraint `x.eq(ll)`.

Example 3

- *Create an unbound l-list* `a`*:*

```
LList a = new LList();
```

- *Create a l-list* `b`*, with external name* `"b"`*, bound to the list value* `[1,2,3]`*:*

```
List l = new ArrayList();
l.add(1); l.add(2); l.add(3);
LList b = new LList("b", l);
```

- *Create a l-list* `c` *equivalent to the l-list* `a`*:*

```
LList c = new LList(a);
```

4.2 Creating new (bound) logical lists

`static LList` **`empty`**`()`
Returns the empty l-list.

`LList` **`ins`**`(Object o)`
If this l-list is bound, returns the (new) l-list whose value is obtained by adding `o` as the *first* element of this l-list value. Otherwise (i.e., this l-list is unbound), it returns the (new) l-list whose value is the list containing `o` as its first element and this l-list as the *tail* part (i.e., `[o | this]`).

`LList` **`insn`**`(Object o)`
Like **ins**, but `o` is added as the *last* element of this l-list value.

`LList` **`insAll`**`(Object[] c)`
`LList` **`insAll`**`(Collection c)`
If this l-list is bound, return the (new) l-list whose value is obtained by adding all elements of `c` in front of this l-list value, respecting the order they have in `c`. Otherwise, returns the (new) l-list whose value is the list containing all elements of `c` and this l-list as its *tail* part (i.e., `[`$\mathtt{a}_1$`,...,`$\mathtt{a}_n$ `| this]`, where $\mathtt{a}_1,\ldots,\mathtt{a}_n$ are the elements of `c`).

`static LList` **`mkList`**`(int n)`
`static LList` **`mkIntList`**`(int n)`
Return a l-list, with default external name, whose value is a list of `n` unbound logical variables (resp., integer logical variables—see Sect. 11).

Remarks

- The `ins`, `insn`, and `insAll` methods do not modify the object on which they are invoked: rather they build and return a new l-list obtained by adding the elements to the given list. Hence, calls to list insertion methods can be concatenated (left associative).
- Elements to be added to the list value through the `ins`, `insn`, and `insAll` methods can be of any type, including bound or unbound logical variables, lists or sets.
- The l-list on which `ins`, `insn`, and `insAll` methods are invoked can be either bound or unbound. In particular, invoking the `ins`, `insn`, and `insAll` methods on an unbound l-list allows an *open* list to be built.
- The order of elements and repetitions do matter in lists (whereas they do not matter in sets—see Sect. 6). Thus, for instance, the list $[1, 2]$ is different from the list $[2, 1]$ and from the list $[1, 2, 2]$.

Example 4

- *Create a l-list* `d`*, bound to the empty list:*

```
LList d = LList.empty();
```

- *Create a l-list* `e`*, bound to the list value* `['c','b','a']`*:*

```
LList e = LList.empty().ins('a').ins('b').ins('c');
```

or

```
Character[] elems = {'a','b','c'};
LList e = LList.empty().insAll(elems);
```

- *Create a partially specified l-list* `f`*, bound to the list value* `[1,x]`*, where* `x` *is an unbound logical variable:*

```
LVar x = new LVar("x");
LList f = LList.empty().insn(1).insn(x);
```

- *Create a l-list* `g`*, representing the open list* `[1,2|r]`*, where* `r` *is an unbound l-list:*

```
LVar z = new LVar(2);
LList r = new LList("r");
LList g = r.insn(1).insn(z);
```

  Note that the list value bound to `g` *is the list* `[1,2]`*, since* `z` *is replaced by its value* `2`*.*

- *Create a l-list* `h`*, bound to the list value* `[[],['c','b','a'],[1,x]]` *(i.e., a list of nested lists):*

```
LList ef = LList.empty().ins(f).ins(e).ins(d);
```

  where `d`*,* `e`*, and* `f` *are the l-lists defined above.*

- *Create a l-list* `i`*, bound to the list of three new unbound logical variables* `[X_1,X_2,X_3]`*:*

```
LList i = LList.mkList(3);
```

4.3 General utility methods

The class `LList` provides the utility methods of the class `LVar` (see Section 3.2), along with a few other methods that take into account the fact that values possibly bound to `LList` objects are collections (namely, lists), possibly containing other logical variables and collections.

4.3.1 Basic methods

The following methods are the same as `LVar`'s methods, but adapted to l-list objects:

- `LList` **clone**`()`
- `boolean` **equals**`(Object o)`
- `String` **getName**`()`
- `ArrayList<?>` **getValue**`()`
- `boolean` **isBound**`()`
- `void` **output**`()`
- `LList` **setName**`(String extName)`
- `LList` **setValue**`(List<?> l)`
- `String` **toString**`()`

Remarks

- `getValue` consider only the list value of the given l-list (i.e., it ignores its *tail* part). In particular, `getValue` returns the *elems* list of the given l-list. In order to obtain also its *tail* part one must use the method `getTail` (see 4.3.2).
- If the l-list denotes an open list, the string returned by the method `toString()` is `"[e_1,...,e_n | _r]"`, where `"e_1",...,"e_n"` are the strings obtained from the list elements and `"r"` is the external name of the unbound l-list denoting the *tail* part of the list. Similarly, the output produced by the method `output()` is `_l = [e_1,...,e_n | _r]` where `l` is the external name of the list.

4.3.2 Logical collection methods

The following methods take into account the fact that the value of a bound logical list is a collection, possibly partially specified (i.e., containing unknown elements), and that a logical list may have a *tail* part. Methods in the first group are specific of the class `LList`, while methods in the second group are common to all collections, in particular to `java.util.List` objects.

Observe that most of these methods take into account only the *elems* part (i.e., the *list value*) of the invocation l-list, while the *tail* part is simply ignored.

`LList` **getTail**`()`
: If this l-list is bound to a not empty l-list, returns its *tail* part, that is either an unbound l-list if this l-list is open, or the empty list if this l-list is closed. Otherwise (i.e., this l-list is unbound or it is the empty l-list), returns a l-list equivalent to the l-list itself.

`boolean` **isClosed**`()`
: If this l-list is bound and it has an empty *tail* part, returns `true`; otherwise (i.e., if this l-list is bound and it has an unbound *tail* part, or this l-list is unbound), returns `false`.

`boolean` **isGround**`()`
: If this l-list is bound and all its elements and the *tail* part are ground, returns `true`; otherwise (i.e., if the list value of this l-list is not ground, or its *tail* part is not ground, or if this l-list is unbound), returns `false`.

`void` **printElems**`(char sep)`
: If this l-list is bound, prints all elements of its list value separated by the character `sep`; otherwise, prints `"_extName"`, where `extName` is the external name of this l-list.

`Object[]` **toArray**`()`
: If this l-list is bound, returns an array containing all elements in its list value, from first to last element; otherwise, returns an empty array.

The following methods provides facilities that are also provided by the interface `List` of `java.util`. They could always be replaced by an invocation to the equivalent method of `List` applied to the list value that is returned by calling `getValue` on the l-list. However, they are provided by `LList` for the user convenience.

`Object` **get**`(int i)`
: If this l-list is bound, returns the `i`-th element of its list value; otherwise, raises an exception `NotInitLObjectException`. When the l-list `v` is bound, `v.get(i)` is equivalent to `v.getValue().get(i)`.

`int` **getSize**`()`
: If this l-list is bound, returns the number of elements in its list value. Otherwise, it raises an exception `NotInitLObjectException`. When the l-list `v` is bound, `v.getSize()` is equivalent to `v.getValue().size()`.

`boolean` **isEmpty**`()`
: If this l-list is bound to the empty l-list, returns `true`. Otherwise:if this l-list is bound to a not empty l-list, returns `false`; if this l-list is unbound, raises an exception `NotInitLObjectException`. When the l-list `v` is bound, `v.isEmpty()` is equivalent to `v.getValue().isEmpty()`.

`Iterator` **iterator**`()`
: Returns an `Iterator` over the elements of its list value. When the l-list `v` is bound, `v.iterator()` is equivalent to `v.getValue().iterator()`.

`boolean` **testContains**`(Object o)`
: If this l-list is bound and its list value contains `o`, returns `true`. Otherwise: if this l-list is bound and its list value does not contain `o`, returns `false`; if this l-list is unbound, raises an exception `NotInitLObjectException`. When the l-list `v` is bound, `v.testContains(o)` is equivalent to either `v.getValue().contains(o.getValue())` or `v.getValue().contains(o)`, depending on whether `o` is a bound logical object (i.e., either a logical variable or a logical collection) or not.
Note that when `o` is a logical object, the value possibly associated with `o` is considered for the membership test; similarly, when elements of the list value associated with this l-list are logical objects, then their values are taken into account, if any.

4.4 Constraint methods

The class `LList` provides methods for generating equality and inequality constraints over lists.

`Constraint` **eq**`(LList ll)`
: Returns the atomic constraint `this` = `ll`, that *unifies* this l-list with the l-list `ll`. If this l-list is unbound and `ll` is bound to a list `l`, this l-list becomes bound to `l` after the constraint has been solved; conversely, if `ll` is unbound, this l-list remains unbound. Note that, solving this constraint causes the unbound variables possibly occurring in `ll` to be bound to the proper values, as required by the unification of the two lists.

`Constraint` **eq**`(List<?> l)`
: Returns the atomic constraint `this` = `l`, that unifies this l-list with the generic Java list `l`. If this l-list is unbound, it becomes bound to `l` after the constraint has been solved.

`Constraint` **neq**`(LList ll)`
`Constraint` **neq**`(List<?> l)`
: Return the atomic constraint `this` $\neq$ `ll`, that requires this l-list to be different from the l-list `ll` (resp., from the generic Java list `l`).

`Constraint` **in**`(LSet ls)`
`Constraint` **in**`(Set<?> s)`
: Return the atomic constraint `this` $\in$ `ls` (resp., `this` $\in$ `s`), that requires this logical list to be a member of the logical set `ls` (resp., of the Java generic set `s`). When solved,

this constraint will nondeterministically unify this logical list with each value in `ls` (resp., in `s`). The constraint will succeed if at least one unification succeeds.

`Constraint` **`nin`**`(LSet ls)`
`Constraint` **`nin`**`(Set<?> s)`
Return the atomic constraint `this` $\notin$ `ls` (resp., `this` $\notin$ `s`), that requires this logical list not to be a member of the logical set `ls` (resp., of of the Java generic set `s`).

The class `LList` provides also a method for generating a specified constraint for all elements of a list:

`Constraint` **`forallElems`**`(LVar y, Constraint c)`
If this l-list is bound, returns a conjunction of atomic constraints $c_1 \wedge c_2 \wedge \cdots \wedge c_n$ for all elements $e_i, \ldots, e_n$ of the list bound to this l-list; otherwise (i.e., this l-list is unbound), it raises an exception `NotInitLObjectException`. Each c_i is obtained from `c` by replacing all occurrences of `y` with e_i. For example, if `l` is the list `[1,2,z]`, where `z` is an unbound logical variable, `l.forallElems(y,y.neq(0))` generates a conjunction of constraints logically equivalent to $1 \neq 0 \wedge 2 \neq 0 \wedge \mathtt{z} \neq 0$.

5 Logical pairs: the class `LPair`

A *logical pair* (or, simply, a *l-pair*) is an ordered sequence of two elements X and Y, represented as (X, Y), where X and Y can be ground or non-ground logical objects.

In `JSetL` a logical pair is an instance of the class `LPair`, which extends `LList`. Since logical pairs are logical lists, they support every method and constraint defined for their direct superclass, along with some more methods that take into account the fact that logical pairs may have just two elements.

5.1 Constructors

`LPair()`
`LPair(String extName)`
Construct an unbound logical pair with default external name (resp. with external name `extName`).

`LPair(Object o1, Object o2)`
`LPair(String extName, Object o1, Object o2)`
Construct a logical pair [o1,o2] with default external name (resp. with external name `extName`).

`LPair(LPair p)`
`LPair(String extName, LPair s)`
Construct a logical pair, with default external name (resp., with external name `extName`), equivalent to the logical pair `p`. Same as to create an unbound logical pair `x` and to post and solve the constraint `x.eq(p)`.

Example 5

Create a new unbound logical pair with external name `"p"`*:*

```
LPair p1 = new LPair("p");
```

Create a new bound logical pair with external name `"pp"` *and value* (`y`,`z`)*:*

```
LVar y = new LVar("y");
LVar z = new LVar("z");
LPair p3 = new LPair(y,z);
```

5.2 General utility methods

`LPair` inherits all `LList` methods and redefines some of them. The following methods are adapted for `LPair`:

`LPair` **clone**`()`

`Object` **getFirst**`()`

`Object` **getSecond**`()`

`LPair` **setName**`(String name)`

`LPair` **setValue**`(List<?> l)` This method throws an exception `IllegalArgumentException` if `l` is not a pair.

5.3 Constraint methods

`LPair` inherits all constraint methods from `LList`.

6 Logical sets: the class `LSet`

A *logical set* (or, simply, a *l-set*) s is a special kind of logical object whose value is a pair $\langle elems,\ tail \rangle$, where *elems* (the *set value*) is a set of n objects of arbitrary types ($n \geq 0$), and *tail* is either an empty or an unbound l-set representing the rest of s. When *elems* is empty (i.e., $n = 0$), s is the *empty l-set* and we use $\{\}$ to denote it. When *elems* is not empty (i.e., $n > 0$): if *tail* is the empty l-set, we say that s represents a *closed* set and we use the (abstract) notation $\{e_0, \ldots, e_n\}$; if *tail* is an unbound l-set r, we say that s represents an *open* set and we use the (abstract) notation $\{e_0, \ldots, e_n \,/\, r\}$ to denote it.

Logical sets are similar to logic lists in many aspects. In particular, like l-lists, l-sets can represent *partially specified* collections. The main difference with l-lists is that the order of elements and repetitions in a l-set do not matter, while they are important in l-lists.

Note that, differently from l-lists, the cardinality of a partially specified set is not determined uniquely (even if the set is closed). For example, the cardinality of the set `{1,x}`, where `x` is an unbound logical variable, can be 1 or 2 depending on whether `x` will be subsequently bound to a value equal to 1 or different from 1, respectively.

In JSetL, a logical set is defined as an instance of the class `LSet`, which extends the class `LCollection`. In particular, the set value (i.e., the *elems* part of a l-set) is an instance of the class `HashSet` which implements the `java.util.Set` interface.

Methods provided by the class `LSet` are basically the same as those of the class `LList` but applied to `LSet` objects. In particular, `LSet` provides methods to create new l-sets, possibly starting from existing ones, to deal with l-sets as logical objects and to deal with values possibly bound to l-sets. Moreover, l-sets can be used to post constraints that implement most of the usual set-theoretical operations.

6.1 Constructors

`LSet()`
`LSet(String extName)`
Construct an unbound l-set, with default external name (resp., with external name `extName`).

`LSet(Set<?> s)`
`LSet(String extName, Set<?> s)`
Construct a (bound) l-set, with default external name (resp., with external name `extName`), and value `s`. Same as to create an unbound l-set `x` and to post and solve the constraint `x.eq(s)`.

`LSet(LSet ls)`
`LSet(String extName, LSet ls)`
Construct a l-set, with default external name (resp., with external name `extName`), equivalent to the l-set `ls`. Same as to create an unbound l-set `x` and to post and solve the constraint `x.eq(ls)`.

Example 6

- *Create an unbound l-set* `a`*:*

```
LSet a = new LSet();
```

- *Create a l-set* `b`*, with external name* `"b"`*, bound to the set value* {`1,2,3`}*:*

```
Set s = new HashSet();
s.add(1); s.add(2); s.add(3);
LSet b = new LSet("b",s);
```

- *Create a l-set* `c` *equivalent to the l-set* `a`*:*

```
LSet c = new LSet(a);
```

6.2 Creating new (bound) logical sets

`static LSet` **`empty`**`()`
Returns the empty l-set.

`LSet` **`ins`**`(Object... o)`
If this l-set is bound, return the (new) l-set whose value is obtained by adding all elements of `o` as elements of the set bound to this l-set. Otherwise, return the (new) l-set whose value is the set containing all elements in `o` and this l-set as its *tail* part (i.e., {$\mathtt{a}_1$`,...,`$\mathtt{a}_n$ `/ this`}, where $\mathtt{a}_1$`,...,`$\mathtt{a}_n$ are the elements of `o`).

`LSet` **`insAll`**`(Object[] c)`
`LSet` **`insAll`**`(Collection c)`
If this l-set is bound, return the (new) l-set whose value is obtained by adding all elements of `c` as elements of the set bound to this l-set. Otherwise, return the (new) l-set whose value is the set containing all elements in `c` and this l-set as its *tail* part (i.e., {$\mathtt{a}_1$`,...,`$\mathtt{a}_n$ `/ this`}, where $\mathtt{a}_1$`,...,`$\mathtt{a}_n$ are the elements of `c`).

`static LSet` **`mkSet`**`(int n)`
If `n` ≥ 0, returns a l-set (with default external name), whose value is a set of `n` unbound logical variables; if `n` < 0, raises an exception `IllegalArgumentException`.

Note that since the order of elements in a set is not important, it is not necessary to supply distinct methods for head and tail insertion because they would produce the same set.

All remarks and examples shown for l-lists in Section 4 are still valid in the case of l-sets, provided `LList` is replaced by `LSet` and `insn` is replaced by `ins`.

