JSetL User’s Manual
Version 2.3

GIANFRANCO ROSSI* AND ROBERTO AMADINI
Dipartimento di Matematica
Università di Parma
Parma (Italy)

Abstract

This manual describes JSetL 2.3, 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 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://cmt.math.unipr.it/jsetl.html.

*Correspondence to: Gianfranco Rossi, Dipartimento di Matematica, Università degli Studi di Parma, Parco Area delle Scienze, 53/A, I-43124 Parma, Italy. E-mail address: gianfranco.rossi@unipr.it
Contents

1 Introduction 1

2 Logical variables: the class LVar 2
  2.1 Constructors ........................................... 2
  2.2 General utility methods .................................. 3
  2.3 Constraint methods ..................................... 4

3 Logical lists: the class LList 5
  3.1 Constructors ........................................... 5
  3.2 Creating new (bound) l-lists .............................. 6
  3.3 General utility methods .................................. 7
    3.3.1 Basic methods ..................................... 8
    3.3.2 Logical collection methods ......................... 8
  3.4 Constraint methods ..................................... 9

4 Logical sets: the class LSet 10
  4.1 Constructors ........................................... 11
  4.2 Creating new (bound) l-sets .............................. 11
  4.3 General utility methods .................................. 12
    4.3.1 Basic methods ..................................... 12
    4.3.2 Logical collection methods ......................... 12
  4.4 Constraint methods ..................................... 13

5 Integer logical variables: the class IntLVar 15
  5.1 Constructors ........................................... 16
  5.2 General utility methods .................................. 17
  5.3 Integer logical expressions ............................. 17
  5.4 Constraint methods ..................................... 18

6 Integer set logical variables: the class SetLVar 20
  6.1 Constructors ........................................... 21
  6.2 General utility methods .................................. 23
  6.3 Integer set expressions ................................. 23
  6.4 Constraint methods ..................................... 24

7 Constraints: the class Constraint 27
  7.1 Constructors ........................................... 28
  7.2 General utility methods .................................. 28
  7.3 Meta-constraints ....................................... 29
  7.4 Constraint methods in other classes ..................... 29

8 Constraint solving: the class SolverClass 29
  8.1 Posting and inspecting constraints ....................... 30
  8.2 Checking constraint satisfiability ....................... 31
  8.3 Getting solutions ...................................... 31
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(SET) [1].

Unification may involve logical variables, as well as list and set objects (“set unification”). 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(SET) [1] 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.. JSetL provides also a simple way (the method setof) to collect into a set all the computed solutions of a given constraint for a specified logical variable.

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://cmt.math.unipr.it/jsetl.html.

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. From that moment on, several students, under the supervision of Gianfranco Rossi, have helped 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. More recently, Federico Bergenti and Luca Chiarabini
joined the JSetL Team at the Department of Mathematics. In the last year, Roberto Ama-
dini was deeply involved in the design and implementation of part of the JSetL constraint
solver. Currently, the students Andrea Longo and Fabio Biselli are still working on JSetL
as part of their “Laurea” Thesis. We thank very much all of them.

2 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 relations (or 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 relation, in particular, allows a precise value to be associated to a logical
variable. For example, if $x$ is a logical variable ranging over the domain of integers, the
equality $x = 3$ forces $x$ to be bound to the value 3. However, the same result can be
obtained through other relations, e.g., $x < 4 \land x > 2$.

The value of a logical variable is immutable. That is it can not be changed, e.g. by an
assignment statement.

A logical variable may have also an external name associated with. 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 con-
structors 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).

The class LVar has a number of subclasses that deal with specific values of specific types.
A primary distinction is between atomic and structured values. The variables which can
assume atomic values are the integer logical variables (class IntLVar, see Section 5) and the
integer set logical variables (class SetLVar, see Section 6) whose values are integer numbers
and set of integers respectively. Conversely, the variables which can assume structured values
are the set logical variables and list logical variables (classes LSet and LList, see Sections 4
and 3), whose values are, respectively, possibly partially specified sets and lists of elements
of any type.

2.1 Constructors

LVar()
LVar(String extName)
    Create an unbound logical variable, with no external name (resp., with external name
    extName). If extName is omitted, a default external name "?" is assigned to the
    variable automatically.

LVar(Object o)
LVar(String extName, Object o)
    Create a (bound) logical variable, with no external name (resp., with external name
    extName), with value o. o can be of any type but a String in the case of the one
parameter constructor (in fact, in a such case, the `LVar(String extName)` constructor would be called). Same as to create an unbound logical variable \( x \) and to post and solve the constraint \( x \text{.eq}(o) \).

\[
\text{LVar(LVar lv)}
\]
\[
\text{LVar(String extName, LVar lv)}
\]

Create a logical variable, with no external name (resp., with external name \texttt{extName}), equivalent to the logical variable \( lv \). Same as to create an unbound logical variable \( x \) and to post and solve the constraint \( x \text{.eq}(lv) \).

We say that two logical variables \( x \) and \( y \) are \textit{equivalent} if they have been (successfully) unified, e.g., by \( x \text{.eq}(y) \). 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 \( x \) (without any external name).
  \[
  \text{LVar x = new LVar();}
  \]
- Create an unbound logical variable \( y \) with external name "y".
  \[
  \text{LVar y = new LVar("y");}
  \]
- Create a logical variable \( z \), with external name "z", bound to the integer value 2.
  \[
  \text{LVar z = new LVar("z", 2);}
  \]
- Create a logical variable \( v \) equivalent to the logical variable \( x \).
  \[
  \text{LVar v = new LVar(x);} \]

Note that, in the last definition, the logical variable \( v \) is unified with the logical variable \( x \). Both variables are unbound. If either \( x \) or \( v \) is subsequently bound to some value, both \( x \) and \( v \) become bound to this value.

### 2.2 General utility methods

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

\[
\text{LVar clone()}
\]

Creates and returns a copy of this logical variable.

\[
\text{boolean equals(LVar lv)}
\]

Returns \( \text{true} \) if this logical variable is equal to the logical variable \( lv \). Two logical variables are equal if 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.

\[
\text{boolean equals(Object o)}
\]

Returns \( \text{true} \) if this logical variable is equal to the object \( o \). A logical variable \( x \) is equal to an object \( o \) (other than a logical variable) if \( x \) is bound to a value equal to \( o \).

\[
\text{String getName()}
\]

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.

void output()
    Prints 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 (for reasons of convenience).

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(); // --> ? = unknown
        y.output(); // --> y = unknown
        z.output(); // --> z = 2

    • Convert to string
        x.toString(); // --> _?
        y.toString(); // --> _y
        z.toString(); // --> 2

2.3 Constraint methods

Constraints (see Sect. 7) can be posted over logical variables. In particular, the class LVar
provides methods for generating equality, inequality, membership and not membership con-
straints.

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 ∈ ls (resp., this ∈ s), that requires this logical variable to be a member of the logical set ls (resp., of the Java generic set s) (see Sect. 4 for the notion of logical set). 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. Note that if this logical variable is unbound, solving the constraint will nondeterministically bind this variable to each value in ls (resp., in s).

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

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

3 Logical lists: the class LList

A logical list l (or, simply, a l-list l) is a special kind of logical variable whose value is a pair (elems, rest), where elems (the list value) is a list \([e_0, \ldots, e_n]\), \(n \geq 0\), of objects of arbitrary types, and rest is either an empty or an unbound l-list representing the remainder of l. When rest 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; conversely, when rest is the empty l-list, we say that l represents a closed list and we use the (abstract) notation \([e_0, \ldots, e_n]\). When elems is the empty list and rest is null, l is the empty l-list and we use \([\] \] to denote it.

A l-list that contains unbound logical objects (i.e., either variables or lists or sets—see Sect. 4) represents a partially specified list. Intuitively, some of the list elements are unknown. Also an open l-list has an unknown part, and hence it also represents a partially specified list (regardless of whether its elements are bound or not).

In JSetL, a logical list is defined as an instance of the class LList, which extends the class LCollection. In particular, list values (i.e., the elems part of the l-list) are instances 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 variables 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.

3.1 Constructors

LList()
LList(String extName)

Create an unbound l-list, with no external name (resp., with external name extName).

LList(List<?> l)
LList(String extName, List<?> l)

Create a (bound) l-list, with no external name (resp., with external name extName), with 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)

Create a l-list, with no external name (resp., with external name extName), equivalent
to the l-list \( b \). Same as to create an unbound l-list \( x \) and to post and solve the constraint \( x . e q ( b ) \).

**Example 3**

- Create an unbound l-list \( a \).
  
  ```java
  LList a = new LList();
  ```

- Create a l-list \( b \), with external name "b", bound to the list value \([1,2,3]\).
  