6.3 General utility methods

The class `LSet` provides most of the utility methods of classes `LVar` (see Section 3.2) and `LList` (see Section 4.3).

6.3.1 Basic methods

The following methods are the same as `LVar`'s methods, but adapted to `LSet` objects:

- `LSet` **clone**`()`
- `boolean` **equals**`(Object o)`
- `String` **getName**`()`
- `HashSet<?>` **getValue**`()`
- `boolean` **isBound**`()`
- `void` **output**`()`
- `LSet` **setName**`(String extName)`
- `LSet` **setValue**`(Set<?> s)`
- `String` **toString**`()`

Remarks of Section 4.3.1 apply to l-sets as well.

6.3.2 Logical collection methods

The class `LSet` provides all the logical collection methods of the class `LList`, except the method `get` (see Section 4.3.2). Note that, like in the case of l-lists, most of the collection methods take into account only the *elems* part (i.e., the *set value*) of the invocation l-set, while the *tail* part is simply ignored.

- `int` **getSize**`()`
- `LSet` **getTail**`()`
- `boolean` **isClosed**`()`
- `boolean` **isEmpty**`()`
- `boolean` **isGround**`()`
- `Iterator` **iterator**`()`
- `void` **printElems**`(char sep)`
- `boolean` **testContains**`(Object o)`
- `Object[]` **toArray**`()`

Remarks

- `LSet` objects may contain multiple occurrences of the same value. For example, if `s` is the set {`x`,`2`}, where `x` is a logical variable, and `x` is successively bound to `2`, then `s` will contain two occurrences of the same value `2`. However, all methods dealing with l-sets, but the method `toArray`, ignore repetitions (i.e., they consider multiple occurrences of the same value as a single set element). For example, calling `getSize` on the set `s` considered above, we get `1` as its result. Conversely, `s.toArray()` returns the array `[2,2]`.

To explicitly remove duplicates from l-sets, the class `LSet` provides also the following method:

`LSet` **normalizeSet**`()`
Returns a copy of this l-set where all duplicates are physically removed.

6.4 Constraint methods

The class `LSet` provides methods for generating constraints over sets (referred to as *l-set constraints*) that implement most of the usual set-theoretical operations.

Comparison constraints

`Constraint` **eq**`(LSet ls)`
Returns the atomic constraint `this` = `ls`, that *unifies* this l-set with the l-set `ls`. If this l-set is unbound and `ls` is bound to a set `s`, this l-set becomes bound to `s` after the constraint has been solved; conversely, if `ls` is unbound, this l-set remains unbound. Note that solving this constraint causes the unbound variables possibly occurring in `ls` to be bound to the proper values as required by *set unification* (see, e.g., [4]).

`Constraint` **eq**`(Set<?> s)`
Returns the atomic constraint `this` = `s`, that unifies this l-set with the set `s`. If this l-set is unbound, it becomes bound to `s` after the constraint has been solved.

`Constraint` **neq**`(LSet ls)`
`Constraint` **neq**`(Set<?> s)`
Return the atomic constraint `this` $\neq$ `s`, that requires this l-set to be different from the l-set `ls` (resp., from the set `s`).

Membership constraints

`Constraint` **in**`(LSet ls)`
`Constraint` **in**`(Set<?> s)`
Return the atomic constraint `this` $\in$ `ls` (resp., `this` $\in$ `s`), that requires this logical set to be a member of the logical set `ls` (resp., of the Java generic set `s`). When solved, this constraint will nondeterministically unify this logical set with each value in `ls` (resp., in `s`). The constraint will succeed if at least one unification succeeds.

`Constraint` **nin**`(LSet ls)`
`Constraint` **nin**`(Set<?> s)`
Return the atomic constraint `this` $\notin$ `ls` (resp., `this` $\notin$ `s`), that requires this logical set not to be a member of the logical set `ls` (resp., of of the Java generic set `s`).

`Constraint` **contains**`(Object o)`
Returns the atomic constraint `o` ∈ `this` that requires this l-set to contain the object `o`. Same as `new LVar(o).in(this)` (see Sect. 3.3).

`Constraint` **ncontains**`(Object o)`
Returns the atomic constraint `o` ∉ `this` that requires this l-set to not contain the object `o`. Same as `new LVar(o).nin(this)` (see Sect. 3.3).

Set-theoretical constraints

`Constraint` **diff**`(LSet s, LSet q)`
`Constraint` **diff**`(LSet s, Set<?> q)`
`Constraint` **diff**`(Set<?> s, LSet q)`
`Constraint` **diff**`(Set<?> s, Set<?> q)`
Return the constraint *diff*$(this, s, q)$, which is true if and only if q is the set-theoretical *difference* of *this* and s, i.e., $q = this \setminus s$.

`Constraint` **disj**`(LSet s)`
`Constraint` **disj**`(Set<?> s)`
Return the constraint *disj*$(this, s)$, which is true if and only if *this* is disjoint from s, i.e., $this \cap s = \emptyset$.

`Constraint` **inters**`(LSet s, LSet q)`
`Constraint` **inters**`(LSet s, Set<?> q)`
`Constraint` **inters**`(Set<?> s, LSet q)`
`Constraint` **inters**`(Set<?> s, Set<?> q)`
Return the constraint *inters*$(this, s, q)$, which is true if and only if q is the *intersection* between *this* and s, i.e., $q = this \cap s$.

`Constraint` **less**`(LVar o, LSet s)`
`Constraint` **less**`(Object o, LSet s)`
`Constraint` **less**`(LVar o, Set<?> s)`
`Constraint` **less**`(Object o, Set<?> s)`
Return the constraint *less*$(this, o, s)$, which is true if and only if o belongs to *this* and s is equal to *this* less o, i.e., $o \in this \wedge s = this \setminus \{o\}$.

`Constraint` **size**`(IntLVar n)`
`Constraint` **size**`(Integer n)`
Return the constraint *size*$(this, n)$, which is true if and only if n is the *cardinality* of *this*, i.e., $n = |\,$`this`$\,|$.

`Constraint` **subset**`(LSet s)`
`Constraint` **subset**`(Set<?> s)`
Return the constraint *subset*$(this, s)$, which is true if and only if *this* is a subset of s, i.e., $this \subseteq s$.

`Constraint` **union**`(LSet s, LSet q)`
`Constraint` **union**`(LSet s, Set<?> q)`
`Constraint` **union**`(Set<?> s, LSet q)`
`Constraint` **union**`(Set<?> s, Set<?> q)`
Return the constraint *union*$(this, s, q)$, which is true if and only if q is the *union* of *this* and s, i.e., $q = this \cup s$.

Negated versions of the constraints **diff**, **disj**, **inters**, **subset**, **union** are also provided by the following methods.

`Constraint` **ndiff**`(LSet s, LSet q)`
`Constraint` **ndiff**`(LSet s, Set<?> q)`
`Constraint` **ndiff**`(Set<?> s, LSet q)`
`Constraint` **ndiff**`(Set<?> s1, Set<?> q)`

`Constraint` **ndisj**`(LSet s)`
`Constraint` **ndisj**`(Set<?> s)`

`Constraint` **ninters**`(LSet s, LSet q)`
`Constraint` **ninters**`(LSet s, Set<?> q)`
`Constraint` **ninters**`(Set<?> s, LSet q)`
`Constraint` **ninters**`(Set<?> s1, Set<?> q)`

`Constraint` **nsubset**`(LSet s)`
`Constraint` **nsubset**`(Set<?> s)`

`Constraint` **nunion**`(LSet s, LSet q)`
`Constraint` **nunion**`(LSet s, Set<?> q)`
`Constraint` **nunion**`(Set<?> s, LSet q)`
`Constraint` **nunion**`(Set<?> s1, Set<?> q)`

Global constraints

`Constraint` **forallElems**`(LVar y, Constraint c)`
Same as `forallElems` of class `LList`.

7 Integer logical sets: the class `IntLSet`

Integer logical sets are logical sets whose elements are either integer values or (possibly unbound) integer logical variables (i.e., instances of `IntLVar`). Integer logical sets are instances of the class `IntLSet`, which is a subclass of `LSet`. Hence, integer logical sets are by any means logical sets as `LSet` instances are. `IntLSet` objects are usually used in conjunction with `IntLVar` objects to post set-theoretical constraints on partially specified sets of integers.

7.1 Constructors

`IntLSet()`
`IntLSet(String extName)`
Construct an unbound integer l-set, with default external name (resp., with external name `extName`).

`IntLSet(int p, int q)`
`IntLSet(String extName, int p, int q)`
Construct a bound integer l-set, with default external name (resp., with external name `extName`), whose elements are all the integers from `p` to `q`.

`IntLSet(MultiInterval mi)`
`IntLSet(String extName, MultiInterval mi)`
Construct a bound integer l-set, with default external name (resp., with external name `extName`), whose elements are all the integers contained in the multi-interval `mi`, that is the union of n ($n \geq 0$) disjoint *intervals* (see Sections A.1 and A.2 for a precise description of intervals and multi-intervals in JSetL).

`IntLSet(Set<Integer> s)`
`IntLSet(String extName, Set<Integer> s)`
Construct a bound integer l-set, with default external name (resp., with external name `extName`), and value `s`.

`IntLSet(IntLSet s)`
`IntLSet(String extName, IntLSet s)`
Construct an integer l-set `s`, with default external name (resp., with external name `extName`), equivalent to the integer logical set `s`. Same as to create an unbound integer logical set `x` and to post and solve the constraint `x.eq(s)`.

Example 7

- *Create an unbound logical set of integers with name* `"a"`*:*
  `IntLSet a = new IntLSet("a");`
- *Create the logical set of integers representing the interval* $[-2..4]$*:*
  `IntLSet b = new IntLSet(-2, 4);`
- *Create a logical set of integers containing 1,2, -10 and 8:*
  ```
  HashSet<Integer> elems = new HashSet<>();
  elems.add(1); elems.add(2); elems.add(-10); elems.add(8);
  IntLSet c = new IntLSet(elems);
  ```

7.2 General utility methods

Class `IntLSet` inherits all member methods of `LSet`. The following is a list of all methods provided by `IntLSet` that are not simply inherited.

`static IntLSet` **`mkIntSet`**`(int n)`
Returns an integer logical set, with default external name, whose value is a set of `n` unbound integer logical variables—see Sect. 11. If `n < 0`, throws an exception `IllegalArgumentException`.

`static IntLSet empty()`
Returns an empty logic set of integers.

`IntLSet` **`clone`**`()`

`HashSet<Integer>` **`getValue`**`()`

`IntLSet` **`ins`**`(Integer n)`

`IntLSet` **`ins`**`(IntLVar var)`

`IntLSet` **`ins`**`(Integer... n)`

`IntLSet` **`ins`**`(IntLVar... var)`

`IntLSet` **insAll**`(ArrayList<Integer> v)`

`IntLSet` **insAll**`(Integer[] arr)`

`IntLSet` **insAll**`(IntLVar[] arr)`

`IntLSet` **normalizeSet**`()`

`IntLSet` **setName**`(String name)`

Example 8

- *Create a partially specified logical set of integers with at most four elements:*
  `IntLSet d = IntLSet.mkIntSet(4);`
- *Create the empty logical set of integers*
  `IntLSet e = IntLSet.empty();`
- *Create an open logical set of integers containing* `1`, `2` *and an (unbound) integer logical variable*
  `IntLSet f = new IntLSet().ins(1).ins(2).ins(new IntLVar());`

7.3 Constraint methods

Besides all constraints supported by its superclass `LSet`, `IntLSet` supports a few other constraints that take into account the fact that the elements of the set are integers or integer logical variables. The following constraints are applied only to the known elements of the `IntLSet` at the moment the method is called; they ignore the possibly unspecified tail of the set.

`Constraint` **domAll**`(int a, int b)`
: Constructs a constraint which requires that all `IntLVar` objects in this set have domain the interval [`a`..`b`] (see Sect. 11).

`Constraint` **labelAll**`()`
`Constraint` **labelAll**`(LabelingOptions lop)`
: Construct a constraint which forces labeling on all integer logical variables occurring in this set, using the default value choice heuristic (resp., the value choice heuristic specified by `lop`).

8 Logical binary relations: the class `LRel`

A logical binary relation (or, simply, a *l-rel*) is a special kind of logical set whose elements are (only) logical pairs.

In JSetL, logical binary relations are instances of the class `LRel`, which extends the class `LSet`. Since logical binary relations are also logical sets, the former inherit all operations that are available for the latter, such as equality, membership, union, disjunction, etc. Binary relations allow the user to post *relational constraints*, such as identity, inverse (or converse), composition, and so on.

8.1 Constructors

Logical binary relations are logical sets whose values must be pairs so each constructor of this class enforces that.

`LRel()`
`LRel(String extName)`
: Construct an unbound l-rel, with default external name (resp., with external name `extName`).

`LRel(Set<LPair> s)`
`LRel(String extName, Set<LPair> s)`
: Construct a (bound) l-rel, with default external name (resp., with external name `extName`), whose elements are all the logical pairs in the *Java* set `s`.

`LRel(LRel s)`
`LRel(String extName, LRel s)`
: Construct a l-rel, with default external name (resp., with external name `extName`), equivalent to the l-rel `s`. Same as to create an unbound l-rel `x` and to post and solve the constraint `x.eq(s)`.

Example 9

- *Create an unbound l-rel with external name* `"a"`*:*

  ```
  LRel a = new LRel("a");
  ```

- *Create a l-rel which is equivalent to* `a` *and has external name* `"b"`*:*

  ```
  LRel b = new LRel("b", a);
  ```

- *Create a l-rel* `c` *containing the pairs* `(x,2), (y,x)`*, where* `x` *and* `y` *are logical variables:*

  ```
  LVar x = new LVar("x");
  LPair p1 = new LPair(x,2);
  LPair p2 = new LPair(new LVar("y"), x);
  Set<LPair> set = new HashSet<>();
  set.add(p1); set.add(p2);
  LRel c = new LRel(set);
  ```

8.2 Creating new (bound) logical binary relations

The class `LRel` provides some methods to create new bound `LRel` objects.

`static LRel` **`empty`**`()`
: Returns the empty l-rel.

`LRel` **`ins`**`(LPair p)`
: Returns a l-rel containing `p` as its element and `this` as its rest.

`LRel` **`ins`**`(LPair... p)`
`LRel` **`insAll`**`(LPair[] arr)`
`LRel` **`insAll`**`(ArrayList<LPair> v)`
: Return a l-rel which contains all logical pairs in `arr` (resp., in `v`) as its elements and `this` as its rest.

Note that neither `ins` nor `insAll` modify the object on which they are invoked. They construct and return a new object instead.

Example 10

- *Create an empty l-rel:*

  ```
  LRel empty = LRel.empty();
  ```

- *Create a new l-rel which has the same elements of the l-rel* `a`*, with the addition of* `(42,43)`*:*

  ```
  LRel b = a.ins(new LPair(42,43));
  ```

- *Create a new l-rel which has the same elements of* `b`*, with the addition of* `("this", "example")` *and* `(3,9)`*:*

  ```
  LPair[] cElems = {new LPair("this", "example"), new LPair(3,9)};
  LRel c = b.insAll(cElems);
  ```

8.3 General utility methods

The class `LRel` inherits all general utility methods from its superclass `LSet`. Below is a list of general utility methods that are overridden by `LRel`.

`LRel` **clone**`()`

`LRel` **setName**`(String name)`

`HashSet<LPair>` **getValue**`()`

8.4 Constraint methods

Positive constraints

`Constraint` **comp**`(LRel s, LRel q)`
Returns the constraint $comp(this, s, q)$, which is true if and only if q is the *relational composition* of *this* and s, i.e., $q = \{(x, z) | \exists y((x, y) \in this \wedge (y, z) \in s\}$.

`Constraint` **dom**`(LSet a)`
Returns the constraint $dom(this, a)$, which is true if and only if a is the *domain* of *this*, i.e., $a = \{x | \exists y((x, y) \in this)\}$.[1]

`Constraint` **dres**`(LSet a, LRel s)`
Returns the constraint $dres(this, a, s)$, which is true if and only if s is the *domain restriction* of *this* with respect to the set a, i.e., $s = \{(x, y) | (x, y) \in this \wedge x \in a\}$.

`Constraint` **id**`(LSet a)`
Returns the constraint $id(a, this)$, which is true if and only if *this* is the *identity relation* with respect to the set a, i.e., $this = \{(x, x) | x \in a\}$.

`Constraint` **inv**`(LRel s)`
Returns the constraint $inv(this, s)$, which is true if and only if s is the *inverse relation* of *this*, i.e., $s = \{(y, x) | (x, y) \in this\}$.

[1] To be not confused with the (finite) domain associated with an integer logical variable (see Sect. 11), which is used to specify the set of all possible values for that variable.

`Constraint` **ran**`(LSet a)`
Returns the constraint $ran(this, a)$, which is true if and only if a is the *range* of *this*, i.e., $a = \{y | \exists x((x, y) \in this)\}$.

`Constraint` **rres**`(LSet a, LRel s)`
Returns the constraint $rres(this, a, s)$, which is true if and only if s is the *range restriction* of *this* with respect to the set a, i.e., $s = \{(x, y) | (x, y) \in r \wedge y \in a\}$.

Negative constraints

`Constraint` **ncomp**`(LRel s, LRel q)`
Returns the constraint $ncomp(this, s, q)$, which is true if and only if q is not the *relational composition* of *this* and s.