  ```java
  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 \( a \).
  
  ```java
  LList c = new LList(a);
  ```

### 3.2 Creating new (bound) l-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 the list bound to this variable. 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 rest part (i.e., \([o | l]\), where \( l \) is this l-list).

**LList insn(Object o)**

Like `ins`, but \( o \) is added as the last element of the list bound to this variable.

**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 \( c \) as the head elements of the list bound to this l-list, respecting the order they have in \( c \). Otherwise, returns the (new) l-list whose value is the list containing all elements in \( c \) and this l-list as its rest part (i.e., \([a_1, \ldots, a_n | l]\), where \( a_1, \ldots, a_n \) are the elements of \( c \) and \( l \) is this l-list).

**static LList mkList(int n)**

**static LList mkList(String extName, int n)**

Returns a l-list, with no external name (resp., with external name \( \text{extName} \)), whose value is a list of \( n \) unbound logical variables.

**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, 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 \texttt{ins}, \texttt{insn}, and \texttt{insAll} methods are invoked can be either bound or unbound. In particular, invoking the \texttt{ins}, \texttt{insn}, and \texttt{insAll} methods on an unbound l-list allows an open list to be built.

In a list the order of elements and the repetitions do matter (whereas they do not matter in a set). Thus, for instance, the list \([1, 2]\) is different from the list \([2, 1]\) or from the list \([1, 2, 2]\).

\textbf{Example 4}

- Create a l-list \(d\), bound to the empty list.
  
  \begin{verbatim}
  LList d = LList.empty();
  \end{verbatim}

- Create a l-list \(e\), bound to the list value \(['c', 'b', 'a']\).
  
  \begin{verbatim}
  LList e = LList.empty().ins('a').ins('b').ins('c');
  \end{verbatim}

  or
  
  \begin{verbatim}
  char[] elems = {'a', 'b', 'c'};
  LList e = LList.empty().insAll(elems);
  \end{verbatim}

- Create a l-list \(f\), bound to the (partially specified) list value \([1, x]\), where \(x\) is an unbound logical variable.
  
  \begin{verbatim}
  LVar x = new LVar();
  LList f = LList.empty().insn(1).insn(x);
  \end{verbatim}

- Create a l-list \(g\), representing the open list \([1, 2|r]\), where \(r\) is an unbound l-list.
  
  \begin{verbatim}
  LVar z = new LVar(2);
  LList r = new LList();
  LList g = r.insn(1).insn(z);
  \end{verbatim}

  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).
  
  \begin{verbatim}
  LList e = LList.empty().ins(f).ins(e).ins(d);
  \end{verbatim}

  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 \([X_1, X_2, X_3]\).
  
  \begin{verbatim}
  LList i = LList.mkList(3);
  \end{verbatim}

\textbf{3.3 General utility methods}

The class \texttt{LList} provides the utility methods of the class \texttt{LVar} (see Section 2.2), along with a few other methods that take into account the fact that values possibly bound to \texttt{LList} objects are collections (namely, lists), possibly containing other logical variables and collections.
3.3.1 Basic methods

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

- LList clone()
- boolean equals(LList l)
- boolean equals(Object o)
- String getName()
- List<? extends LList> getRest()
- boolean isBound()
- void output()
- LList setName(String extName)
- String toString()

Remarks

- equals and getValue consider only the list value of the given l-list (i.e., they ignore its rest part). In particular, getValue returns the elems list of the given l-list. In order to obtain also its rest part one must use the getRest method (see 3.3.2).

- If the l-list denotes an open list, the string returned by 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 rest 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.

3.3.2 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 rest part. The first group of methods are specific of the class LList, while the second group is common to all collections, in particular to List objects.

Object get(int i)

If this l-list is bound, returns the i-th element of its list value. Otherwise, it raises an exception NotInitVarException.

LList getRest()

If this l-list is bound to a not empty l-list, returns its rest part, that is either an unbound l-list if this l-list is open, or the empty list if this l-list is closed. Returns null if this l-list is the empty l-list; the variable itself if this l-list is unbound.

boolean isClosed()

If this l-list is bound, returns true if it has an empty rest part, false otherwise (i.e., if it denotes an open list). Raises an exception NotInitVarException if this l-list is unbound.

boolean isEmpty()

If this l-list is bound, returns true if it is the empty l-list. Raises an exception NotInitVarException if this l-list is unbound.

\footnote{All exceptions defined in JSetIL are extensions of the class java.lang.Exception}
boolean isGround()
If this l-list is bound, returns true if its value is ground, that is it does not contain any unbound logical variable or collection. Raises an exception NotInitVarException if this l-list is unbound.

void printElems(char sep)
If this l-list is bound, prints all elements of its list value separated by the character sep (without the surrounding square brackets). If this l-list is unbound, prints ".extName", where extName is the external name of this l-list.

Vector toVector()
If this l-list is bound, returns a vector containing all of the elements of its list value. Otherwise, raises an exception NotInitVarException.

The following methods are also provided by the interface List of java.util and are intended to work on the list value possibly associated with a LList object. 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.

int getSize()
If this l-list is bound, returns the number of elements in its list value. Otherwise (i.e., if this l-list is unbound) it raises an exception NotInitVarException.
When the l-list v is bound, v.getSize() is equivalent to v.getValue().size().

Iterator iterator()
If this l-list is bound, returns an Iterator over the elements of its list value. Otherwise, it raises an exception NotInitVarException.
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, returns true if its list value contains o. Note that o can be a logical object (i.e., either a logical variable or a logical collection): in this case the value possibly associated with o is considered for the membership test. Similarly, elements of the list value associated with this l-list can be logical objects and their values are taken into account, if any. An exception NotInitVarException is raised if this l-list is unbound.
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 (e.g., a logical variable) or not.

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

### 3.4 Constraint methods

The class LList provides methods for generating equality and inequality constraints over lists.\(^2\)

\(^2\)The current version of the JSetL library does not support constraints over lists other than list equality and inequality. This limitation will be possibly removed in future releases by integrating other basic constraints over lists with set constraints as proposed for instance in [2].
Constraint `eq(List 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(List ll)`

Constraint `neq(List<?> l)`

Return the atomic constraint `this ≠ ll`, that requires this l-list to be different from the l-list `ll` (resp., from the generic Java list `l`).

Constraint `allDiff()`

If this l-list is bound, returns a conjunction of atomic constraints `e_i ≠ e_j` for all elements `e_i, e_j` (i different from j) of the list bound to this l-list. Raises an exception `NotInitVarException` if this l-list is unbound.

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 ∧ c_2 ∧ · · · ∧ c_n` for all elements `e_i, . . . , e_n` of the list bound to this l-list. Each `c_i` is obtained from `c` by replacing all occurrences of `y` in it 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 ≠ 0 ∧ 2 ≠ 0 ∧ z ≠ 0`. Raises an exception `NotInitVarException` if this l-list is unbound.

4 Logical sets: the class `LSet`

A logical set `s` (or, simply, a l-set `s`) is a special kind of logical variable whose value is a pair `(elems, rest)`, where `elems` is a set `{e_0, . . . , e_n}`, `n ≥ 0`, of objects of arbitrary types (the set value), and `rest` is either an empty or an unbound l-set representing the remainder of `s`. When `rest` is an unbound l-set `r`, we say that `s` represents an open set and we use the (abstract) notation `{e_0, . . . , e_n | r}` to denote it; conversely, when `rest` is the empty l-set, we say that `s` represents a closed set and we use the (abstract) notation `{e_0, . . . , e_n}`. When `elems` is the empty set and `rest` is `null`, `s` is the empty l-set and we use `{}` 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 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 \texttt{LSet}, which extends the class \texttt{LCollection}. In particular, set values (i.e., the \texttt{elems} part of the l-set) are instances of the class \texttt{HashSet} which implements the \texttt{java.util.Set} interface. Methods provided by the class \texttt{LSet} are basically the same as those of the class \texttt{LList} but applied to \texttt{LSet} objects. In particular, it provides methods to create new l-sets, possibly starting from existing ones, to deal with l-sets as logical variables 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 operations.

4.1 Constructors

\texttt{LSet()}

Create an unbound l-set, with no external name (resp., with external name \texttt{extName}).

\texttt{LSet(String extName)}

Create an unbound l-set, with no external name (resp., with external name \texttt{extName}).

\texttt{LSet(Set<?> s)}

Create a (bound) l-set, with no external name (resp., with external name \texttt{extName}), with value \texttt{s}. Same as to create an unbound l-set \texttt{x} and to post and solve the constraint \texttt{x.eq(s)}.

\texttt{LSet(LSet ls)}

Create a l-set, with no external name (resp., with external name \texttt{extName}), equivalent to the l-set \texttt{ls}. Same as to create an unbound l-set \texttt{x} and to post and solve the constraint \texttt{x.eq(ls)}.

Example 5

- Create an unbound l-set \texttt{a}.
  \begin{verbatim}
  LSet a = new LSet();
  \end{verbatim}

- Create a l-set \texttt{b}, with external name "b", bound to the set value \{1,2,3\}.
  \begin{verbatim}
  Set s = new HashSet();
  s.add(1); s.add(2); s.add(3);
  LSet b = new LSet("b",s);
  \end{verbatim}

- Create a l-set \texttt{c} equivalent to the l-set \texttt{a}.
  \begin{verbatim}
  LSet c = new LSet(a);
  \end{verbatim}

4.2 Creating new (bound) l-sets

\texttt{static LSet empty()}

Returns the empty l-set.

\texttt{LSet ins(Object o)}

If this l-set is bound, returns the (new) l-set whose value is obtained by adding \texttt{o} as an element of the set bound to this variable. Otherwise (i.e., this l-set is unbound), it returns the (new) l-set whose value is the set containing \texttt{o} as its element and this l-set as the rest part (i.e., \{o | ls\}, where \texttt{ls} is this l-set.
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 rest part
(i.e., \{a_1, \ldots, a_n \mid s\}, where a_1, \ldots, a_n are the elements of c and s is this l-set).

static LSet mkSet(int n)
static LSet mkSet(String extName, int n)
Returns a l-set, with no external name (resp., with external name extName), whose
value is a set of n unbound logical variables.

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 3 are still valid in the case of l-sets:
provided LList is replaced by LSet and insn is replaced by ins.

4.3 General utility methods
The class LSet provides all the utility methods of classes LVar (see Section 2.2) and LList
(see Section 3.3).

4.3.1 Basic methods
The following methods are the same as LVar’s methods, but adapted to l-set objects:

- LSet clone()
- boolean equals(LSet ls)
- boolean equals(Object o)
- String getName()
- Set<?> getValue()
- boolean isBound()
- void output()
- LSet setName(String extName)
- String toString()

Remarks of Section 3.3 apply to l-sets as well.

4.3.2 Logical collection methods
The class LSet provides all the collection methods of class LList (see Section 3.3.2). 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 rest part is simply ignored.

- Object get(int i)
- LSet getRest()
- int getSize()
- boolean isClosed()
- boolean isEmpty()
- boolean isGround()
- Iterator iterator()
- void printElems(char sep)
- boolean testContains(Object o)
- Vector toVector()

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 bound to 2, then \( s \) contains two occurrences of the same value 2. However, all methods dealing with l-sets, but method \texttt{toVector}, ignore repetitions (i.e., they consider multiple occurrences of the same value as a single set element). For example, calling \texttt{getSize} on the set \( s \) considered above, will get 1 as its result. Conversely, \( s\.toVector() \) will get the vector \( [2, 2] \).

• Since the order of elements in a l-set does not matter, \( s\.get(i) \), where \( s \) is a l-set, can return \textit{any} element of \( s \). The implementation only guarantees that different values of \( i \) correspond to different elements of \( s \).

4.4 Constraint methods

The class \texttt{LSet} provides methods for generating constraints over sets (i.e., \textit{l-set constraints}) that implement most of the usual set-theoretical operations.

Comparison constraints

Constraint \texttt{eq(LSet $ls$)}

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

Constraint \texttt{eq(Set<?> $s$)}

Returns the atomic constraint \texttt{this} = \texttt{$s$}, that unifies this l-set with the set \texttt{$s$}. If this l-set is unbound, it becomes bound to \texttt{$s$} after the constraint has been solved.

Constraint \texttt{neq(LSet $ls$)}
Constraint \texttt{neq(Set<?> $s$)}

Return the atomic constraint \texttt{this} $\neq$ \texttt{$s$}, that requires this l-set to be different from the l-set \texttt{$ls$} (resp., from the set \texttt{$s$}).

Membership constraints

Constraint \texttt{contains(LVar $lv$)}
Constraint \texttt{contains(Object $o$)}

Return the atomic constraint \texttt{$lv \in this$} (resp., \texttt{$o \in this$}), that requires this l-set to contain the logical variable \texttt{$lv$} (resp., the object \texttt{$o$}). Same as \texttt{$lv\.in(this)$} (resp., \texttt{new LVar($o$).in(this)}) (see Sect. 2.3).
Constraint `ncontains(LVar lv)`
Constraint `ncontains(Object o)`
  Return the atomic constraint `lv \notin this` (resp., `o \notin this`), that requires this l-set to not contain the logical variable `lv` (resp., the object `o`). Same as `lv.nin(this)` (resp., `new LVar(o).nin(this)`) (see Sect. 2.3).

**Set-theoretical constraints**
Constraint `diff(LSet ls1, LSet ls2)`
Constraint `diff(LSet ls, Set<?> s)`
Constraint `diff(Set<?> s, LSet ls)`
Constraint `diff(Set<?> s1, Set<?> s2)`
  Return the atomic constraint `this = ls1 \setminus ls2` (resp., `this = ls \setminus s`, `this = s \setminus ls`, `this = s1 \setminus s2`), that requires this l-set to be the difference of the other two l-set (resp., set) objects.

Constraint `disj(LSet ls)`
Constraint `disj(Set<?> s)`
  Return the atomic constraint `this \parallel ls` (resp., `this \parallel s`), that requires this l-set to be disjoint from the l-set `ls` (resp., from the set `s`) (i.e., `this \cap ls = \emptyset`, `this \cap o = \emptyset`).

Constraint `inters(LSet ls1, LSet ls2)`
Constraint `inters(LSet ls, Set<?> s)`
Constraint `inters(Set<?> s, LSet ls)`
Constraint `inters(Set<?> s1, Set<?> s2)`
  Return the atomic constraint `this = ls1 \cap ls2` (resp., `this = ls \cap s`, `this = s \cap ls`, `this = s1 \cap s2`).

Constraint `less(LSet ls, LVar lv)`
Constraint `less(LSet ls, Object o)`
Constraint `less(Set<?> s, LVar lv)`
Constraint `less(Set<?> s, Object o)`
  Return the atomic constraint `this = less(ls, lv)` (resp., `this = less(ls, o)`, `this = less(s, lv)`, `this = less(s, o)`), that requires this l-set to be equal to the l-set `ls` (resp., to the set `s`) without the object `lv` (resp., `o`), provided `lv` (resp., `o`) belongs to `ls` (resp., to `s`).

Constraint `size(IntLVar lv)`
Constraint `size(Integer i)`
  Return the atomic constraint `|this| = lv` (resp., `|this| = i`), that requires the logical variable `lv` (resp., the integer variable `i`) to be equal to the cardinality of this l-set.

Constraint `subset(LSet ls)`
Constraint `subset(Set<?> s)`
  Return the atomic constraint `this \subseteq ls` (resp., `this \subseteq s`).

Constraint `union(LSet ls1, LSet ls2)`
Constraint `union(LSet ls, Set<?> s)`
Constraint `union(Set<?> s, LSet ls)`
Constraint `union(Set<?> s1, Set<?> s2)`
  Return the atomic constraint `this = ls1 \cup ls2` (resp., `this = ls \cup s`, `this = s \cup ls`, `this = s1 \cup s2`).
Global constraints

Constraint allDiff()
Same as allDiff of class LList.

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

5 Integer logical variables: the class IntLVar

Integer logical variables are a special case of the logical variables described in Section 2, 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 is represented as a multi-interval, 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).

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 \( \text{sum, sub, mul and div} \), 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

\[
X_1 = e_1 \land X_2 = e_2 \land \ldots \land X_n = e_n,
\]

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 \( x.\text{sum}(y.\text{sub}(1)) \), where \( x \) and \( y \) are integer logical variables, the evaluation of \( e \) returns the integer logical variable \( X_1 \) with the associated constraint \( X_1 = x + X_2 \land X_2 = y - 1 \).

In JSetL, an integer logical variable is an instance of the class \textbf{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 \textbf{MultiInterval} (see Section A.2). Constraints possibly associated with integer logical variables are instances of the class Constraint (see Section 7).

The class \textbf{MultiInterval} provides the static fields \texttt{INF} and \texttt{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 \( \alpha \), respectively, and we will use \( \mathbb{Z}_\alpha \equiv [-\alpha..\alpha] \) 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 \( M \cap \mathbb{Z}_\alpha \).

\(^3\text{Note that MultiInterval.INF and MultiInterval.SUP have the same value as Interval.INF and}\)}
5.1 Constructors

IntLVar()
IntLVar(String extName)
Create 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., $[\text{MultiInterval.INF}..\text{MultiInterval.SUP}]$). The constraint associated with this variable is the empty conjunction.

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

IntLVar(IntLVar v)
IntLVar(String extName, IntLVar v)
Create 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.

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

IntLVar(MultiInterval m)
IntLVar(String extName, MultiInterval m)
Create 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 the 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 6

- Create an integer logical variable v with domain $\{-1,1..3\}$
  
  ```java
  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 $[\text{MultiInterval.INF}..0]$ (since $\|\{\text{MultiInterval.INF} - 2..0\}\|_\alpha = [-\alpha - 2..0] \cap \mathbb{Z}_\alpha = [-\alpha..0]$)

  ```java
  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.

*Interval.SUP*, respectively; then the latter can either be used in place of the former.
5.2 General utility methods

The class `IntLVar` provides all utility methods of the class `LVar` (see Section 2.2), 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(LVar lv)**
  Returns true iff this and lv are equal logical variables and they have equal domain and associated integer arithmetic constraint.

- **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.

5.3 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 $X_1 = X_0 + k \land C_0$, 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 $X_1 = X_0 + v \land C_v \land C_0$, 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 $X_1 = X_0 / k \land C_0 \land k \neq 0$, where $X_0$ is this logical variable and $C_0$ is the associated constraint.
IntLVar \texttt{div}(\texttt{IntLVar }v) \\