`Constraint` **ndom**`(LSet a)`
Returns the constraint $ndom(this, a)$, which is true if and only if a is not the *domain* of *this*.

`Constraint` **ndres**`(LSet a, LRel s)`
Returns the constraint $ndres(this, a, s)$, which is true if and only if s is not the *domain restriction* of *this* with respect to the set a.

`Constraint` **nid**`(LSet a)`
Returns the constraint $nid(a, this)$, which is true if and only if *this* is not the *identity relation* with respect to the set a.

`Constraint` **ninv**`(LRel s)`
Returns the constraint $ninv(this, s)$, which is true if and only if s is not the *inverse relation* of *this*.

`Constraint` **nran**`(LSet a)`
Returns the constraint $nran(this, a)$, which is true if and only if a is not the *range* of *this*.

`Constraint` **nrres**`(LSet a, LRel s)`
Returns the constraint $nrres(this, a, s)$, which is true if and only if s is not the *range restriction* of *this* with respect to the set a.

9 Logical maps: the class `LMap`

A logical map (or, simply, a *l-map*) is a special kind of logical binary relation whose elements are logical pairs with the additional property that there can't be two pairs with the same first element and different second elements in the same logical map. Hence, logical maps represent *partial functions*.

In JSetL, logical maps are instances of the class `LMap`, which extends the class `LRel`.

9.1 Constructors

`LMap()`
`LMap(String extName)`
Construct an unbound l-map, with default external name (resp., with external name `extName`).

`LMap(Set<LPair> s)`
`LMap(String extName, Set<LPair> s)`
Construct a (bound) l-map, with default external name (resp., with external name `extName`), whose elements are all the logical pairs in the *Java* set `s`.
If `s` can't represent a partial function (i.e., there are two pairs in `s` with the same first element and different ground second elements) calling this method will cause an exception `NotPFunException` to be thrown.

`LMap(LMap s)`
`LMap(String n, LMap s)`
Construct a l-map, with default external name (resp., with external name `extName`), equivalent to the l-map `s`. Same as to create an unbound l-map `x` and to post and solve the constraint `x.eq(s)`.

Example 11

- *Create an unbound l-map with external name* `"a"`*:*

```
LMap a = new LMap("a");
```

- *Create a l-map bound to* {`(1,2),(x,2),(x,y)`}*:*

```
LVar x = new LVar("x"), y = new LVar("y");
Set<LPair> set = new HashSet<>();
set.add(new LPair(1,2)); set.add(new LPair(x,2)); set.add(new LPair(x,y));
LMap b = new LMap(set);
```

- *Trying to create the l-map* {`(x,2),(1,3),(x,3)`} *will throw an exception* `NotPFunException`*:*

```
Set<LPair> set2 = new HashSet<>();
set2.add(new LPair(x,2));
set2.add(new LPair(1,3)); set2.add(new LPair(x,3));
LMap c = new LMap(set2);
```

9.2 Creating new (bound) logical maps

The class `LMap` provides a number of methods to create bound logical maps. Below is a list of such methods: they are basically the same as those found in `LRel` but return a `LMap` and throw an exception `NotPFunException` if the constructed set can not be a partial function.

`static LMap empty()`

`LMap ins(LPair p)`

`LMap ins(LPair... p)`

`LMap insAll(LPair[] arr)`

`LMap insAll(ArrayList<LPair> v)`

9.3 General utility methods

The class `LMap` inherits and supports all general utility methods from its superclasses `LRel` and `LSet`. Below is a list of general utility methods that are overridden by `LMap`.

`LMap` **clone**`()`

`LMap` **setName**`(String name)`

9.4 Constraint methods

Since `LMap`s are also `LRel`s, which in turn are also `LSet`s, `LMap`s support all constraints that can be created by those superclasses. In addition, `LMap` provides a constraint to ensure that an `LMap` object is indeed a partial function (constraint `pfun`), along with some overridden `LRel` constraints which have the same meaning of the corresponding `LRel` constraints but a different implementation that takes into account the special nature of partial functions.

`Constraint` **`comp`**`(LMap s, LMap q)`
`Constraint` **`dom`**`(LSet a)`
`Constraint` **`ran`**`(LSet a)`
: Same as the corresponding methods provided by `LRel` but taking into account the fact that `this` is a partial function.

`Constraint` **`pfun`**`()`
: Returns the constraint *pfun*(*this*), which is true if and only if *this* is a *partial function*, i.e., a logical binary relation such that there can't be two pairs in the relation with the same first element and different second elements. Note that, in this case, the second elements can be either ground or not ground objects.

10 Restricted intensional sets: the class `Ris`

Intensional sets are sets defined by the property that their elements must satisfy; conversely, extensional sets (i.e., the sets implemented by the class `LSet`) are defined by enumerating their elements. For example, if $D = \{1, 2, 3, 4, 5\}$, then $\{2x | x \in D \wedge x \mod 2 = 0\}$ is an intensional set, denoting $\{4, 8\}$.

Restricted intensional sets (abbreviated as RIS) are intentional sets of the form

$$\{c[\boldsymbol{x}] : D \,|\, F[\boldsymbol{x}] \bullet P[\boldsymbol{x}]\}$$

in which c is the *control term*, $\boldsymbol{x} \triangleq \langle x_1, \ldots, x_n \rangle$, $n > 0$, is the vector of all variables occurring in c, D is the *domain* of the control term and must be a finite (but possibly unbounded) set, $F[\boldsymbol{x}]$ is the *filter* and is a constraint involving $\boldsymbol{x}$, and $P[\boldsymbol{x}]$ is the *pattern* and is an expression involving $\boldsymbol{x}$. The intuitive meaning of a RIS is the set of all instances of $P[\boldsymbol{x}]$ such that $c[\boldsymbol{x}]$ belongs to D and $F[\boldsymbol{x}]$ holds. For example, the intensional set $\{2x | x \in D \wedge x \mod 2 = 0\}$ can be written as a RIS as follows:

$$\{x : D \,|\, x \mod 2 = 0 \bullet 2x\}.$$

In `JSetL`, restricted intensional sets are instances of the class `Ris`, which extends `LSet`. Methods provided by `Ris` include most of those provided by `LSet` adapted for RISs, along with some others, such as getters for the control term, domain, filter and pattern of the RIS. Some of the methods provided by `LSet` are not supported by `Ris` and they throw an exception `UnsupportedOperationException` when called.

10.1 Constructors

`Ris(LObject cT, LSet D, Constraint f)`
: Constructs the restricted intensional set {`cT : D | f • cT`}.

`Ris(LObject cT, LSet D, Constraint f, LObject p, LObject... dummyVars)`
Constructs the restricted intensional set $\{$`cT : D | f • p`$\}$, treating each variable in `dummyVars` as a dummy variable (i.e., it creates a new instance of the variables in `dummyVars` for every application of `f` and `p`).

`Ris(Ris r)`
Constructs a restricted intensional set that has the same control term, domain, filter, pattern and dummy variables as the argument `r`.

None of the arguments in the constructors can be `null`, otherwise an exception `NullPointerException` is raised.

Example 12

- *Create the RIS* $\{x : D \mid x < 9 \bullet 2x\}$ *with* D *unbound variable:*

```
IntLSet D = new IntLSet("D");
IntLVar x = new IntLVar("x");
Ris a = new Ris(x, D, x.lt(9), x.mul(2));
```

- *Create the RIS* $\{x : [-3..3] \mid x \mod 2 = 0 \bullet x + 1\}$*:*

```
IntLSet D = new IntLSet("D",-3,3);
IntLVar x = new IntLVar("x");
Ris b = new Ris(x, D, x.mod(2).eq(0), x.sum(1));
```

- *Create the RIS* $\{[x_1, x_2, x_3|R] : D \mid x_1 \neq x_2 \wedge x_2 = x_3 \bullet [x_1, x_2, x_3|R]\}$*:*

```
LSet D = new LSet("D");
LVar x1 = new LVar("x1");
LVar x2 = new LVar("x2");
LVar x3 = new LVar("x3");
LList cT = new LList("R").ins(x1,x2,x3);
Ris c = new Ris(cT, D, x1.neq(x2).and(x2.eq(x3)));
```

10.2 General utility methods

Restricted intensional sets support almost all the general utility methods of `LSet`, though some of them are implemented in a rather different way.

Basic methods

`Ris clone()`

`boolean equals(Object object)`
Returns `true` if and only if the argument `object == this` or if the parameter is a `LSet` and the `Ris` is expandable and its expansion is equal to the parameter `object`.

`String getName()`

`HashSet<?> getValue()`
Calling this method is the same as invoking the method `getValue()` on the `LSet` returned by `this.expand` (so it will throw an exception `IllegalStateException` if this `Ris` is not expandable).

`boolean isBound()`
: Calling this method is the same as invoking the method `isBound() on the domain of this Ris.`

`void output()`
: Outputs to STDOUT a string representation of the `Ris`. The output is of the form "_RIS_NAME = RIS_STRING" where RIS_NAME is the name of the `Ris`, RIS_STRING is the result of the `toString()` method call on this `Ris`.

`Ris setName(String name)`

`String toString()`

The following method, although inherited from `LSet` or from its superclasses, is not supported by `Ris` and will throw an exception `UnsupportedOperationException` when called:

`LSet setValue(Set<?> value)`

Logical collections methods

`int getSize()`
: Calling this method is the same as invoking the method `getSize()` on the `LSet` returned by `this.expand`.

`Ris getTail()`
: Returns the `Ris` itself.

`boolean isClosed()`
: Calling this method is the same as invoking the method `isClosed()` on the domain of this `Ris`.

`boolean isEmpty()`
: Calling this method is the same as invoking the method `isEmpty()` on the domain of this `Ris`.

`boolean isGround()`
: Calling this method is the same as invoking the method `isGround()` on the domain of this `Ris`.

`Iterator<Object> iterator()` Calling this method is the same as invoking the method `iterator()` on the `LSet` returned by `this.expand`.

`void printElems(char sep)` Calling this method is the same as invoking the method `printElems(sep)` on the `LSet` returned by `this.expand`.

`boolean testContains(Object object)`
: Calling this method is the same as calling it on the result of the expansion of the `Ris`.

`Object[] toArray()` Calling this method is the same as invoking the method `toArray()` on the `LSet` returned by `this.expand`.

The following method, although inherited from `LSet` or from its superclasses, is not supported by `Ris` and will throw an exception `UnsupportedOperationException` when called:

`LSet normalizeSet()`

10.3 RIS methods

The class `Ris` introduces some new methods, not provided by `LSet` or by its superclasses.

Getters

The following methods are getters for the control term, domain, filter and pattern of restricted intensional sets.

```
LObject getControlTerm()
```

```
LSet getDomain()
```

```
Constraint getFilter()
```

```
LObject getPattern()
```

RIS expansion

The following methods are used to deal with the expansion of a restricted intensional set to an extensional set. We will say that a `Ris` $\{c : D \mid F] \bullet P\}$ is *expandable* if and only if either D is empty or D contains at least a ground element and the filter F does not contain any free variable. The expanded form of a `Ris` is a `LSet` object containing the result of the application of the pattern to each element of the domain that is ground and satisfies the filter. In particular, if the domain is empty the expansion of the `Ris` is the empty `LSet`.

`boolean isExpandable()`
: Returns `true` if this `Ris` is expandable, `false` otherwise.

`LSet expand()` Returns the logical set representing the expansion of this `Ris` if it is expandable, throws an exception `IllegalStateException` otherwise.

Example 13

- *Create the RIS* $\{x : \{1, 2, 3, y/R\} \mid x > 1 \bullet x * 2\}$ *and expand it:*

```
IntLSet domain = new IntLSet("R")
                .ins(1).ins(2).ins(3).ins(new IntLVar("y")).setName("D");
IntLVar x = new IntLVar("x");
Ris ris = new Ris(x, domain, x.gt(1), x.mul(2));
if( ris.isExpandable()) {
    LSet expandedRis = ris.expand().setName("expandedRis");
    expandedRis.output();
}
```

The generated output will be:

```
_expandedRis = {6,4/{ _x : {_y/_R} | _x > 1 @ _N14, _N14 = _x * 2 }}
```

that is an extensional logical set containing two elements `6` *and* `4` *and a rest part represented by the restricted intensional set* `{_x : {_y/_R} | _x > 1 @ _N14, _N14 = _x * 2 }`*, where* `_N14` *is the internal* `IntLVar` *containing the result of* `x * 2`*.*

10.4 Constraint methods

The class `Ris` supports all constraint methods of `LSet` and its superclasses except for a few that are not supported at the moment.

The following are the l-set constraints supported by `Ris`:

- `eq` and `neq`
- `in` and `nin`
- `contains` and `ncontains`
- `diff` and `ndiff`
- `disj` and `ndisj`
- `inters` and `ninters`
- `less`
- `subset` and `nsubset`
- `union` and `nunion`
- `forallElems`

Below is a list of the unsupported constraint methods. Each of them will throw an exception `UnsupportedOperationException` when called.

- `size`
- `allDiff`

Example 14

- *Post and try to solve some constraints on the RIS defined in Example 13.*

```
IntLSet s = new IntLSet().ins(6).ins(8);
Solver solver = new Solver();
solver.add(s.eq(ris));
domain.output();   // D = {1..3, _y/_R}
solver.check();    // -> true
domain.output();   // -> D = {1..4/_R}
solver.add(ris.contains(0));
solver.check();    // -> false
```

where `{1..3, _y/_R}` *represents the multi-interval* $[1..3] \cup [y..y] \cup R$.

11 Integer logical variables: the class `IntLVar`

Integer logical variables are a special case of the logical variables described in Section 3, in which values are restricted to be integer numbers. Moreover, an integer logical variable has a *finite domain* and a (possibly empty) *integer arithmetic constraint* associated with it.

The domain specifies the set of all possible values that can be bound to the variable and is represented as a *multi-interval* (see Section A.2).

A domain for an integer logical variable v can be specified when the variable v is created, and it is automatically updated when the constraints possibly posted on v are solved in order to maintain constraint consistency. For example, if x and y are integer logical variables both with domain [1..10] and we add the constraint $x > y$, then the domain of x is updated to [2..10] and the domain of y to [1..9]. When the domain of a variable is restricted to a single

value k (i.e., it is a singleton $\{k\}$), the variable becomes *bound* to this value. Conversely, if the domain is reduced to the empty set, it means that the constraints involving that variable are not satisfiable.

The arithmetic constraints associated with integer logical variables are generated by evaluating *integer logical expressions*, i.e., expressions built using the usual arithmetic operators `sum`, `sub`, `mul` and `div`, applied to integer logical variables and integer constants. Evaluating an integer logical expression e yields a new integer logical variable X_1 with an associated constraint

$$X_1 = e_1 \wedge X_2 = e_2 \wedge \ldots \wedge X_n = e_n,$$

where $e_1, \ldots, e_n$ are the subexpressions occurring in e and $X_1, \ldots, X_n$ are internal integer logical variables, that represents a flattened form of the expression e.

For example, if e is the integer logical expression `x.sum(y.sub(1))`, where `x` and `y` are integer logical variables, the evaluation of e returns the integer logical variable X_1 with the associated constraint $X_1 = \mathtt{x} + X_2 \wedge X_2 = \mathtt{y} - 1$.

In JSetL, an integer logical variable is an instance of the class `IntLVar`, which extends the class `LVar`. Values of integer logical variables are integer numbers. Unions of intervals, used to represent variable domains, are instances of the class `MultiInterval` (see Section A.2). Constraints possibly associated with integer logical variables are instances of the class `Constraint` (see Section 14).

The class `MultiInterval` provides the static fields `INF` and `SUP` to represent, respectively, the minimum and maximum representable values for a multi-interval. As a notational convention (see Sections A.1 and A.2), we will denote these values $-\alpha$ and α, respectively, and we will use $\mathbb{Z}_\alpha \stackrel{def}{=} [-\alpha..\alpha]$ to denote the *universe* multi-interval, which corresponds to the maximum representable multi-interval. Moreover, if M is a multi-interval, the notation $\|M\|_\alpha$ will be used to indicate the *normalization* operation over M which is defined as $M \cap \mathbb{Z}_\alpha$.[2]

11.1 Constructors

`IntLVar()`
`IntLVar(String extName)`
: Construct an unbound integer logical variable, with no external name (resp., with external name `extName`). The domain of this variable is the universe (multi-)interval $\mathbb{Z}_\alpha$ (i.e., `[MultiInterval.INF..MultiInterval.SUP]`). The constraint associated with this variable is the empty conjunction.

`IntLVar(Integer k)`
`IntLVar(String extName, Integer k)`
: Construct an integer logical variable, with no external name (resp., with external name `extName`) and value `k`. The domain of this variable is $\|\{\mathtt{k}\}\|_\alpha$ and the associated constraint is the empty conjunction.

`IntLVar(IntLVar v)`
`IntLVar(String extName, IntLVar v)`
: Construct an integer logical variable, with no external name (resp., with external name `extName`), equivalent to the logical variable `v`. The domain and the constraint of this variable are the domain and the constraint of the variable `v` .

[2]Note that `MultiInterval.INF` and `MultiInterval.SUP` have the same value as `Interval.INF` and `Interval.SUP`, respectively; then the latter can either be used in place of the former.

`IntLVar(Integer a, Integer b))`
`IntLVar(String extName, Integer a, Integer b)`
Construct an unbound integer logical variable, with no external name (resp., with external name `extName`) and with domain the multi-interval $\|[\texttt{a..b}]\|_\alpha$. The associated constraint is the empty conjunction.

`IntLVar(MultiInterval m))`
`IntLVar(String extName, MultiInterval m)`
Construct an integer logical variable, with no external name (resp., with external name `extName`) and with domain the multi-interval `m`. The associated constraint is the empty conjunction.

All such constructors may raise an exception `NotValidDomainException` if the domain of the logical variable is empty. In this way, we anticipate a certain failure when trying to solve a constraint involving that variable.

Example 15

- *Create an integer logical variable* `v` *with domain* $\{-1, 1..3\}$

```
MultiInterval m = new MultiInterval();
m.add(-1);
m.add(1);
m.add(2);
m.add(3);
IntLVar v = new IntLVar(m);
```

- *Create an integer logical variable* `v` *with domain* `[MultiInterval.INF..0]` *(since* $\|[\texttt{MultiInterval.INF} - 2..0]\|_\alpha = [-\alpha - 2..0] \cap \mathbb{Z}_\alpha = [-\alpha..0]$*)*

```
IntLVar v = new IntLVar(MultiInterval.INF - 2, 0);
```

Note that when the multi-interval is actually an interval, we use the more standard notation with the square brackets to represent it.

- *Raise an exception* `NotValidDomainException` *(since* $\|\{\texttt{MultiInterval.SUP} + 1\}\|_\alpha = \{\alpha + 1\} \cap \mathbb{Z}_\alpha = \varnothing$*)*

```
IntLVar v = new IntLVar(MultiInterval.SUP + 1);
```

11.2 General utility methods

The class `IntLVar` provides all utility methods of the class `LVar` (see Section 3.2), suitably adapted to `IntLVar` and `Integer` objects, along with a few other methods that take into account the presence of domains and arithmetic constraints.

`boolean` **equals**`(IntLVar lv)`
Returns `true` iff `this` and `lv` are equal logical variables and they have equal domains.

`Constraint` **getConstraint**`()`
Returns the conjunction of constraints associated with this integer logical variable.

`MultiInterval` **getDomain**`()`
Returns the multi-interval representing the domain associated with this integer logical variable.

`void` **output**`()`
Like `output()` of `LVar`, but if the variable is unbound also information about the domain and the arithmetic constraint associated with this variable are printed.

11.3 Integer logical expressions

`IntLVar` objects can be created also by using the (integer) arithmetic operation methods `sum`, `sub`, `mul` and `div`. These methods are invoked on `IntLVar` objects and returns `IntLVar` objects; hence they can be concatenated to form compound arithmetic expressions.

`IntLVar` **sum**`(Integer k)`
Returns an integer logical variable X_1 with an associated constraint $X_1 = X_0 + \mathtt{k} \wedge C_0$, where X_0 is this logical variable and C_0 is the associated constraint.

`IntLVar` **sum**`(IntLVar v)`
Returns an integer logical variable X_1 with an associated constraint $X_1 = X_0 + \mathtt{v} \wedge C_v \wedge C_0$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the logical variable `v`.

`IntLVar` **sub**`(Integer k)`
`IntLVar` **sub**`(IntLVar v)`
Same as above, but with + replaced by − in the associated constraint.

`IntLVar` **mul**`(Integer k)`
`IntLVar` **mul**`(IntLVar v)`
Same as above, but with + replaced by ∗ in the associated constraint.

`IntLVar` **div**`(Integer k)`
Same as above, but with an associated constraint $X_1 = X_0 \ / \ \mathtt{k} \wedge C_0 \wedge k \neq 0$, where X_0 is this logical variable and C_0 is the associated constraint.

`IntLVar` **div**`(IntLVar v)`
Same as above, but with an associated constraint $X_1 = X_0 \ / \ \mathtt{v} \wedge C_v \wedge C_0 \wedge \mathtt{v} \neq 0$, where X_0 is this logical variable, C_0 is the associated constraint and C_v is the constraint associated with the logical variable `v`.

`IntLVar` **truncDiv**`(Integer k)`
Same as `div`, but with the truncated division instead of exact division.

`IntLVar` **truncDiv**`(IntLVar v)`
Same as `div`, but with the truncated division instead of exact division.

`IntLVar` **mod**`(Integer k)`
Same as above, but with an associated constraint $X_1 = X_0 \ mod \ \mathtt{k} \wedge C_0 \wedge k \neq 0$, where X_0 is this logical variable and C_0 is the associated constraint.

`IntLVar` **mod**`(IntLVar v)`
Same as above, but with an associated constraint $X_1 = X_0 \ mod \ \mathtt{v} \wedge C_v \wedge C_0 \wedge \mathtt{v} \neq 0$, where X_0 is this logical variable, C_0 is the associated constraint and C_v is the constraint associated with the logical variable `v`.

Note that the `div` operator refers to the "exact" integer division: a constraint of the form $z = x/y$ is satisfiable iff the constraint $x = z * y \wedge y \neq 0$ is satisfiable. For example, $z = x/y$ with $x = 7$ and $y = 2$ is not satisfiable because the constraint $7 = z * 2$ is unsatisfiable for each integer value that z could take.

Example 16

- *Create an integer logical variable with an associated integer arithmetic constraint.*

```
IntLVar x = new IntLVar("x");
IntLVar y = new IntLVar("y");
IntLVar z = x.sum(y.sub(1)).setName("z");
z.output();
```

Output:

```
_z = unknown -- Constraint: _z = _x + _N4 AND _N4 = _y - 1
```

where `_N4` *represents the internal name of the* `IntLVar` *object created in correspondence with the subexpression* `y.sub(1)`.

Note that the precedence order of operators is implicitly defined by invoking the corresponding methods. For example, the expression `x.sum(y).mul(2)` allows us to represent the arithmetic expression $(\mathtt{x} + \mathtt{y}) \cdot 2$. If instead we wanted to build the term $\mathtt{x} + \mathtt{y} \cdot 2$ we should write something like `x.sum(y.mul(2))`.

11.4 Constraint methods

The class `IntLVar` provides methods for generating the usual arithmetic comparison constraints. Moreover, it provides some methods for generating other kinds of constraints such as domain, membership, all-different and labeling constraints.

Integer comparison constraints

`Constraint` **`eq`**`(Integer k)`
: Returns the constraint $X_0 = \mathtt{k} \wedge C_0$, where X_0 is this logical variable and C_0 is its associated constraint.

`Constraint` **`eq`**`(IntLVar v)`
: Returns the constraint $X_0 = \mathtt{v} \wedge C_0 \wedge C_v$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the logical variable `v`.

`Constraint` **`neq`**`(Integer k)`
`Constraint` **`neq`**`(IntLVar v)`
: Same as above, but with $=$ replaced by $\neq$ in the generated constraint.

`Constraint` **`le`**`(Integer k)`
`Constraint` **`le`**`(IntLVar v)`
: Same as above, but with $=$ replaced by $\leq$ in the generated constraint.

`Constraint` **`lt`**`(Integer k)`
`Constraint` **`lt`**`(IntLVar v)`
: Same as above, but with $=$ replaced by $<$ in the generated constraint.

`Constraint` **`ge`**`(Integer k)`
`Constraint` **`ge`**`(IntLVar v)`
: Same as above, but with $=$ replaced by $\geq$ in the generated constraint.

`Constraint` **`gt`**`(Integer k)`
`Constraint` **`gt`**`(IntLVar v)`
: Same as above, but with $=$ replaced by $>$ in the generated constraint.

Example 17

- *The method invocation*

  ```
  x.sub(1).lt(y.sum(3))
  ```

  where `x` *and* `y` *are unbound logical variables (with external names* `"x"` *and* `"y"`*, respectively), returns the constraint:*

  ```
  _N1 < _N2 AND _N1 = _x - 1 AND _N2 = _y + 3
  ```

  where `_N1` *and* `_N2` *in the* $<$ *constraint represent the internal names of the* `IntLVar` *objects created in correspondence with the subexpressions* `x.sub(1)` *and* `y.sum(3)`*, respectively.*

Domain handling constraints

`Constraint` **dom**`(Integer a, Integer b)`
: Returns the constraint $X_0 :: \|[\texttt{a..b}]\|_\alpha \wedge C_0$, where X_0 is this logical variable and C_0 is its associated constraint. The *domain constraint* $X_0 :: \|[\texttt{a..b}]\|_\alpha$ constrains X_0 to belong to the domain $\|[\texttt{a..b}]\|_\alpha$.
If such domain is empty, an exception `NotValidDomainException` is raised.

`Constraint` **dom**`(MultiInterval m)`
: Same as above, but with domain constraint $X_0 :: \texttt{m}$.

`Constraint` **dom**`(Set<Integer> s)`
: Same as above, but with domain constraint $X_0 :: \|\texttt{s}\|_\alpha$. Throws an exception `NullPointerException` if some of the elements of `s` are `null`.

`Constraint` **ndom**`(Integer a, Integer b)`
`Constraint` **ndom**`(MultiInterval m)`
: Same as the `dom` methods, except that these methods constrain this logical variable to *not* belong to the domain $\|[\texttt{a..b}]\|_\alpha$ (resp., to the multi-interval `m`).

Membership constraints

Membership constraints involve integer logical variables and integer set logical variables (see Section 12)

`Constraint` **in**`(SetLVar X)`
`Constraint` **in**`(MultiInterval A)`
: Return the constraint `this` $\in$ `X` (resp., the constraint `this` $\in$ `A`). Throws an exception `NotValidDomainException` if `MultiInterval A` is empty.

`Constraint` **nin**`(SetLVar X)`
`Constraint` **nin**`(MultiInterval A)`
: Return the constraint `this` $\notin$ `X` (resp., the constraint `this` $\notin$ `A`).

Labeling constraints

Given $n \geq 0$ integer logical variables $v_1, \ldots, v_n$, *labeling* them means try to assign to each variable an integer value belonging to its domain. For a more formal and comprehensive explanation of labeling and its heuristics, see Appendix B.

In this section, we will only list the methods that the class `IntLVar` provides to support labeling. All these methods return an object of class `Constraint`, since the labeling operations for one or more variables are treated as particular kinds of constraints over them.

`Constraint` **`label`**`()`
`Constraint` **`label`**`(ValHeuristic val)`
Label this variable, using the default value choice heuristic `GLB` (resp., using the value choice heuristic `val`).

`static Constraint` **`label`**`(List<IntLVar> vars)`
`static Constraint` **`label`**`(IntLVar... vars)`
Label the variables in `vars`, using the default value and variable choice heuristics `GLB` and `LEFT_MOST` respectively.

`static Constraint` **`label`** `(LabelingOptions lop, List<IntLVar> vars)`
`static Constraint` **`label`** `(LabelingOptions lop, IntLVar... vars)`
Label the variables in `vars`, using the heuristics specified in `lop`.

12 Integer set logical variables: the class `SetLVar`

Integer set logical variables (or more briefly *set variables*) are a special case of the logical variables described in Section 3, in which values are restricted to be set of integers. Like integer logical variables, a set variable has a *finite domain* and a (possibly empty) *set constraint* associated with it. Moreover, each set variable has an associated integer logical variable which represents its *cardinality*.

The domain is represented as a *set-interval*, that is a lattice of integer sets (see Section A.3 for a precise description of set-intervals in JSetL).

A domain for a set variable s can be specified when the variable s is created, and it is automatically updated when the constraints possibly posted on s are solved in order to maintain constraint consistency. For example, if X is a set variable with domain $[\varnothing..\{1,2,3\}]$ and we add the cardinality constraint $|X| = 3$ then the domain of X will be restricted to the singleton $\{\{1,2,3\}\}$, because the only set belonging to the domain of X which has cardinality 3 is precisely $\{1,2,3\}$. Note that, when the domain of a variable is restricted to a singleton $\{A\}$, the variable becomes *bound* to this value. Conversely, if the domain is reduced to the empty set, it means that the constraints involving that variable are not satisfiable. Moreover, when the domain of a set variable is specified or updated, the domain of its cardinality variable is updated accordingly: if the domain of a set variable is the set-interval $[A..B]$ then the domain of its cardinality will be $[|A|..|B|]$.

The constraints associated with set variables are generated by evaluating *integer set expressions*, i.e., expressions built using the usual set operations (union, intersection, difference, complementation, cardinality, ...) applied to set variables and integer set constants. Moreover, when a set constraint is posted, it is possible that other constraints inferable from it are added to the store. For example, when the constraint $X \subseteq Y$ is posted, also the constraint $|X| \leq |Y|$ is added to the store since $X \subseteq Y$ implies that the cardinality of X is less than or equal to the cardinality of Y.

In JSetL, an integer set logical variable is an instance of the class `SetLVar`, which extends the class `LVar`. Values of such variables are set of integers, modeled by objects of class `MultiInterval` (see Section A.2). Set-intervals, used to represent set variable domains, are instances of the class `SetInterval` (see Section A.3). The cardinality variable associated

to a set variable is an instance of the class `IntLVar` (see Section 11). Constraints possibly associated with integer set logical variables are instances of the class `Constraint` (see Section 14).

The class `SetInterval` provides two static fields `INF` and `SUP` to represent, respectively, the minimum and maximum representable values for set intervals. In practice, `INF` is the empty multi-interval, while `SUP` is fixed to be the multi-interval `[-Interval.SUP / 2..Interval.SUP / 2]`. As a notational convention (see Section A.3), we will use β to denote the value of the bounds of the maximum multi-interval (currently fixed to be `Interval.SUP / 2`), and we will use $\mathbb{Z}_\beta \stackrel{def}{=} [-\beta..\beta]$ to represent such multi-interval. Moreover, if D is a set-interval, $\|D\|_\beta$ will be used to indicate the *normalization* operation over D which is defined as $D \cap \mathcal{P}(\mathbb{Z}_\beta)$, while $\mathcal{CH}_\beta$ is used to denote the *convex closure* operation which is defined as $min_\subseteq\{S \in \mathbb{S}_\beta : \|D\|_\beta \subseteq S\}$.

12.1 Constructors

`SettLVar()`
`SetLVar(String extName)`
: Construct an unbound integer set logical variable, with no external name (resp., with external name `extName`). The domain of this variable is the 'universe' set-interval `[SetInterval.INF..SetInterval.SUP]`, which corresponds to the maximum representable set-interval. The constraint associated with this variable is the empty conjunction.

`SetLVar(MultiInterval m)`
`SetLVar(String extName, MultiInterval m)`
: Construct an integer set logical variable, with no external name (resp., with external name `extName`) and value `m`. The domain of this variable is $\|\{\texttt{m}\}\|_\beta$ and the associated constraint is the empty conjunction.

`SetLVar(Set<Integer> s)`
`SetLVar(String extName, Set<Integer> s)`
: Construct an integer set logical variable, with no external name (resp., with external name `extName`) and value `s`. The domain of this variable is $\|\{\texttt{s}\}\|_\beta$ and the associated constraint is the empty conjunction.

`SetLVar(SetLVar l)`
`SetLVar(String extName, SetLVar l)`
: Construct an integer set logical variable, with no external name (resp., with external name `extName`), equivalent to the set variable `l`. The domain and the constraint of this variable are the domain and the constraint of the variable `l` .

`SetLVar(MultiInterval a, MultiInterval b))`
`SetLVar(String extName, MultiInterval a, MultiInterval b)`
: Construct an unbound integer set logical variable, with no external name (resp., with external name `extName`) and with domain the set-interval $\|[\texttt{a..b}]\|_\beta$. The associated constraint is the empty conjunction.

`SetLVar(Set<Integer> s, Set<Integer> t))`
`SetLVar(String extName, Set<Integer> s, Set<Integer> t)`
: Construct an unbound integer set logical variable, with no external name (resp., with

external name `extName`) and with domain the set-interval $\|[\mathtt{s}..\mathtt{t}]\|_\beta$. The associated constraint is the empty conjunction.

`SetLVar(SetInterval s))`
`SetLVar(String extName, SetInterval s)`
: Construct an integer set logical variable, with no external name (resp., with external name `extName`) and with domain the set-interval `s`. The associated constraint is the empty conjunction.

`SetLVar(SetInterval s, Integer k))`
`SetLVar(String extName, SetInterval s, Integer k)`
: Construct an integer set logical variable, with no external name (resp., with external name `extName`), with domain the set-interval `s` and cardinality `k`. The associated constraint is the empty conjunction.

`SetLVar(SetInterval s, MultiInterval m))`
`SetLVar(String extName, SetInterval s, MultiInterval m)`
: Construct an integer set logical variable, with no external name (resp., with external name `extName`), with domain the set-interval `s` and cardinality variable domain `m`. The associated constraint is the empty conjunction.

Note that, like integer logical variables, such constructors may raise an exception `NotValidDomainException` if the domain of the set variable is empty. Moreover, observe that some constructors allow set variables domain to be defined by using generic implementation of `Set<Integer>` interface, which can be different from the class `MultiInterval`.