Same as above, but with an associated constraint \( X_1 = X_0 \div v \land C_v \land C_0 \land v \neq 0 \), 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 \texttt{div} operator refers to the "exact" integer division: a constraint of the form \( z = x / y \) is satisfiable iff the constraint \( x = z \times y \land y \neq 0 \) is satisfiable. For example, \( z = x / y \) with \( x = 7 \) and \( y = 2 \) is not satisfiable because the constraint \( 7 = z \times 2 \) is unsatisfiable for each integer value that \( z \) could take.

Example 7

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

\begin{verbatim}
IntLVar x = new IntLVar("x");
IntLVar y = new IntLVar("y");
IntLVar z = x.sum(y.sub(1)).setName("z");
z.output();
\end{verbatim}

Output:

\( z = \text{unknown} \quad -- \text{Constraint: } [\_z = \_x + \?, \_? = \_y - 1] \);

where \( \_? \) represents the internal name of the \texttt{IntLVar} object created in correspondence with the subexpression \texttt{y.sub(1)}.

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

5.4 Constraint methods

The class \texttt{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 \texttt{eq(Integer }k) \\
Returns the constraint \( X_0 = k \land C_0 \), where \( X_0 \) is this logical variable and \( C_0 \) is its associated constraint.

Constraint \texttt{eq(IntLVar }v) \\
Returns the constraint \( X_0 = v \land C_0 \land C_v \), 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 \texttt{neq(Integer }k) \\
Constraint \texttt{neq(IntLVar }v) \\
Same as above, but with \( = \) replaced by \( \neq \) in the generated constraint.

Constraint \texttt{le(Integer }k) \\
Constraint \texttt{le(IntLVar }v) \\
Same as above, but with \( = \) replaced by \( \leq \) in the generated constraint.
Constraint \texttt{lt(Integer k)}
Constraint \texttt{lt(IntLVar v)}
   Same as above, but with = replaced by < in the generated constraint.

Constraint \texttt{ge(Integer k)}
Constraint \texttt{ge(IntLVar v)}
   Same as above, but with = replaced by \geq in the generated constraint.

Constraint \texttt{gt(Integer k)}
Constraint \texttt{gt(IntLVar v)}
   Same as above, but with = replaced by > in the generated constraint.

Example 8
\begin{itemize}
\item The method invocation
\begin{verbatim}
  x.sub(1).lt(y.sum(3))
\end{verbatim}
where \texttt{x} and \texttt{y} are unbound logical variables (with external names "\texttt{x}" and "\texttt{y}", respectively), returns the constraint:
\begin{verbatim}
  [? < ?, ? = _x - 1, ? = _y + 3];
\end{verbatim}
where the two ? in the < constraint represent the internal names of the \texttt{IntLVar} objects created in correspondence with the subexpressions \texttt{x.sub(1)} and \texttt{y.sum(3)}, respectively.
\end{itemize}

Domain handling constraints

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

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

Constraint \texttt{dom(Set<Integer> s)}
Same as above, but with domain constraint \(X_0 :: |s|_\alpha\).

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

Membership constraints

Membership constraints involve integer logical variables and integer set logical variables (see Section 6)

Constraint \texttt{in(SetLVar X)}
Constraint \texttt{in(MultiInterval A)}
   Return the constraint \texttt{this \in X} (resp., the constraint \texttt{this \in A}).

Constraint \texttt{nin(SetLVar X)}
Constraint \texttt{nin(MultiInterval A)}
   Return the constraint \texttt{this \notin X} (resp., the constraint \texttt{this \notin A}).
All different constraints

Given \( n \geq 0 \) integer logical variables \( v_1, \ldots, v_n \) the constraint \( \text{allDifferent}(v_1, \ldots, v_n) \) is logically equivalent to the constraint:

\[
e = \bigwedge_{1 \leq i < j \leq n} v_i \neq v_j.
\]

The class \( \text{IntLVar} \) has two static methods called \( \text{allDifferent} \) which take as input a collection of integer logical variables and return such a constraint:

\[
\text{static Constraint allDifferent(AbstractList<IntLVar> vars)}
\]

\[
\text{static Constraint allDifferent(IntLVar[] vars)}
\]

Return the constraint \( \bigwedge_{1 \leq i < j \leq n} v_i \neq v_j \) if \( \text{vars} \) is an \( \text{AbstractList} \) (resp., a Java array) of \( \text{IntLVar} [v_1, \ldots, v_n] \).

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 \( \text{IntLVar} \) provides to support labeling.

\[
\text{Constraint label()}
\]

\[
\text{Constraint label(ValHeuristic val)}
\]

Label this variable, using the default value choice heuristic \( \text{GLB} \) (resp., using the value choice heuristic \( \text{val} \)).

\[
\text{static Constraint label(AbstractList<IntLVar> vars)}
\]

\[
\text{static Constraint label(IntLVar[] vars)}
\]

Label the variables in \( \text{vars} \), using the default value and variable choice heuristics \( \text{GLB} \) and \( \text{LEFT-MOST} \) respectively.

\[
\text{static Constraint label (AbstractList<IntLVar> vars, LabelingOptions lop)}
\]

\[
\text{static Constraint label (IntLVar[] vars, LabelingOptions lop)}
\]

Label the variables in \( \text{vars} \), using the heuristics specified in \( \text{lop} \).

Note that all these methods return an object of class \( \text{Constraint} \), since the labeling requests on one or more variables are treated as particular kinds of constraints over them.

6 Integer set logical variables: the class \( \text{SetLVar} \)

\emph{Integer set logical variables} (or more briefly \emph{set variables}) are a special case of the logical variables described in Section 2, in which values are restricted to be set of integers. Like integer logical variables, a set variable has a \emph{finite domain} and a (possibly empty) \emph{constraint associated} with it. Moreover, each set variable has an associated integer logical variable which represents its \emph{cardinality}.

The domain is represented as a \emph{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 \( \emptyset..\{1, 2, 3\} \) 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 5). Constraints possibly associated with integer set logical variables are instances of the class Constraint (see Section 7).

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 \([-\text{Interval}.\text{SUP} / 2..\text{Interval}.\text{SUP} / 2]\). As a notational convention (see Section A.3), we will use \( \beta \) to denote the value of the bounds of the maximum multi-interval (currently fixed to be \( \text{Interval}.\text{SUP} / 2 \)), and we will use \( Z_\beta \equiv [-\beta..\beta] \) 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 \( D \cap P(Z_\beta) \), while \( CH_\beta \) is used to denote the convex closure operation which is defined as \( \min\subseteq \{S \in \mathbb{S}_\beta : \|D\|_\beta \subseteq S\} \).

### 6.1 Constructors

**SetLVar()**

Create an unbound integer set logical variable, with no external name (resp., with external name \( \text{extName} \)). The domain of this variable is the 'universe' set-interval \([\text{SetInterval}.\text{INF}..\text{SetInterval}.\text{SUP}]\), which corresponds to the maximum representable set-interval. The constraint associated with this variable is the empty conjunction.

**SetLVar(String extName)**

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

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

SetLVar(Set<Integer> s, Set<Integer> t))
SetLVar(String extName, Set<Integer> s, Set<Integer> t)
Create an unbound integer set logical variable, with no external name (resp., with external name extName) and with domain the set-interval $\|[s..t]\|_\beta$. The associated constraint is the empty conjunction.

SetLVar(SetInterval s))
SetLVar(String extName, SetInterval s)
Create 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)
Create 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)
Create 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 the 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 9

• Create a SetLVar with domain $\emptyset..[1..3]$.

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, $\text{CH}_\beta(\{\{0\}, \{1\}, [2..\beta+1]\}) = \text{min}_{S \in S_\beta} \{S \subseteq \{\{0\}, \{1\}\} \subseteq S\} = [\varnothing, \{0, 1\}]$).

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 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);

6.2 General utility methods

The class SetLVar provides all utility methods of the class LVar (see Section 2.2), 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.

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.

6.3 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
$X_1 = \sim X_0 \land |X_0| + |X_1| = |\mathbb{Z}_\beta| \land C_0$, 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 \( \text{diff}(\text{MultiInterval } m) \)

Returns \( \text{this.diff(new SetLVar(m))} \).

SetLVar \( \text{diff}(\text{SetLVar } v) \)

Returns an integer set logical variable \( X_1 \) with an associated constraint
\[
X_1 = X_0 \setminus v \land X_1 \subseteq X_0 \land v \land |X_1| \geq |X_0| - |v| \land C_0 \land C_v
\]
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 \( v \parallel X_1 \) corresponds to the set disjointness between \( v \) and \( X_1 \) (thus, their intersection must be empty).

SetLVar \( \text{intersect}(\text{MultiInterval } m) \)

Returns \( \text{this.intersect(new SetLVar(m))} \).

SetLVar \( \text{intersect}(\text{SetLVar } v) \)

Returns an integer set logical variable \( X_1 \) with an associated constraint
\[
X_1 = X_0 \cap v \land X_1 \subseteq X_0 \land v \land |X_1| \leq |X_0| + |v| \land C_0 \land C_v
\]
where \( X_0 \) is this logical variable, \( C_0 \) its associated constraint and \( C_v \) is the constraint associated with the logical variable \( v \).

\text{static SetLVar \( \text{singleton}(\text{IntLVar } v) \) }

Returns an integer set logical variable \( X \) such that \( X = \{v\} \).

SetLVar \( \text{union}(\text{MultiInterval } m) \)

Returns \( \text{this.union(new SetLVar(m))} \).

SetLVar \( \text{union}(\text{SetLVar } v) \)

Returns an integer set logical variable \( X_1 \) with an associated constraint
\[
X_1 = X_0 \cup v \land X_0 \subseteq X_1 \land v \subseteq X_1 \land |X_1| \leq |X_0| + |v| \land C_0 \land C_v
\]
where \( X_0 \) is this logical variable, \( C_0 \) its associated constraint and \( C_v \) is the constraint associated with the logical variable \( v \).

Example 10

- Create an integer set logical variable with an associated set complement constraint.