Example 18

- *Create a* `SetLVar` *with domain* $[\varnothing..[1..3]]$.

```
MultiInterval a  = new MultiInterval();
MultiInterval b  = new MultiInterval(1, 3);
SetLVar x = new SetLVar(a, b);
```

- *Create a* `SetLVar` *with domain* $[\varnothing..\{0,1\}]$ *and cardinality 1 (in fact,* $\mathcal{CH}_\beta(\{\{0\},\{1\},[2..\beta+1]\}) = min_\subseteq\{S \in \mathbb{S}_\beta : \{\{0\},\{1\}\} \subseteq S\} = [\varnothing..\{0,1\}]$*).*

```
MultiInterval m  = new MultiInterval(2, SetInterval.SUP.getLub() + 1);
MultiInterval m0 = new MultiInterval(0);
MultiInterval m1 = new MultiInterval(1);
Vector<MultiInterval> v = new Vector<MultiInterval>();
v.add(m);
v.add(m0);
v.add(m1);
SetInterval s = new SetInterval(v);
SetLVar x = new SetLVar(s, 1);
```

- *Raise an exception* `NotValidDomainException` *(since* $\|\{[2..\beta+1]\}\|_\beta = \{[2..\beta+1]\} \cap \mathcal{P}(\mathbb{Z}_\beta) = \varnothing$*)*

```
MultiInterval m = new MultiInterval(2, SetInterval.SUP.getLub() + 1);
SetLVar x = new SetLVar(m);
```

12.2 General utility methods

The class `SetLVar` provides all utility methods of the class `LVar` (see Section 3.2), suitably adapted to `SetLVar` and `MultiInterval` objects, along with a few other methods that take into account the presence of domains and set constraints.

`Constraint` **getConstraint**`()`
: Returns the conjunction of constraints associated with this integer set logical variable (and with its cardinality).

`SetInterval` **getDomain**`()`
: Returns the set-interval representing the domain associated with this integer set logical variable.

`void` **output**`()`
: Like `output()` of `LVar`, but if the variable is unbound also information about the domain, the cardinality and the arithmetic constraint associated with this variable are printed.

Moreover, `SetLVar` provides a method to compute the cardinality of the set possibly bound to a set variable. Since this method returns an integer logical variable it can be used within integer logical expressions and to post `IntLVar` constraints.

`IntLVar` **card**`()`
: Returns an integer logical variable which represents the cardinality of `this`.

12.3 Integer set expressions

`SetLVar` objects can be created also by using the (integer) set operation methods `compl`, `intersect`, `union`, `diff`, and `singleton`. These methods are invoked on `SetLVar` objects and returns `SetLVar` objects; hence they can be concatenated to form compound set expressions.

`SetLVar` **compl**`()`
: Returns an integer set logical variable X_1 with an associated constraint $X_1 = \sim X_0 \ \wedge \ |X_0| + |X_1| = |\mathbb{Z}_\beta| \ \wedge \ C_0$, where X_0 is this logical variable, $\sim$ is the set complementation with respect to the universe $\mathbb{Z}_\beta$ and C_0 is the constraint associated with X_0.

`SetLVar` **diff**`(MultiInterval m)`
: Returns `this.diff(new SetLVar(m))`.

`SetLVar` **diff**`(SetLVar v)`
: Returns an integer set logical variable X_1 with an associated constraint $X_1 = X_0 \ \setminus \ \mathtt{v} \ \wedge \ X_1 \subseteq X_0 \ \wedge \ \mathtt{v} \ || \ X_1 \ \wedge \ |X_1| \geq |X_0| - |\mathtt{v}| \ \wedge \ C_0 \ \wedge \ C_v$, where X_0 is this logical variable, C_0 its associated constraint and C_v is the constraint associated with the logical variable `v`. The constraint $\mathtt{v} \ || \ X_1$ corresponds to the set disjointness between `v` and X_1 (thus, their intersection must be empty).

`SetLVar` **intersect**`(MultiInterval m)`
: Returns `this.intersect(new SetLVar(m))`.

`SetLVar` **intersect**`(SetLVar v)`
: Returns an integer set logical variable X_1 with an associated constraint

$X_1 = X_0 \cap \mathtt{v} \wedge X_1 \subseteq X_0 \wedge X_1 \subseteq \mathtt{v} \wedge C_0 \wedge C_v$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the integer set logical variable `v`.

`static SetLVar` **`singleton`**`(IntLVar v)`
Returns an integer set logical variable X such that $X = \{\mathtt{v}\}$.

`SetLVar` **`union`**`(MultiInterval m)`
Returns `this.union(new SetLVar(m))`.

`SetLVar` **`union`**`(SetLVar v)`
Returns an integer set logical variable X_1 with an associated constraint $X_1 = X_0 \cup \mathtt{v} \wedge X_0 \subseteq X_1 \wedge \mathtt{v} \subseteq X_1 \wedge |X_1| \leq |X_0| + |\mathtt{v}| \wedge C_0 \wedge C_v$ where X_0 is this logical variable, C_0 its associated constraint and C_v is the constraint associated with the integer set logical variable `v`.

Example 19

- *Create an integer set logical variable with an associated set complement constraint:*

```
SetLVar x = new SetLVar("x");
SetLVar y = x.compl().setName("y");
y.output();
```

Output:

```
_y = _N2 -- Domain: [{}..[-536870911..536870911]]
      -- Size: [0..1073741823] -- Constraint: _N2 = compl(_x)
```

where `_N2` *represents the internal name of the* `SetLVar` *object created in correspondence with the subexpression* `x.compl()`.

12.4 Constraint methods

The class `SetLVar` provides methods for generating the usual set-theoretic constraints. Moreover, it allows to deal with set domains, labeling and partially specified sets.

Integer set constraints

`Constraint` **`disj`**`(MultiInterval m)`
Returns the constraint $X_0 \mid\mid \mathtt{m} \wedge C_0$, where X_0 is this logical variable, C_0 is its associated constraint and $\mid\mid$ is the set disjointness.

`Constraint` **`disj`**`(SetLVar v)`
Returns the constraint $X_0 \mid\mid \mathtt{v} \wedge |X_0| + |\mathtt{v}| \leq |\mathbb{Z}_\beta| \wedge C_0 \wedge C_v$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the logical variable `v`.

`Constraint` **`eq`**`(MultiInterval m)`
Returns the constraint $X_0 = \mathtt{m} \wedge C_0$, where X_0 is this logical variable and C_0 is its associated constraint.

`Constraint` **`eq`**`(SetLVar v)`
Returns the constraint $X_0 = \mathtt{v} \wedge |X_0| = |\mathtt{v}| \wedge C_0 \wedge C_v$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the logical variable `v`.

`Constraint` **neq**`(MultiInterval m)`
Returns the constraint $X_0 \neq \texttt{m} \ \wedge \ C_0$, where X_0 is this logical variable and C_0 is its associated constraint.

`Constraint` **neq**`(SetLVar v)`
Returns the constraint $X_0 \neq \texttt{v} \ \wedge \ C_0 \ \wedge \ C_v$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the logical variable `v`.

`Constraint` **strictSubset**`(MultiInterval m)`
Returns the constraint $X_0 \subseteq \texttt{m} \ \wedge \ |X_0| < |\texttt{m}| \ \wedge \ C_0$, where X_0 is this logical variable and C_0 is its associated constraint.

`Constraint` **strictSubset**`(SetLVar v)`
Returns the constraint $X_0 \subseteq \texttt{v} \ \wedge \ |X_0| < |\texttt{v}| \ \wedge \ C_0 \ \wedge \ C_v$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the logical variable `v`.

`Constraint` **subset**`(MultiInterval m)`
Returns the constraint $X_0 \subseteq \texttt{m} \ \wedge \ C_0$, where X_0 is this logical variable and C_0 is its associated constraint.

`Constraint` **subset**`(SetLVar v)`
Returns the constraint $X_0 \subseteq \texttt{v} \ \wedge \ |X_0| \leq |\texttt{v}| \ \wedge \ C_0 \ \wedge \ C_v$, where X_0 is this logical variable, C_0 is its associated constraint and C_v is the constraint associated with the logical variable `v`.

Example 20

- *Generate the constraint* $X \subseteq Y \cup Z$.

```
X.subset(Y.union(Z));
```

- *Generate the constraint* $X \cap Y = Y \setminus X$.

```
X.intersect(Y).eq(Y.diff(X));
```

- *Generate the constraint* $X \neq \{-2, 7\}$.

```
X.neq(new MultiInterval(-2).union(new MultiInterval(7)));
```

Domain handling constraints

`Constraint` **dom**`(MultiInterval a, MultiInterval b)`
Returns the constraint $X_0 :: \|[\texttt{a..b}]\|_\beta \ \wedge \ C_0$, where X_0 is this logical variable and C_0 is its associated constraint. The domain constraint $X_0 :: \|[\texttt{a..b}]\|_\beta$ constrains X_0 to belong to the domain $\|[\texttt{a..b}]\|_\beta$.
If such domain is empty, an exception `NotValidDomainException` is raised.

`Constraint` **dom**`(SetInterval s)`
Same as above, but with domain constraint $X_0 :: \texttt{s}$.

Labeling constraints

Given $n \geq 0$ integer set logical variables $v_1, \ldots, v_n$, *labeling* them means try to assign to each variable an integer set value belonging to its domain. For a more formal and comprehensive explanation of labeling and its heuristics, see Appendix B.

In this section, we will only list the methods that the class `SetLVar` provides to support labeling. As for class `IntLVar`, all these methods return an object of class `Constraint`, since the labeling operations on one or more variables are treated as particular kinds of constraints over them.

`Constraint` **label**`()`
`Constraint` **label**`(ValHeuristic val)`
Label this variable, using the default value choice heuristic `GLB` (resp., using the value choice heuristic `val`) and the default set heuristic `FIRST_NIN`.

`static Constraint` **label**`(List<SetLVar> vars)`
`static Constraint` **label**`(SetLVar... vars)`
Label the variables in `vars`, using the default heuristics `GLB`, `LEFT_MOST` and `FIRST_NIN`.

`static Constraint` **label**`(LabelingOptions lop, List<SetLVar> vars)`
`static Constraint` **label**`(LabelingOptions lop, SetLVar... vars)`
Label the variables in `vars`, using the heuristics specified in `lop`.

Partially specified integer sets

The class `SetLVar` allows to define *partially specified* integer sets according to the CLP($\mathcal{SET}$) approach. Specifically, if $X_1, \ldots, X_n$ are integer logical variables ($n \geq 0$) and S and R are integer set logical variables, then we can define and solve constraints of the form:

$$S = \{X_1, \ldots, X_n \mid R\}.$$

whose meaning is $S = \{X_1\} \cup \ldots \cup \{X_n\} \cup R$. The following methods allow the user to define this kind of constraints.

`Constraint` **eq**`(IntLVar x)`
Returns the constraint $X_0 = \{\texttt{x}\} \wedge C_0$, where X_0 is this logical variable and C_0 is its associated constraint.

`Constraint` **eq**`(IntLVar x, SetLVar R)`
Returns the constraint $X_0 = \{\texttt{x}|\texttt{R}\} \wedge C_0$, where X_0 is this logical variable and C_0 is its associated constraint.

`Constraint` **eq**`(IntLVar[] vars)`
`Constraint` **eq**`(Collection<IntLVar> vars)`
Returns the constraint $X_0 = \{X_1, \ldots, X_n\} \wedge C_0$, where X_0 is this logical variable, C_0 is its associated constraint and $X_1, \ldots, X_n$ is the collection of integer logical variables belonging to `vars`.

`Constraint` **eq**`(IntLVar[] vars, SetLVar r)`
`Constraint` **eq**`(Collection<IntLVar> vars, SetLVar r)`
Returns the constraint $X_0 = \{X_1, \ldots, X_n \mid \texttt{r}\} \wedge C_0$, where X_0 is this logical variable, C_0 is its associated constraint and $X_1, \ldots, X_n$ is the collection of integer logical variables belonging to `vars`.

Note that, in the current implementation, constraints of the form $S = \{X_1, \ldots, X_n \mid R\}$ are simply unfolded in n union constraints:

$$S = S_1 \ \cup \ \ldots \ \cup \ S_n \ \cup \ R \qquad \text{where } S_i = \{X_i\} \text{ for each } i = 1, \ldots, n.$$

Moreover, in order to represent each singleton $S_i = \{X_i\}$, the following constraints are added to the constraint store, for $i = 1, \ldots, n$:

$$X_i \in S_i \ \wedge \ |S_i| = 1.$$

13 Boolean logical variables: the class `BoolLVar`

Boolean logical variables are a special kind of `LVar` (see Section 3). As `LVar`s, Boolean logical variables may be unbound or bound and, when bound, their value is restricted to being `true` or `false`. Moreover, a boolean logical variable has a (possibly empty) *boolean constraint* associated with it, that is a flat representation of the boolean expression defining that logical variable, in the same way as seen in 11.

13.1 Constructors

`BoolLVar()`
`BoolLVar(String extName)`
: Construct an unbound `BoolLVar` with an empty constraint associated with it and with default name (resp., with external name `extName`).

`BoolLVar(boolean b)`
`BoolLVar(String extName, boolean b)`
: Construct a `BoolLVar` bound to the boolean value `b`, with an empty constraint associated with it, and with default name (resp., with external name `extName`).

`BoolLVar(BoolLVar b)`
`BoolLVar(String extName, BoolLVar b)`
: Construct a `BoolLVar` object, with default external name (resp., with external name `extName`), equivalent to the `BoolLVar` object `b`. Same as to create an unbound `BoolLVar` object `x` and to post and solve the constraint `x.eq(b)`.

Example 21

- *Create an unbound boolean logical variable with name* `"a"`*:*

  ```
  BoolLVar a = new BoolLVar("a");
  ```

- *Create a boolean logical variable with name* `"b"` *and value* `true`*:*

  ```
  BoolLVar b = new BoolLVar("b", true);
  ```

- *Create a boolean logical variable which is equivalent to* `a`

  ```
  BoolLVar c = new BoolLVar(a);
  ```

13.2 General utility methods

`BoolLVar` inherits all methods of `LVar`, and provides some new methods of its own. Here is a list of the methods that are overridden by `BoolLVar`.

`BoolLVar clone()`

`Boolean getValue()`

`Constraint` **getConstraint**`()` Returns the conjunction of constraints associated with this boolean logical variable.

`void output()` Like `output()` of `LVar`, but if the variable is unbound also information about the boolean constraint associated with this variable are printed.

`BoolLVar setName(String name)`

Below is a list of the methods that are declared in `BoolLVar`.

`boolean isFalse()`
Returns true if this boolean logical variable is bound and its value is false, returns false otherwise.

`boolean isTrue()`
Returns true if this boolean logical variable is bound and its value is true, returns false otherwise.

13.3 Boolean logical expression

The class `BoolLVar` provides methods for constructing boolean logical variables whose value is equal to the result of a boolean expression containing other (possibly unbound) boolean logical variables. Below is a list of such methods and a brief description of their meaning.

`BoolLVar and(BoolLVar l)`
Returns a `BoolLVar` which is the result of the expression $this \wedge l$.

`BoolLVar or(BoolLVar l)`
Returns a `BoolLVar` which is the result of the expression $this \vee l$.

`BoolLVar not()`
Returns a `BoolLVar` which is the result of the expression $\neg this$.

`BoolLVar implies(BoolLVar l)`
Returns a `BoolLVar` which is the result of the expression $this \implies l$.

`BoolLVar iff(BoolLVar l)`
Returns a `BoolLVar` which is the result of the expression $this \iff l$.

13.4 Constraint methods

There are three kinds of constraints that can be posted specifically for `BoolLVar`: equality, inequality and labeling. Below is a list of the methods used to construct equality and inequality constraints.

`Constraint eq(BoolLVar b)`
`Constraint eq(Boolean b)`
: Return a constraint representing the equality between `this` and `b`. The returned constraint is conjoined with the boolean constraints (possibly) associated with `this` (and `b`).

`Constraint neq(BoolLVar b)`
`Constraint neq(Boolean b)`
: Returns a constraint which represents the inequality between `this` and the parameter `b`. The returned constraint is conjoined with the boolean constraints (possibly) associated with `this` (and `b`).

Below is a list of the methods available for the labeling of boolean logical variables.

`Constraint label()`
: Returns the constraint which forces this variable to be labeled with values from {true, false}.

`Constraint label(BoolHeuristic opt)`
`Constraint label(LabelingOptions opt)`
: Return the constraint which forces this variable to be labeled with values from {true, false}, according to the heuristic (resp., the labeling options) given by *opt*.

```
static Constraint label(List<BoolLVar> vars)
static Constraint label(BoolLVar...  vars)
static Constraint label(LabelingOptions opt, List<BoolLVar> vars)
static Constraint label(LabelingOptions opt, BoolLVar...  vars)
```
: Return the constraint which forces each variable in *vars* to be labeled with values from {true, false}, using the default labeling options (resp., the options given by *opt*). Throw an exception `NullPointerException` if some of the variables in the collections (or the parameters themselves) are `null`.

Example 22 *Boolean constraints*

```
Solver solver = new Solver();
BoolLVar d = a.or(b.and(c.not()));
solver.add(d.eq(true).and(a.label()));
solver.check();
```

now a *is* false *and so is* c

```
solver.nextSolution();
```

now a *is* true *and so is* c*; there are no more solutions.*

14 Constraints: the class `Constraint`

Constraints represent operations that can be applied to logical variables and logical collections, as well as to objects created through their derived sub-classes. These operations can be performed even if the involved logical objects have no precise value associated with them.

A constraint in JSetL is an expression that can take one of the forms:

- *atomic constraints*:

  - the *empty constraint*, denoted `[]`
  - e_0.*op* $(e_1, \ldots, e_n)$ or *op* $(e_0, e_1, \ldots, e_n)$ with $n = 0, \ldots, 3$

    where *op* is the name of the constraint and e_i $(0 \leq i \leq 3)$ are expressions whose type depends on *op*. In particular *op* can be one of a collection of predefined methods that implement general operations, such as equality, inequality, integer comparison, as well as basic set-theoretic operations, such as membership, union, intersection, etc.

- *compound constraints*:
  - c_1.`and` (c_2) (conjunction)
  - c_1.`or` (c_2) (disjunction)
  - c_1.`orTest` (c_2) (disjunction)
  - c_1.`impliesTest` (c_2) (implication)

  where c_1 and c_2 are JSetL constraints and `and`, `or`, `orTest`, `impliesTest` represent the logical conjunction $(c_1 \wedge c_2)$, disjunction $(c_1 \vee c_2)$, and implication $(c_1 \rightarrow c_2)$, between c_1 and c_2, respectively.

- *negation constraint*:
  - c_1.`notTest` () (negation)

  where c_1 is a JSetL constraint and `notTest` represents the negation of c_1 $(\neg c_1)$.

Constraints in JSetL are defined as instances of the class `Constraint`. `Constraint` objects are created by using constructors and other methods of the class `Constraint` (e.g., `and`, `or`), as well as the result of calling a number of *constraint methods* supplied by the classes implementing logical objects presented in the previous sections.

14.1 Constructors

`Constraint()`
Constructs the empty constraint (default name: `"no name"`).

`Constraint(String constrName, Object... arguments)`
Constructs a constraint with name `constrName` and a sequence of `0` to `Constraint.MAX_ARGUMENTS_PER_CONSTRAINT` arguments, as specified by `arguments`. If too many arguments are provided, an exception `IllegalArgumentException` is thrown.

`constrName` must be the name of a user-defined constraint. It can be any string value, not beginning with character `'_'`. In fact, names beginning with `'_'` are reserved for library defined constraints (e.g., `"_eq"`, `"_nin"`, `"_union"`, etc.).

Example 23

- *Create an empty constraint:*

```
Constraint emptyConstraint = new Constraint();
```

- *Create a user defined constraint with three arguments:*

```
Constraint c = new Constraint("myConstraint","arg1",2,new LList("arg3"));
```

14.2 Static members

`static final Constraint.MAX_ARGUMENTS_PER_CONSTRAINT`
The maximum number of arguments that can be used in the constructor of `Constraint`. This number is currently set to `4`.

`static Constraint truec()`
`static Constraint falsec()`
Return a constraint that is always (resp., never) satisfiable.

14.3 General utility methods

`Constraint` **clone**`()`
Returns a copy of this constraint, creating a clone of each atomic constraint in the conjunction.

`boolean` **equals**`(Constraint c)`
`boolean` **equals**`(Object o)`
Return `true` if the argument is a constraint and all the atomic constraints occurring in this constraint and in the argument constraint are ordinately equals. Atomic constraints are considered equals if all their non-null arguments are equal.

`Object` **getArg**`(int i)`
Returns the `i`-th argument of this constraint if $1 \leq$ `i` $\leq k$ (where k is the number of arguments of this constraint); `null` otherwise. Note that if applied to a constraint conjunction, `getArg(1)` returns the first conjunct, whereas `getArg(2)` returns the rest of the conjunction.

`String` **getName**`()`
Returns the name associated with this constraint. Note that if applied to a constraint conjunction, `getName()` returns `"and"`.

`boolean` **isGround**`()`
Returns `true` if this constraint does not contain any unbound logical object.

`String` **toString**`()`
Returns the string corresponding to the "external view" of this constraint (e.g., using standard infix arithmetic operators).

`String` **toStringInternals**`()`
Returns the string corresponding to the "internal view" of this constraint, i.e.,

$$\texttt{constraint(name,arg}_1\texttt{,arg}_2\texttt{,arg}_3\texttt{,arg}_4\texttt{)},$$

where $\texttt{arg}_i$ is either the i-th argument of this constraint (if $1 \leq i \leq k$) or `null` (if $i > k$), where k is the number of arguments of this constraint.

14.4 Global constraints

`static Constraint` **allDifferent**`(List<?> objs)`
`static Constraint` **allDifferent**`(Object... objs)`
Return a constraint c that is satisfied if and only if all elements e_i in `objs`, $n \geq 0$, are different from each other, i.e., $c = \bigwedge_{1 \leq i < j \leq n} e_i \neq e_j$.

14.5 Meta-constraints

Meta-constraints allow to create new constraints starting from existing ones.

`Constraint` **and**`(Constraint c)`
Returns the constraint `this` $\wedge$ `c`.

`Constraint` **impliesTest**`(Constraint c)`
Returns the constraint `this` $\rightarrow$ `c`, where $\rightarrow$ is the logical implication.

`Constraint` **notTest**`()`
Returns the constraint $\neg$`this`, where $\neg$ is the logical negation.

`Constraint` **or**`(Constraint c)`
`Constraint` **orTest**`(Constraint c)`
Return the constraint `this` $\vee$ `c`.

The difference between `or` and `orTest` is that the latter is just a test over two ground (i.e., completely specified) constraints, and it is simply left unchanged by the solver if either `c` or `this` are not ground; conversely, the former is always evaluated even if `c` or `this` are not ground, using backtracking to try the second constraint if the first fails. As an example, the constraint:

```
x.eq(1).orTest(x.eq(2)).and(x.neq(1)).and(x.neq(2))
```

where `x` is an unbound logical variable, is simply left unchanged when the solver tries to solve it, whereas the (logically equivalent) constraint

```
x.eq(1).or(x.eq(2)).and(x.neq(1)).and(x.neq(2))
```

is found to be unsatisfiable by the solver.

Similar considerations apply to constraints `notTest` and `impliesTest`.

In addition to the above constraint methods, the following method is also provided which is logically equivalent to `and` but modifies the invocation object.

`void` **add**`(Constraint c)`
Modifies this constraint in such a way it represents `this` $\wedge$ `c`.

14.6 Constraint methods in other classes

Constraints are generated also by a number of methods provided by classes implementing logical variables and logical collections.

Specifically:

- constraints over `LVar` objects: see Sect. 3.3
- constraints over `LList` objects: see Sect. 4.4
- constraints over `LPair` objects: see Sect. 5.3
- constraints over `LSet` objects: see Sect. 6.4
- constraints over `IntLSet` objects: see Sect. 7.3
- constraints over `LRel` objects: see Sect. 8.4

- constraints over `LMap` objects: see Sect. 9.4
- constraints over `IntLvar` objects: see Sect. 11.4
- constraints over `SetLVar` objects: see Sect. 12.4.
- constraints over `BoolLVar` objects: see Sect. 13.4.
- constraints over `Ris` objects: see Sect. 10.4.

14.7 Constraint-solving methods

The class `Constraint` provides methods to solve constraints. Since these methods internally use a temporary solver, they are subject to the same restrictions for the similar methods in `Solver` (see Sect. 15.5). Moreover, since the local temporary solver is created when the methods are called, only one of the following methods can be called on a single constraint, and it can be called only once.

- `boolean check()`
  Solves this constraint using a local solver; returns `true` if the constraint is satisfiable, `false` otherwise.
- `void solve()`
  Solves this constraint using a local solver; throws a `Failure` exception if the constraint is not satisfiable.
- `boolean test()`
  Solves this constraint using a local solver but leaving the constraint and all logical objects occurring in it unchanged; returns `true` if the constraint is satisfiable, `false` otherwise.
- `LSet setof(LVar x)`
  Returns an `LSet` object whose elements are all possible solutions for `x` which satisfy this constraint conjunction.

14.8 Control methods

The following methods are provided to support nondeterminism handling in conjunction with user-defined constraint facilities (see Sect. 16). A brief description of each method can be found in Sect. 16.3.

`void` **fail**`()`

`int` **getAlternative**`()`

`void` **notSolved**`()`

15 Constraint solving: the class `Solver`

Constraints are solved using a *constraint solver*. In JSetl a constraint solver can be created as an instance of the class `Solver`. Basically, this class provides methods for posting constraints, i.e., adding constraints to the current collection of constraints (*constraint store*), as well as inspecting, checking satisfiability, and finding (all) solutions of the posted constraints.

The class `Solver` and its methods are described in detail in this section.

15.1 Constructor

`Solver()`
: Constructs a constraint solver, with the empty constraint in its constraint store.

15.2 Posting and inspecting constraints

Constraints can be posted to a specific constraint solver by adding them to its constraint store. The collection of constraints in the constraint store is logically interpreted as a conjunction of constraints. Each addition to the constraint store adds a conjunct to the constraint conjunction represented by the store.

`void` **`add`**`(Constraint c)`
: Adds a constraint `c` (either atomic or compound) to the constraint store of this solver. No processing of the added constraint is performed at this stage.

`void` **`addChoicePoint`**`(Constraint c)`
: See Sect. 16.3.

`void` **`clearStore`**`()`
: Removes all constraints from the constraint store of this solver. It also removes all choice-points possibly associated with the current collection of constraints.

`Constraint` **`getConstraint`**`()`
: Returns the conjunction of non-solved constraints stored in the constraint store of this solver.

`void` **`showStore`**`()`
: Prints the conjunction of non-solved constraints stored in the constraint store of this solver. Same as printing the result of `this.getConstraint()`.

`void` **`showStoreAll`**`()`
: Prints the conjunction of all the constraints stored in the constraint store of this solver, including solved constraints which have been possibly left in the constraint store. Basically used for debugging purposes.

`void` **`showStoreInternals`**`()`
: Like `showStoreAll` but it prints constraints in their internal format (see Sect. 14, method `toStringInternals()`). Basically used for debugging purposes.

`int` **`size`**`()`
: Returns the number of non-solved constraints stored in the constraint store of this solver.

Remark. The statement `solver.add(`c_1`.and(`c_2`)....and(`c_n`))` is equivalent to the sequence of statements:

```
solver.add(c1);
solver.add(c2);
...
solver.add(cn);
```

The order in which atomic constraints are added to the constraint store is completely immaterial.

15.3 Checking constraint satisfiability

`boolean` **check**`(Constraint c)`

If C is the constraint currently in the constraint store of this solver, checks whether the constraint $C \wedge$ `c` is satisfiable or not, and returns `true` or `false`, respectively. If $C \wedge$ `c` is satisfiable, a viable constraint solution (i.e., a set of substitutions for the unbound logical objects occurring in the constraint) is also computed, if possible. Computing a solution may involve nondeterminism. The resulting constraint store will contain a possibly simplified form of the constraint $C \wedge c$. Conversely, if $C \wedge$ `c` is unsatisfiable, all unbound variables in $C \wedge$ `c` and the constraint store remain unchanged.

`boolean` **check**`()`

Same as `check(c)` in which `c` is the empty constraint.

`void` **failure**`()`

Raises a `Failure` exception.

`void` **solve**`(Constraint c)`
`void` **solve**`()`

Same as `check(c)` (resp., `check()`) but if the constraint $C \wedge c$ (resp., C) is found to be unsatisfiable, a `Failure` exception is raised.

`boolean` **test**`()`

Tests whether the constraint in the store is satisfiable or not, like `check()` does, but it leaves everything unchanged after it returns.

15.4 Getting solutions

`boolean` **nextSolution**`()`

If issued after a `check` or a `solve` or another `nextSolution`, it tries to compute the next solution for the constraint in the constraint store of this solver. If a solution exists, it returns `true`; otherwise, it returns `false`. In the last case, the content of the resulting constraint store is undefined.

`LSet` **setof**`(LVar x, Constraint c)`

If C is the constraint currently in the constraint store of this solver, returns the logical set obtained by adding to it all solutions for `x` that makes the constraint $C \wedge$ `c` satisfiable. If $C \wedge$ `c` is unsatisfiable, it returns the empty `LSet`. In all cases, all unbound variables in $C \wedge$ `c` remain unchanged.

`LSet` **setof**`(LVar x)`

Same as `setof(x,c)` in which `c` is the empty constraint.

`int` **forEachSolution**`(Consumer<Integer> consumer)`

If C is the constraint currently in the constraint store of this solver, runs the given `consumer` for each solution of C passing the index of the solution (starting from 1) to the consumer. At the end it returns the number of computed solutions.

`Integer` **maximize**`(IntLVar x)`

If C is the constraint currently in the constraint store of this solver, and `x` is a variable in C, returns the maximum value of `x` for which C is satisfiable. If C is unsatisfiable, returns `java.lang.Integer.MIN_VALUE/2 + 1`.

`Integer` **minimize**`(IntLVar x)`

If C is the constraint currently in the constraint store of this solver, and `x` is a variable in C, returns the minimum value of `x` for which C is satisfiable. If C is unsatisfiable, returns `java.lang.Integer.MAX_VALUE`.

Example 24

- *Print all solutions.*

```
LVar x = new LVar("x");
LSet s = LSet.empty().ins(3).ins(2).ins(1).setName("s");
Solver solver = new Solver();
solver.solve(x.in(s));
do {
    x.output();
} while(solver.nextSolution());
System.out.println("No more solutions");
```

or

```
Solver solver = new Solver();
LVar x = new LVar("x");
LSet s = LSet.empty().ins(3).ins(2).ins(1);
solver.add(x.in(s));
solver.forEachSolution(()-> {x.output();});
System.out.println("no more solutions");
```

Executing this code will output:

```
x = 1
x = 2
x = 3
No more solutions
```

- *Collect all solutions.*

```
LVar x = new LVar("x");
LSet s = LSet.empty().ins(3).ins(2).ins(1).setName("s");
LSet r = solver.setof(x,x.in(s));
r.output();
```

Executing this code will output:

```
? = {1,2,3}
```

- *No solutions.*

```
LVar x = new LVar("x");
LSet s = LSet.empty().ins(2).ins(1).setName("s");
solver.add(x.neq(1).and(x.neq(2)));
LSet r = solver.setof(x,x.in(s)).setName("r");
r.output();
```

Executing this code will output:

```
r = {}
```

15.5 Restrictions

In order to guarantee the correct behaviour of the backtracking mechanism, the usage of solvers is limited by some restrictions. If these restrictions are not met, the behaviour of the solvers on which the restrictions are violated is not guaranteed to be correct (and in most cases it will indeed be wrong).

There are three main kinds of restrictions.

- **Thread restrictions**. This restriction regards the concurrent executions in multiple threads. All the logical objects appearing in the constraint store of a solver must be created in the same thread as the solver itself. Moreover, each solver can only be used (e.g., for adding and solving constraints) in the thread where it was created.

- **Shared logical objects** This restriction regards the sharing of logical objects among different solvers. No logical objects must be shared among multiple solvers. When a constraint is posted to a solver all the logical objects appearing in it should be considered bound to that solver and thus not available for the posting of constraints in other solvers.

15.6 Optimization Options

The class `Solver` provides some options that allow the user to specify different behaviours of the solver (for example the usage of different rewrite rules for some constraints) in order to increase performances or to get more explanatory answers.

Each instance of `Solver` contains an instance of the static class `Solver.OptimizationOptions`, which can be retrieved by using the `Solver` method `getOptimizationOptions()` to check which optimizations are enabled and to turn on or off each option. Here is a list of the optimizations which are available for the general user.[3]

- *fast* `comp` *rules*: using different rewrite rules for relational composition (constraint `comp`), may dramatically increase performance. This optimization is enabled by default.

- *fast* `union` *rules*: using different rewrite rules for union of logical sets (constraint `union`), may increase performance if large sets are involved. Note that this optimization does not currently work if instances of `Ris` are involved in `union` constraints and in general may not work if they are present in the constraint store. For this reason this optimization is not enabled by default.

- *RIS expansion optimization*: expands `Ris` (see Sect. 10.4) to logical sets (which may have a `Ris` tail) whenever possible when solving constraints involving `Ris` objects. This optimization is enabled by default since itmay greatly improve performances with ground elements of the domain of `Ris`.

- *Ris expansion cache*: uses a cache to store the expansion of the ground elements of `Ris` to improve performance. Adds a little overhead but greatly improve performance of expandable recursive `Ris`. The default size of the cache is 1000.

The following methods of `Solver.OptimizationOptions` return `true` if the corresponding optimization is enabled, `false` otherwise.

[3]Few other optimization options are mainly intended to be used by the library developers and are not described in this manual.

```
boolean areFastCompRulesEnabled()

boolean areFastUnionRulesEnabled()

boolean isRisExpansionOptimizationEnabled()

boolean isRisExpansionCacheEnabled()
```

The following methods of `Solver.OptimizationOptions` take one boolean input and enable (resp., disable) the corresponding optimization if it is true (resp., false).

```
void setUseFastCompRulesFlag(boolean flag)

void setUseFastUnionRulesFlag(boolean flag)

void setUseSetUnificationOptimizationsFlag(boolean flag)

void setUseRisExpansionCacheFlag(boolean flag)

void setRisExpansionCacheSize(int maxEntries)
```

16 User-defined constraints

JSetL allows the user to define new, possibly nondeterministic, constraints and to deal with them as the built-in constraints.