  ```java
  SetLVar x = new SetLVar("x");
  SetLVar y = x.compl().setName("y");
  y.output();
  Output:
  y = _? -- Domain: [{},[-536870911..536870911]]
  -- Size: [0..1073741823] -- Constraint: _? = compl(_x)
  where _? represents the internal name of the SetLVar object created in correspondence with the subexpression x.compl().
  ```

6.4 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 \( \text{disj}(\text{MultiInterval } m) \)

Returns the constraint \( X_0 \parallel m \land C_0 \), where \( X_0 \) is this logical variable, \( C_0 \) is its associated constraint and \( \parallel \) is the set disjointness.
Constraint \texttt{disj(SetLVar } v) 
Returns the constraint \( X_0 \parallel v \lor |X_0| + |v| \leq |Z| \land C_0 \land C_v \), 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 \texttt{eq(MultiInterval } m) 
Returns the constraint \( X_0 = m \land C_0 \), where \( X_0 \) is this logical variable and \( C_0 \) is its associated constraint.

Constraint \texttt{eq(SetLVar } v) 
Returns the constraint \( X_0 = v \land |X_0| = |v| \land C_0 \land C_v \), 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 \texttt{neq(MultiInterval } m) 
Returns the constraint \( X_0 \neq m \land C_0 \), where \( X_0 \) is this logical variable and \( C_0 \) is its associated constraint.

Constraint \texttt{neq(SetLVar } v) 
Returns the constraint \( X_0 \neq v \land C_0 \land C_v \), 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 \texttt{strictSubset(MultiInterval } m) 
Returns the constraint \( X_0 \subseteq m \land |X_0| < |m| \land C_0 \), where \( X_0 \) is this logical variable and \( C_0 \) is its associated constraint.

Constraint \texttt{strictSubset(SetLVar } v) 
Returns the constraint \( X_0 \subseteq v \land |X_0| < |v| \land C_0 \land C_v \), 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 \texttt{subset(MultiInterval } m) 
Returns the constraint \( X_0 \subseteq m \land C_0 \), where \( X_0 \) is this logical variable and \( C_0 \) is its associated constraint.

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

\textbf{Example 11}

- \textit{Generate the constraint } \( X \subseteq Y \cup Z \).
  \[ X\text{.subset(Y.union(Z))}; \]

- \textit{Generate the constraint } \( X \cap Y = Y \setminus X \).
  \[ X\text{.intersect(Y).eq(Y.diff(X))}; \]

- \textit{Generate the constraint } \( X \neq \{-2, 7\} \).
  \[ X\text{.neq(new MultiInterval(-2).union(new MultiInterval(7))}); \]
Domain handling constraints

Constraint `dom(MultiInterval a, MultiInterval b)`
Returns the constraint \(X_0 :: [[a..b]] \land C_0\), where \(X_0\) is this logical variable and \(C_0\) is its associated constraint. The domain constraint \(X_0 :: [[a..b]]\) constrains \(X_0\) to belong to the domain \([[a..b]]\).

If such domain is empty, the exception `NotValidDomainException` is raised.

Constraint `dom(SetInterval s)`
Same as above, but with domain constraint \(X_0 :: s\).

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.

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

Note that, as for class `IntLVar`, all these methods return an object of class `Constraint`, since the labeling requests on one or more variables are treated as particular kinds of constraints over them.

Partially specified integer sets

The class `SetLVar` allows to define partially specified integer sets according to the CLP(\(S\text{ET}\)) 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:

\[
S = \{X_1, \ldots, X_n \mid R\} = \{X_1\} \cup \ldots \cup \{X_n\} \cup R.
\]

The following methods allow the user to define this kind of constraints.

Constraint `eq(IntLVar x)`
Returns the constraint \(X_0 = \{x\} \land C_0\), 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 \(X_0 = \{X_1, \ldots, X_n\} \land C_0\), 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`. 

26
Constraint \( \text{eq}(\text{IntLVar[]} \, \text{vars}, \, \text{SetLVar} \, r) \)

Constraint \( \text{eq}(\text{Collection<IntLVar>} \, \text{vars}, \, \text{SetLVar} \, r) \)

Returns the constraint \( X_0 = \{X_1, \ldots, X_n \mid r \} \land C_0 \), 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 \( \text{vars} \).

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:

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

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

\[
X_i \in S_i \land |S_i| = 1.
\]

7 Constraints: the class Constraint

Constraints represent operations that can be applied to logical variables and logical collections, as well as to objects created by 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 \([\;]\)
  - \( e_0.\text{op}(e_1, \ldots, e_n) \) or \( \text{op}(e_0, e_1, \ldots, e_n) \) with \( n = 0, \ldots, 3 \)
    where \( \text{op} \) is the name of the constraint and \( e_i \) (\( 0 \leq i \leq 3 \)) are expressions whose type depends on \( \text{op} \). In particular \( \text{op} \) can be one of a collection of predefined methods that implement operations such as equality, inequality, integer comparison, as well as basic set-theoretic operations (e.g., membership, union, intersection, etc.).

- (compound constraints):
  - \( c_1.\text{and}(c_2) \) (conjunction)
  - \( c_1.\text{or}(c_2) \) (disjunction)
  - \( c_1.\text{orTest}(c_2) \) (disjunction)
  - \( c_1.\text{impliesTest}(c_2) \) (implication)

where \( c_1 \) and \( c_2 \) are JSetL constraints and \( \text{and}, \text{or}, \text{orTest}, \text{impliesTest} \) represent the logical conjunction \( (c_1 \land c_2) \), disjunction \( (c_1 \lor c_2) \), and implication \( (c_1 \rightarrow c_2) \), between \( c_1 \) and \( c_2 \), respectively.

- (negation constraint):
  - \( c_1.\text{notTest}() \) (negation)

where \( c_1 \) is a JSetL constraint and \( \text{notTest} \) 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., \( \text{and}, \text{or} \)), as well as the result of calling a number of constraint methods supplied by the classes implementing logical objects presented in the previous sections.
7.1 Constructors

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

`Constraint(String extName, Object o1)`  
`Constraint(String extName, Object o1, Object o2)`  
`Constraint(String extName, Object o1, Object o2, Object o3)`  
`Constraint(String extName, Object o1, Object o2, Object o3, Object o4)`  
Create a constraint, with name `extName`, and from 1 to 4 arguments `o1`, ..., `o4`.

7.2 General utility methods

`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.

`int getAlternative()`  
See Sect. 9.3.

`Object getArg(int i)`  
Returns the `i`-th argument of this constraint if `1 ≤ i ≤ 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.

`void fail()`  
See Sect. 9.3.

`String getName()`  
Returns the external 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 variable or logical collection.

`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.,