16.1 The class `NewConstraints`

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

```
public class MyOps extends NewConstraints {
// public and private methods implementing new constraints
}
```

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

Once objects of the new class have been created, one can use the user-defined constraints contained in it as the built-in ones: user-defined constraints can be added to the constraint store using the method `add` and solved using the `Solver` methods for constraint solving.

For example, the statements

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

create an object of type `MyOps`, called `myOps`, and use it to invoke and solve the constraint `c1` over two objects `o1` and `o2`, using the constraint solver `solver` (provided `c1` is one of the constraints defined in `MyOps`).

16.2 Implementing new constraints

The actual implementation of the class that extends the JSetL abstract class `NewConstraints` requires some programming conventions to be respected. The following example shows the implementation of the class `MyOps` which offers two new constraints `c1(o1,o2)` and `c2(o3)`, where `o1`, `o2`, `o3` are objects of type `t1`, `t2`, and `t3`, respectively.

Example 25 *(Implementing new constraints)*

```
public class MyOps extends NewConstraints{
   public MyOps(Solver currentSolver) {
       super(currentSolver);
   }

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

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

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

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

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

The other public methods simply construct and return new objects of class `Constraint`. Each different constraint is identified by a string name which is specified as a parameter of the constraint constructor.

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

Finally, the private methods provide the implementation of the new constraints. These methods must, first of all, retrieve the constraint arguments, whose number and type depend on the constraint itself.

The following is an example of the definition of a class derived from `NewConstraints` that implements, among others, a new constraint `absTest`. `absTest(x,y)`, where `x` and `y` are `IntLVar`, is true if `x` is bound and `y = |x|`; if `x` is unbound, the constraint is simply left unchanged.

Example 26 *(New constraint* `absTest`*)*

```
public class MathOps extends NewConstraints{
    public MathOps(Solver s) {
        super(s);
    }

    public Constraint absTest(IntLVar x, IntLVar y) {
        return new Constraint("absTest", x, y);
    }

    protected void user_code(Constraint c)
    throws NotDefConstraintException {
        if (c.getName() == "absTest") absTest(c);
        else if  ...  // other constraints implemented by MarhOps
        else throw new NotDefConstraintException();
    }

    private void absTest(Constraint c) {
        IntLVar x = (IntLVar)c.getArg(1);
        IntLVar y = (IntLVar)c.getArg(2);
        if (!x.isBound()) {            // irreducible case
            c.notSolved();
            return;
        };
        if (x.getValue() >= 0)
            Solver.add(x.eq(y));                      // x = y
        else
            Solver.add(x.eq(new IntLVar(0).sub(y)));  // x = 0 - y
        return;
    }
    ...
}
```

A possible use of the new constraint `absTest` *is:*

```
MathOps mathOps = new MathOps(solver);
IntLVar x = new IntLVar("x",-3);
IntLVar y = new IntLVar("y");
solver.check(mathOps.absTest(x,y));
y.output();
```

whose execution generates the output:

```
y = 3
```

A user-defined constraint is set by default to "solved" whenever it is processed by the solver. "Solved" constraints possibly occurring in the store are simply ignored by the solver; e.g., they are not printed at all by method `showStore()`. There are cases, however, in which one would like to state that the constraint is still "unsolved". For instance, in the above example, if `x` is unbound then the constraint must be simply left unchanged in the constraint store, in order to be possibly taken into account by subsequent call to the solver. The following method of the class `Constraint` can be used for these purposes:

`void` **notSolved**`()`
: Sets the `solved` flag of this constraint to `false` (i.e., this constraint is "unsolved").

16.3 Exploiting nondeterminism

Implementation of user-defined constraints can exploit the nondeterministic facilities of JSetL. In particular the following three methods are provided to support nondeterminism handling:

`int` **getAlternative**`()` (in class `Constraint`)
: Returns an integer associated with the invocation constraint `c` that can be used to count nondeterministic alternatives within this constraint. Its initial value is 0. Each time the constraint `c` is re-considered due to backtracking, the value returned by `getAlternative()` is automatically incremented by 1.

`void` **fail**`()` (in class `Constraint`)
: Raises an exception `Fail`. This exception is handled by the `Solver` which will backtrack if there are open choice points or throw an exception `Failure` otherwise.

`void` **addChoicePoint**`(Constraint c)` (in class `Solver`)
: Adds a choice point to the alternative stack of the invocation solver. This allows the solver to backtrack and re-consider the constraint `c` if a failure occurs subsequently. Upon backtracking, the original constraint state is restored as before except for the alternative counter that is incremented by 1.

The following is a nondeterministic version of the new constraint `absTest(x,y)` shown in Example 26. In this case, if `x` is unbound, the constraint solving procedure opens two nondeterministic alternatives: one in which `x` is assumed to be non-negative, and another one in which `x` is assumed to be negative (the new version of the absolute value constraint is simply called `abs`).

Example 27 *(New constraint* `abs`*)*

```
private void abs(Constraint c) {
    IntLVar x = (IntLVar)c.getArg(1);
    IntLVar y = (IntLVar)c.getArg(2);
    switch (c.getAlternative()) {
    case 0:            // x >= 0 and x = y
        Solver.addChoicePoint(c);
        Solver.add(x.ge(0).and(x.eq(y)));
        break;
    case 1:            // x < 0 and x = 0 - y
```

```
            Solver.add(x.lt(0).and(x.eq(new IntLVar(0).sub(y))));
        }
        return;
    }
```

A sample usage of the new constraint `abs` *is:*

```
IntLVar x = new IntLVar("x");
IntLVar y = new IntLVar("y",3);
solver.check(mathOps.abs(x,y));
x.output();
solver.nextSolution();
x.output();
```

whose execution generates the output:

```
x = 3
x = -3
```

References

[1] M. Cristiá and G. Rossi. A Decision Procedure for Sets, Binary Relations and Partial Functions. in *Computer Aided Verification - 28th International Conference (CAV 2016)*, Lecture Notes in Computer Science, Vol. 9779, Springer, 179–198, 2016.

[2] A. Dal Palù, A. Dovier, E. Pontelli, and G. Rossi. Integrating Finite Domain Constraints and CLP with Sets. In *PPDP'03 — Proc. of the Fifth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, 219–229, 2003.

[3] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.

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

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

[6] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19–20*, 503–581, 1994.

A Data structures for finite domain modeling

The following classes represent three different finite domains: intervals, multi-intervals and set-intervals. Their primary purpose is to model the domain of integer and integer set logical variables (see Sections 11 and 12).

A.1 The class `Interval`

Given two integers $a, b \in \mathbb{Z}$, the *(integer) interval* bounded by a and b is the set of integers:

$$[a..b] \stackrel{def}{=} \{x \in \mathbb{Z} : a \leq x \leq b\}$$

a is the GLB (*Greatest Lower Bound*) while b is the LUB (*Least Upper Bound*) of the interval.

Fixed an integer constant $\alpha \geq 0$, we first define an *universe* $\mathbb{Z}_\alpha \stackrel{def}{=} [-\alpha..\alpha]$ and then the set $\mathbb{I}_\alpha$ of all the intervals contained in $\mathbb{Z}_\alpha$, that is:

$$\mathbb{I}_\alpha \stackrel{def}{=} \{[a..b] : a, b \in \mathbb{Z}_\alpha\}$$

The class `Interval` allows to represent and manipulate the intervals $[a..b] \in \mathbb{I}_\alpha$. First of all, note that $\mathbb{Z}_\alpha$ is defined by:

- $\alpha =$ `Interval.`*`SUP`* $\stackrel{def}{=}$ `Integer.`*`MAX_VALUE`* `/ 2` $= 1073741823$
- $-\alpha =$ `Interval.`*`INF`* $\stackrel{def}{=}$ `-Interval.`*`SUP`* $= -1073741823$.

In addition to static fields `INF` and `SUP`, this class also provides the static method `universe()` which returns an `Interval` corresponding to the universe $\mathbb{Z}_\alpha$.

Constructors

Before introducing class constructors, it is worth noting that intervals defined in this way have two main restrictions. Indeed, they are:

- *finite*: $(\forall\ I \in \mathbb{I}_\alpha)(\forall\ x \in I)\ -\alpha \leq x \leq \alpha$
- *convex*: $(\forall\ I \in \mathbb{I}_\alpha)(\forall\ x, y \in I)\ [x..y] \subseteq I$.

Therefore, to overcome these limitations, two special operations are needed in order to represent generic integer sets as intervals belonging to $\mathbb{I}_\alpha$:

- *normalization*: is an operation $\|\cdot\|_\alpha : \mathcal{P}(\mathbb{Z}) \longrightarrow \mathcal{P}(\mathbb{Z}_\alpha)$ such that, for each $A \subseteq \mathbb{Z}$, $\|A\|_\alpha \stackrel{def}{=} A \cap \mathbb{Z}_\alpha$.
- *convex closure*: is an operation $\mathcal{CH}_\alpha : \mathcal{P}(\mathbb{Z}) \longrightarrow \mathbb{I}_\alpha$ such that, for each $A \subseteq \mathbb{Z}$, $\mathcal{CH}_\alpha(A) \stackrel{def}{=} min_\subseteq\{I \in \mathbb{I}_\alpha : \|A\|_\alpha \subseteq I\}$.

Hence, the class constructors are defined as follows:

`Interval()`
: Creates the empty interval $\varnothing$.

`Interval(Integer a)`
: Creates the interval $\|\{\texttt{a}\}\|_\alpha$.

`Interval(Integer a, Integer b)`
: Creates the interval $\|[\texttt{a..b}]\|_\alpha$.

`Interval(Set<Integer> s)`
: Creates the interval $\mathcal{CH}_\alpha(\texttt{s})$.

Example 28 *(Interval constructors)*

- *Create an empty interval, since* $\|\{\texttt{Interval.SUP} + 1\}\|_\alpha = \{\alpha + 1\} \cap \mathbb{Z}_\alpha = \varnothing$

```
Interval i = new Interval(Interval.SUP + 1);
```

- *Create the interval* $\|[\texttt{Interval.INF} - 2..0]\|_\alpha = [-\alpha - 2..0] \cap \mathbb{Z}_\alpha = [-\alpha..0]$

```
Interval i = new Interval(Interval.INF - 2, 0);
```

- *Create the interval* $\mathcal{CH}_\alpha(\{-3, 1, 0, 5\}) = [-3..5]$

```
HashSet<Integer> set = new HashSet<Integer>();
set.add(-3);
set.add(1);
set.add(0);
set.add(5);
Interval i = new Interval(set);
```

Set Operations

Since intervals are sets of integers, it is possible to define set operations on them. However, note that only those operations which do not *over-approximate* the result have a public interface.

`boolean` **subset**`(Interval I)`
Returns `true` iff `this` $\subseteq$ `I`.

`Interval` **intersect**`(Interval I)`
Returns the interval corresponding to `this` $\cap$ `I`.

`Interval` **sum**`(Interval I)`
Returns the interval corresponding to `this` $\oplus$ `I`, where in general:

$$[a..b] \oplus [c..d] \stackrel{def}{=} \|[a + c..b + d]\|_\alpha$$

`Interval` **sub**`(Interval I)`
Returns the interval corresponding to `this` $\ominus$ `I`, where in general:

$$[a..b] \ominus [c..d] \stackrel{def}{=} \|[a - d..b - c]\|_\alpha$$

`Interval` **opposite**`()`
Returns the interval corresponding to $\ominus$`this` $\stackrel{def}{=} \{0\} \ominus$ `this`.

Other utility methods

`boolean` **contains**`(Integer k)`
Returns `true` iff `k` $\in$ `this`.

`boolean` **isEmpty**`()`
Returns `true` iff `this` $= \varnothing$.

`boolean` **isSingleton**`()`
Returns `true` iff $|$`this`$| = 1$.

`boolean` **isUniverse**`()`
Returns `true` iff `this` $= \mathbb{Z}_\alpha$.

`int` **size**`()`
Returns |`this`|.

`Integer` **getGlb**`()`
Returns the GLB of `this` if `this` $\neq \varnothing$, `null` otherwise.

`Integer` **getLub**`()`
Returns the LUB of `this` if `this` $\neq \varnothing$, `null` otherwise.

`TreeSet<Integer>` **toSet**`()`
Returns a `java.util.TreeSet` containing all the elements of `this`.

`Iterator<Integer>` **iterator**`()`
Returns an iterator over the elements of `this`, in ascending order.

`Interval` **clone**`()`
Returns a copy of `this`.

`Interval` **equals**`(Object obj)`
Returns `true` iff `this` is equals to `obj`.

`String` **toString**`()`
Returns a string representation of `this`.

A.2 The class `MultiInterval`

A *multi-interval (of integers)* is a set of integers $M \subset \mathbb{Z}$ defined by $n \geq 0$ intervals $I_1, I_2, \ldots, I_n \in \mathbb{I}_\alpha \setminus \varnothing$ such that:

(i) $M = I_1 \cup I_2 \cup \ldots \cup I_n$

(ii) $I_1 \prec I_2 \prec \ldots \prec I_n$

where $[a..b] \prec [c..d] \stackrel{def}{\iff} b < c - 1$.

The set of all the multi-intervals $M \subseteq \mathbb{Z}_\alpha$ will be named $\mathbb{M}_\alpha$.

Example 29 *Examples of multi-intervals are:*

- $M = \varnothing$
- $M = [1..10]$
- $M = [-3..0] \cup [5..5] \cup [15..30]$

For multi-intervals defined by $n > 1$ intervals we will use a simpler notation, where intervals are simply listed in curly brackets and singleton intervals of the form $[k..k]$ are replaced by k. For example, the last multi-interval of the above example can be written as $\{-3..0, 5, 15..30\}$.

It is important to observe that $\mathbb{M}_\alpha = \mathcal{P}(\mathbb{Z}_\alpha)$; in other terms, every subset of $\mathbb{Z}_\alpha$ is *uniquely identified* by a multi-interval in $\mathbb{M}_\alpha$ (and viceversa).

The class `MultiInterval` allows to represent and manipulate all the multi-intervals $M \in \mathbb{M}_\alpha$.

Note that, although an interval is a particular case of multi-interval (is trivial to prove that $\mathbb{I}_\alpha \subset \mathbb{M}_\alpha$), `MultiInterval` *is not a super-class* of `Interval`.

Moreover, this class implements the Java *interface* `Set<Integer>`: for this reason, all the methods of `Set` (and its *super-interfaces* `Collection` and `Iterable`) must be implemented (for more details, see Java APIs specification).

Finally, like class `Interval`, `MultiInterval` has static fields `INF` and `SUP`, which represent $-\alpha$ and α respectively, and a static method `universe()`, which returns the universe $\mathbb{Z}_\alpha$.

Constructors

`MultiInterval()`
: Creates the empty multi-interval $\varnothing$.

`MultiInterval(Integer a)`
: Creates the multi-interval $\|\{\texttt{a}\}\|_\alpha$.

`MultiInterval(Integer a, Integer b)`
: Creates the multi-interval $\|[\texttt{a}..\texttt{b}]\|_\alpha$.

`MultiInterval(Set<Integer> s)`
: Creates the multi-interval corresponding to $\|\texttt{s}\|_\alpha$.

`MultiInterval(Collection<Interval> I)`
: Creates the multi-interval corresponding to $I_1 \cup \ldots \cup I_n$, if `I` is an interval collection of the form $[I_1, \ldots, I_n]$.

Example 30 *(Multi-interval constructors)*

- *Create the multi-interval* $\|\{10, 5, 8, \alpha + 1, -1, 9, 0\}\|_\alpha = \{-1..0, 5, 8..10\}$

```
TreeSet<Integer> set = new TreeSet<Integer>();
set.add(10);
set.add(5);
set.add(8);
set.add(MultiInterval.SUP + 1);
set.add(-1);
set.add(9);
set.add(0);
MultiInterval m = new MultiInterval(set);
```

- *Create the multi-interval* $[-2..4] \cup [3..5] \cup \varnothing \cup [10..20] = \{-2..5, 10..20\}$

```
Vector<Interval> v = new Vector<Interval>();
v.add(new Interval(-2, 4));
v.add(new Interval(3, 5));
v.add(new Interval());
v.add(new Interval(10, 20));
MultiInterval m = new MultiInterval(v);
```

Set Operations

Since multi-intervals are all and only the subsets of $\mathbb{Z}_\alpha$, it is possible to define on them the same set operations applicable to $\mathbb{Z}_\alpha$.

`boolean` **subset**`(MultiInterval M)`
Returns `true` iff `this` $\subseteq$ `M`.

`MultiInterval` **complement**`()`
Returns the set complement of `this` with respect to the universe $\mathbb{Z}_\alpha$.

`MultiInterval` **complement**`(MultiInterval U)`
Returns the set complement of `this` with respect to the universe `U`.

`MultiInterval` **union**`(MultiInterval M)`
Returns `this` $\cup$ `M`.

`MultiInterval` **intersect**`(MultiInterval M)`
Returns `this` $\cap$ `M`.

`MultiInterval` **diff**`(MultiInterval M)`
Returns `this` $\setminus$ `M`.

`MultiInterval` **sum**`(MultiInterval M)`
Returns `this` $\boxplus$ `M`, where in general:

$$A \boxplus B \stackrel{def}{=} \|\{c \in \mathbb{Z} : (\exists a \in A)(\exists b \in B)\ c = a + b\}\|_\alpha .$$

`MultiInterval` **sub**`(MultiInterval M)`
Returns `this` $\boxminus$ `M`, where in general:

$$A \boxminus B \stackrel{def}{=} \|\{c \in \mathbb{Z} : (\exists a \in A)(\exists b \in B)\ c = a - b\}\|_\alpha .$$

`MultiInterval` **opposite**`()`
Returns $\boxminus$`this` $\stackrel{def}{=}$ $\{0\} \boxminus$ `M`.

Other utility methods

In addition to the utility methods seen for the class `Interval` and the methods inherited from `java.util.Set`, `MultiInterval` offers the following methods:

`Interval` **convexClosure**`()`
Returns the convex closure $\mathcal{CH}_\alpha$(`this`).

`int` **getOrder**`()`
Returns the *order* of `this`, i.e. the number of disjoint intervals which define it.

A.3 The class `SetInterval`

Given two sets of integers $A, B \subseteq \mathbb{Z}$, the *(integer) set-interval* bounded by A and B is the set of integer sets (more precisely, the *lattice* of integer sets):

$$[A..B] \stackrel{def}{=} \{X \subseteq \mathbb{Z} : A \subseteq X \subseteq B\}$$

A is the GLB (*Greatest Lower Bound*) while B is the LUB (*Least Upper Bound*) of the set-interval.

Similarly to what done for integer intervals, we fix an integer constant $\beta \geq 0$ and define an *universe* $\mathbb{Z}_\beta \stackrel{def}{=} [-\beta..\beta]$. The set $\mathbb{S}_\beta$ of all the set-intervals whose bounds are subsets of $\mathbb{Z}_\beta$ is:

$$\mathbb{S}_\beta \stackrel{def}{=} \{[A..B] : A, B \subseteq \mathbb{Z}_\beta\}$$

The class `SetInterval` allows to represent and manipulate the set-intervals $[A..B] \in \mathbb{S}_\beta$. First of all, it is worth noting that the integer sets belonging to set-intervals are modelled by the class `MultiInterval`. This is not surprising: as seen in the previous section, multi-intervals and sets of integers belonging to a fixed universe are in bijective correspondence.

Moreover, as for integer intervals, observe that set-intervals are:

- *finite*, even if they contain an exponential number of elements: $|[A..B]| = 2^{|B|-|A|}$. Thus, a *normalization* operator $\|\cdot\|_\beta : \mathcal{P}^2(\mathbb{Z}) \longrightarrow \mathcal{P}^2(\mathbb{Z}_\beta)$ such that:

  $$\|D\|_\beta \stackrel{def}{=} D \cap \mathcal{P}(\mathbb{Z}_\beta)$$

  is needed.

- *convex*: a *convex closure* operator $\mathcal{CH}_\beta : \mathcal{P}^2(\mathbb{Z}) \longrightarrow \mathbb{S}_\beta$ such that:

  $$\mathcal{CH}_\beta(D) \stackrel{def}{=} min_\subseteq\{S \in \mathbb{S}_\beta : \|D\|_\beta \subseteq S\}$$

  is needed.

Since for each set-interval $[A..B] \in \mathbb{S}_\beta$ we have that $\varnothing \subseteq A$ and $B \subseteq \mathbb{Z}_\beta = [-\beta..\beta]$, `SetInterval` class will have two static fields:

- `public static final MultiInterval INF = new MultiInterval();`
- `public static final MultiInterval SUP =`
  `new MultiInterval(-Interval.SUP / 2, Interval.SUP / 2);`

which represent, respectively, the minimum and the maximum value (according to the partial order $\subseteq$) that a set-interval element can take.

Obviously, `INF` is the empty set. Instead, `SUP` corresponds to $\mathbb{Z}_\beta$: it means that the fixed value of β is `Interval.SUP / 2`. Moreover, `SetInterval` provides the static method `universe()` which returns the universe $[\varnothing..\mathbb{Z}_\beta]$.

Constructors

`SetInterval()`
: Creates the empty set-interval $\varnothing$.

`SetInterval(MultiInterval A)`
: Creates the set-interval $\|\{\texttt{A}\}\|_\beta$.

`SetInterval(MultiInterval A, MultiInterval B)`
: Creates the interval $\|[\texttt{A}..\texttt{B}]\|_\beta$.

`SetInterval(Collection<MultiInterval> D)`
: Creates the interval $\mathcal{CH}_\beta(\texttt{D})$.

Example 31 *(Set-interval constructors)*

- *Create an empty set-interval, since* $\|\{[2..\beta+1]\}\|_\beta = \{[2..\beta+1]\} \cap \mathcal{P}(\mathbb{Z}_\beta) = \varnothing$

```
MultiInterval m = new MultiInterval(2, SetInterval.SUP.getLub() + 1);
SetInterval s   = new SetInterval(m);
```

- *Create the set-interval* $\mathcal{CH}_\beta(\{\{0\},\{1\},[2..\beta+1]\}) = min_\subseteq\{S \in \mathbb{S}_\beta : \{\{0\},\{1\}\} \subseteq S\} = [\varnothing..\{0,1\}]$

```
MultiInterval m  = new MultiInterval(2, SetInterval.SUP.getLub() + 1);
MultiInterval m0 = new MultiInterval(0);
MultiInterval m1 = new MultiInterval(1);
Vector<MultiInterval> v = new Vector<MultiInterval>();
v.add(m);
v.add(m0);
v.add(m1);
SetInterval s = new SetInterval(v);
```

Other utility methods

`boolean` **contains**`(MultiInterval M)`
Returns `true` iff `M` $\in$ `this`.

`boolean` **isEmpty**`()`
Returns `true` iff `this` $= \varnothing$.

`boolean` **isSingleton**`()`
Returns `true` iff $|$`this`$| = 1$.

`boolean` **isUniverse**`()`
Returns `true` iff `this` $= [\varnothing..\mathbb{Z}_\beta]$.

`double` **size**`()`
Let A and B be the GLB and LUB of `this`, respectively. This method returns $|$`this`$| = 2^{|A|-|B|}$ iff $|B| - |A| \leq$ `Double.MAX_EXPONENT` $= 1023$; otherwise, it returns `Double.POSITIVE_INFINITY`.

`MultiInterval` **getGlb**`()`
Returns the GLB of `this` if `this` $\neq \varnothing$, `null` otherwise.

`MultiInterval` **getLub**`()`
Returns the LUB of `this` if `this` $\neq \varnothing$, `null` otherwise.

`SetInterval` **intersect**`(SetInterval S)`
Returns the set-interval `this` $\cap$ `S`.

`SetInterval` **clone**`()`
Returns a copy of `this`.

`SetInterval` **equals**`(Object obj)`
Returns `true` iff `this` is equals to `obj`.

`String` **toString**`()`
Returns a string representation of `this`.

Particular attention should be paid to method `size()`. Indeed, since a set-interval may contain an exponential number of elements, the Java scalar type `double` is used to represent its size. However, note that if a set-interval is too big (e.g., the universe $[\varnothing..\mathbb{Z}_\beta]$) the constant value `Double.POSITIVE_INFINITY` is returned.

B Data structures for dealing with labeling

As is usually the case with finite domain constraint solvers, the JSetL constraint solver is *not complete*. Specifically, if the constraint store contains constraints over `IntLVar` and `SetLVar` objects, the solver does not ensure that all the constraints belonging to it are satisfiable. In order to check satisfiability and find one (or all) possible solution(s), suitable research strategies are therefore needed: *labeling* is one of these.

Given $n \geq 0$ logical variables $v_1, \ldots, v_n$, *labeling* them means trying to assign to each variable a value belonging to its domain.

Obviously, considering every possible labeling of all the variables of the constraint store, we obtain completeness (i.e, every possible value assignment to the variables is computed). However, the excessive simplicity of this method implies a not reasonable computational complexity. For this reason, labeling is improved by special *heuristics* which allow to reduce the search space. Specifically:

- *Variable Choice Heuristics*: determine the order in which variables are selected for assignment;
- *Value Choice Heuristics*: determine the order in which domain values are assigned to a selected variable.

The next subsections will describe techniques and data structures for dealing with labeling on `IntLVar` and `SetLVar` objects in JSetL.

B.1 Labeling on integer logical variables

JSetL provides three data structures for modeling choice heuristics: the enumerations `VarHeuristic` and `ValHeuristic`, and the class `LabelingOptions`.

The enumeration `VarHeuristic`

`VarHeuristic` is a Java enumeration that implements the possible *variable choice heuristics* for a given collection $x_1, \ldots, x_n$ of `IntLVar`'s. Such `enum` consists of the following fields:

`LEFT_MOST`
: Selects the leftmost variable x_1.

`RIGHT_MOST`
: Selects the rightmost variable x_n.

`MID_MOST`
: Selects the midmost variable x_k, where $k = \left\lfloor \frac{n}{2} \right\rfloor$.

`MIN`
: Selects the leftmost variable with the smallest GLB.

`MAX`
: Selects the leftmost variable with the greatest LUB.

`FIRST_FAIL`
Selects the leftmost variable with the smallest domain.

`RANDOM`
Selects a variable x_k, where k is a pseudorandom equidistributed value in $\{1, \ldots, n\}$.

Example 32 *Let us consider three integer logical variables x, y and z with associated domains $D_x = \{1..10, 28..30\}$, $D_y = \{1..50, 100, 1000\}$ and $D_z = [0..40]$, respectively. Let us see how the variable choice heuristics work:*

- `LEFT_MOST`*: selects the variable x*
- `MID_MOST`*: selects the variable y*
- `RIGHT_MOST`*: selects the variable z*
- `MIN`*: selects the variable z*
- `MAX`*: selects the variable y*
- `FIRST_FAIL`*: selects the variable x*
- `RANDOM`*: selects a variable $v \in \{x, y, z\}$ such that $\mathbb{P}[v = x] = \mathbb{P}[v = y] = \mathbb{P}[v = z] = \frac{1}{3}$.*[4]

The enumeration `ValHeuristic`

`ValHeuristic` is a Java enumeration that implements the possible *value choice heuristics* for a selected `IntLVar` x with domain the multi-interval $D_x = I_1 \cup \ldots \cup I_n$. Such `enum` consists of the following fields:

`GLB`
Selects the GLB of D_x.

`LUB`
Selects the LUB of D_x.

`MID_MOST`
Selects the middle point $\left\lfloor \frac{I_k^- + I_k^+}{2} \right\rfloor$ of the 'central' interval I_k, where $k = \left\lfloor \frac{n}{2} \right\rfloor$.

`MEDIAN`
Selects the median value of D_x (note that if $|D_x|$ is even, the minimum between the two median values of D_x will be selected).

`EQUI_RANDOM`
Selects a pseudorandom equidistributed value in D_x.

`RANGE_RANDOM`
Selects a pseudorandom equidistributed value in I_k, where k is a pseudorandom equidistributed value in $\{1, \ldots, n\}$.

`MID_RANDOM`
Selects the midpoint of an interval I_k, where k is a pseudorandom equidistributed value in $\{1, \ldots, n\}$.

[4] The notation $\mathbb{P}[E]$ indicates the probability that a given event E occurs.

Note that, unlike `EQUI_RANDOM`, `RANGE_RANDOM` does not select an equidistributed value in D_x: the probability that a value $d \in D_x$ will be chosen is inversely proportional to the size of the interval I_k to which d belongs. However, if D_x is an interval then `EQUI_RANDOM` and `RANGE_RANDOM` are in fact the same heuristic.

`MID_RANDOM` is an hybrid solution: first a random interval I_k is chosen and then the midpoint of I_k is selected. Thus, each midpoint has a probability $1/n$ to be selected. Note that if D_x is an interval then `MID_RANDOM`, `MID_MOST` and `MEDIAN` are in fact the same heuristic.

Example 33 *Let us consider again the integer logical variables x, y and z with associated domains $D_x = \{1..10, 28..30\}$, $D_y = \{1..50, 100, 1000\}$ and $D_z = [0..40]$ of Example 32. Let us see now how the value choice heuristics work on them, indicating with $\lambda(v)$ the selected value for each variable $v \in \{x, y, z\}$.*

`GLB` :	$\lambda(x) = 1$	$\lambda(y) = 1$	$\lambda(z) = 0$
`LUB` :	$\lambda(x) = 30$	$\lambda(y) = 1000$	$\lambda(z) = 40$
`MID_MOST` :	$\lambda(x) = 5$	$\lambda(y) = 100$	$\lambda(z) = 20$
`MEDIAN` :	$\lambda(x) = 7$	$\lambda(y) = 26$	$\lambda(z) = 20$

Moreover, for each $a \in D_x$, $b \in D_y$ e $c \in D_z$ we have that:

$$\texttt{EQUI\_RANDOM}: \quad \mathbb{P}[\lambda(x) = a] = \frac{1}{13}$$
$$\mathbb{P}[\lambda(y) = b] = \frac{1}{52}$$
$$\mathbb{P}[\lambda(z) = c] = \frac{1}{41}$$

$$\texttt{RANGE\_RANDOM}: \quad \mathbb{P}[\lambda(x) = a] = \begin{cases} \dfrac{1}{20}, & \textit{se } a \in [1..10] \\ \dfrac{1}{6} & \textit{se } a \in [28..30] \end{cases}$$

$$\mathbb{P}[\lambda(y) = b] = \begin{cases} \dfrac{1}{150}, & \textit{se } b \in [1..50] \\ \dfrac{1}{3}, & \textit{se } b = 100 \\ \dfrac{1}{3} & \textit{se } b = 1000 \end{cases}$$

$$\mathbb{P}[\lambda(z) = c] = \frac{1}{41}$$

$$\texttt{MID\_RANDOM}: \quad \mathbb{P}[\lambda(x) = a] = \begin{cases} \dfrac{1}{2}, & \textit{if } a \in \{5, 29\} \\ 0 & \textit{otherwise} \end{cases}$$

$$\mathbb{P}[\lambda(y) = b] = \begin{cases} \dfrac{1}{3}, & \textit{if } b \in \{25, 100, 1000\} \\ 0 & \textit{otherwise} \end{cases}$$

$$\lambda(z) = 20$$

The class `LabelingOptions`

The class `LabelingOptions` allows the user to set up the labeling heuristics by properly setting its public fields, which are:

`VarHeuristic` **var**
: The variable choice heuristic.

`ValHeuristic` **val**
: The value choice heuristic.

`SetHeuristic` **set**
: For labeling on `SetLVar`'s (see Section B.2).

This class provides only one constructor `LabelingOptions()` that initializes such fields to their default values, which are:

- `var = LEFT_MOST`
- `val = GLB`
- `set = FIRST_NIN`

Example 34 *The statements*

```
LabelingOptions lop = new LabelingOptions();
lop.var = VarHeuristic.FIRST_FAIL;
lop.val = ValHeuristic.MEDIAN;
```

set the variable choice heuristic to `FIRST_FAIL` *and the value choice heuristic to* `MEDIAN`*.*

Note that, since the field `set` *is not modified,* `lop.set` *will retain the default value* `SetHeuristic.FIRST_NIN`*.*

Objects of type `LabelingOptions` are used as parameters for labeling constraint methods (see Sections 11.4 and 12.4).

B.2 Labeling on integer set logical variables

In addition to the data structures presented in the previous section, JSetL provides the enumeration `SetHeuristic`.

Before introducing such enumeration, it is important to note that labeling on set variables is quite different from labeling on integer variables. Indeed, while for integer logical variables each labeled variable x is directly instantiated with a value k belonging to its domain, for set variables the same approach turns out to be impracticable. This is because each set variable X with domain $[A..B]$ could be instantiated by an exponential number of elements (precisely $2^{|B|-|A|}$) belonging to its domain.

Thus, given n set variables $X_1, \ldots, X_n$ to be labeled, the following approach is used:

- a variable $X \in \{X_1, \ldots, X_n\}$ is selected according to a certain *variable choice heuristic*
- an integer value $k \in B \setminus A$, where $[A..B]$ is the domain of X, is selected according to a certain *value choice heuristic*. Note that the integer set $B \setminus A$ corresponds to all the integer values that *could* belong to X (but they do not necessarily belong to it)
- the (meta) constraint $k \in X \vee k \notin X$ is added to the store and solved. Note that such logic disjunction is *nondeterministic*: thus, a choice must be made about which constraint will be solved first, depending on the value of a certain *set choice heuristic*.

In this way, the domain of a set variable is refined (by adding values to its GLB or removing values from its LUB) until the variable results (possibly) bound.

The enumeration `SetHeuristic`

Variable and value choice heuristics are modelled by using the enumerations `VarHeuristic` and `ValHeuristic`, respectively (see Section B.1). To decide which (non-)membership constraint will be solved first, instead, the enumeration `SetHeuristic` is used. Such `enum` consists of the following fields:

`FIRST_IN`
: The membership constraint $k \in X$ is solved first.

`FIRST_NIN`
: The non-membership constraint $k \notin X$ is solved fist.

As for the integer logical variables, the class `LabelingOptions` allows the labeling heuristics to be setted up by properly setting its public fields. Remind that the default value for the `set` field in class `LabelingOptions` is `SetHeuristic.FIRST_NIN`.

Example 35 *The statements*

```
LabelingOptions lop = new LabelingOptions();
lop.var = VarHeuristic.RANDOM;
lop.val = ValHeuristic.LUB;
lop.set = SetHeuristic.FIRST_IN;
```

set the variable choice heuristic to `RANDOM`*, the value choice heuristic to* `LUB`*, and the set choice heuristic to* `FIRST_IN`*.*