```
constraint(extName, arg1, arg2, arg3, arg4),
```

where `arg_i` is either the `i`-th argument of this constraint (if `1 ≤ i ≤ k`) or `null` (for `i > k`), where `k` is the number of arguments of this constraint.
7.3 Meta-constraints

Constraint and(Constraint c)
Returns the constraint this ∧ c.

Constraint impliesTest(Constraint c)
Returns the constraint this → c, where → is the logical implication.

Constraint notTest()
Returns the constraint ¬this, where ¬ is the logical negation.

Constraint or(Constraint c)
Constraint orTest(Constraint c)
Return the constraint this ∨ 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
this or c 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.

7.4 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. 2.3
- constraints over LList objects: see Sect. 3.4
- constraints over LSet objects: see Sect. 4.4
- constraints over IntLvar objects: see Sect. 5.4
- constraints over SetLVar objects: see Sect. 6.4.

8 Constraint solving: the class SolverClass

Constraints are solved using a constraint solver. In JSetl a constraint solver can be created
as an instance of the class SolverClass. 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 SolverClass and its methods are described in detail in this section.
8.1 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.

```java
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. 9.3.

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. 7, 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:

```java
    solver.add(c_1);
    solver.add(c_2);
    ...
    solver.add(c_n);
```

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

boolean check(Constraint c)
If $C$ is the constraint currently in the constraint store of this solver, checks whether the constraint $C \land c$ is satisfiable or not, and returns true or false, respectively. If $C \land c$ is satisfiable, a viable constraint solution (i.e., a set of substitutions for the unbound logical variables occurring in the constraint) is also computed, if possible. Computing the solution may involve nondeterminism. The resulting constraint store will contain a possibly simplified form of the constraint $C \land c$. Conversely, if $C \land c$ is unsatisfiable, all unbound variables in $C \land c$ 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 $C \land c$ (resp., $C$) is found to be unsatisfiable, a failure exception is raised.

Remark. The JSetL constraint solving procedure is guaranteed to be complete for constraints involving only equality, inequality and l-set constraints. Conversely, if also constraints over IntLVar and SetLVar are considered, completeness is guaranteed only if solutions are forced to be computed through labeling (Sections 5.4 and 6.4).

8.3 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 assignments for x that makes the constraint $C \land c$ satisfiable. If $C \land c$ is unsatisfiable, it returns the empty LSet. In all cases, all unbound variables in $C \land c$ remain unchanged.

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

Example 12

• Print all solutions.

    LVar x = new LVar("x");
    LSet s = LSet.empty().ins(3).ins(2).ins(1).setName("s");
    solver.solve(x.in(s));
    do {
        x.output();
    }
while(solver.nextSolution());
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:

r = {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));
r.output();

Executing this code will output:

r = {}

9 User-defined constraints

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

9.1 The class NewConstraintsClass

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

```java
public class MyOps extends NewConstraintsClass {
    // 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 SolverClass methods for constraint solving.

For example, the statements
create an object of type \texttt{MyOps}, called \texttt{myOps}, and use it to invoke and solve the constraint \texttt{c1} over two objects \texttt{o1} and \texttt{o2}, using the constraint solver \texttt{solver} (provided \texttt{c1} is one of their constraint defined in \texttt{MyOps}).

### 9.2 Implementing new constraints

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

\textbf{Example 13 (Implementing new constraints)}

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

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

  public Constraint c2(t3 o3) {
    return new Constraint("c2", o3);
  }

  protected void user_code(Constraint c)
      throws Failure, NotDefConstraintException {
    if (c.getName() == "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 \texttt{ListOps} initializes the field \texttt{Solver} in the super class \texttt{NewConstraintsClass} with a reference to the solver currently in use by the user program.
The other public methods simply construct and return new objects of class `Constraint`. 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 `NewConstraintsClass`, 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 `NewConstraintsClass` 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 14 (New constraint absTest)**

```java
class MathOps extends NewConstraintsClass{
public MathOps(SolverClass s) {
    super(s);
}

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

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

private void absTest(Constraint c)
throws Failure {
    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:
```java
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:

```java
y = 3
```

Note that, by default, a user-defined constraint is always set to “solved” whenever it is processed by the solver. If this is not the case (e.g., the constraint is simply left unchanged in the constraint store), then the user should explicitly state that the constraint is still “unsolved”. This is obtained by using the following method of the class `Constraint`:

```java
void notSolved()
```

Sets the `solved` flag of this constraint to `false` (i.e., this constraint is “unsolved”).

### 9.3 Exploiting nondeterminism

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

```java
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.

```java
void fail() (in class Constraint)
```

Causes the invocation constraint to fail. This in turn causes the solver to backtrack to the nearest open choice point. If no open choice point exists, a `failure` exception is raised.

```java
void addChoicePoint(Constraint c) (in class solverClass)
```

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 14. 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 15 (New constraint abs)**

```java
private void abs(Constraint c)
    throws Failure {
        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


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 5 and 6).
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:

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

\(a\) 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 \(\mathbb{Z}_\alpha \overset{\text{def}}{=} [-\alpha..\alpha]\) and then the set \(I_\alpha\) of all the intervals contained in \(\mathbb{Z}_\alpha\), that is:

\[
I_\alpha \overset{\text{def}}{=} \{ [a..b] : a, b \in \mathbb{Z}_\alpha \}
\]

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

- \(\alpha = \text{Interval.SUP} \overset{\text{def}}{=} \text{Integer.MAX\_VALUE} / 2 = 1073741823\)
- \(-\alpha = \text{Interval.INF} \overset{\text{def}}{=} -\text{Interval.SUP} = -1073741823.\)

In addition to static fields \(\text{INF}\) and \(\text{SUP}\), this class also provides the static method \(\text{universe()}\) 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: \((\forall I \in I_\alpha)(\forall x \in I) -\alpha \leq x \leq \alpha\)
- convex: \((\forall I \in I_\alpha)(\forall x, y \in I) [x..y] \subseteq I.\)

Therefore, to overcome these limitations, two special operations are needed in order to represent generic integer sets as intervals belonging to \(I_\alpha\):

- \textbf{normalization:} is an operation \(|\cdot|_\alpha : \mathcal{P}(\mathbb{Z}) \rightarrow \mathcal{P}(\mathbb{Z}_\alpha)\) such that, for each \(A \subseteq \mathbb{Z}\),
  \[
  |A|_\alpha \overset{\text{def}}{=} A \cap \mathbb{Z}_\alpha.
  \]

- \textbf{convex closure:} is an operation \(\mathcal{CH}_\alpha : \mathcal{P}(\mathbb{Z}) \rightarrow I_\alpha\) such that, for each \(A \subseteq \mathbb{Z}\),
  \[
  \mathcal{CH}_\alpha(A) \overset{\text{def}}{=} \min\subseteq \{ I \in I_\alpha : |A|_\alpha \subseteq I \}.
  \]

Hence, the class constructors are defined as follows:

\textbf{Interval()} \hspace{1cm} Creates the empty interval \(\emptyset\).

\textbf{Interval(Integer a)} \hspace{1cm} Creates the interval \(|\{a\}|_\alpha\).

\textbf{Interval(Integer a, Integer b)} \hspace{1cm} Creates the interval \(|[a..b]|_\alpha\).

\textbf{Interval(Set<Integer> s)} \hspace{1cm} Creates the interval \(\mathcal{CH}_\alpha(s)\).
Example 16 (Interval constructors)

- Create an empty interval, since $\parallel\{\text{Interval.SUP + 1}\}\parallel_\alpha = \{\alpha + 1\} \cap \mathbb{Z}_\alpha = \emptyset$
  
  ```java
  Interval i = new Interval(Interval.SUP + 1);
  ```

- Create the interval $\parallel\{\text{Interval.INF - 2..0}\}\parallel_\alpha = [-\alpha - 2..0] \cap \mathbb{Z}_\alpha = [-\alpha..0]$
  
  ```java
  Interval i = new Interval(Interval.INF - 2, 0);
  ```

- Create the interval $\mathcal{CH}_\alpha(\{-3, 1, 0, 5\}) = [-3..5]$
  
  ```java
  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 $\text{this} \subseteq I$.

- **Interval intersect(Interval I)**
  
  Returns the interval corresponding to $\text{this} \cap I$.

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

  $$[a..b] \oplus [c..d] \overset{def}{=} \parallel\{a + c..b + d\}\parallel_\alpha$$

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

  $$[a..b] \ominus [c..d] \overset{def}{=} \parallel\{a - d..b - c\}\parallel_\alpha$$

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

Other utility methods

- **boolean contains(Integer k)**
  
  Returns true iff $k \in \text{this}$.

- **boolean isEmpty()**
  
  Returns true iff $\text{this} = \emptyset$.

- **boolean isSingleton()**
  
  Returns true iff $|\text{this}| = 1$. 

38
boolean isUniverse()
    Returns true iff this = $\mathbb{Z}_\alpha$.

int size()
    Returns $|\text{this}|$.

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

Integer getLub()
    Returns the LUB of this if this $\neq \emptyset$, 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 $M \subseteq \mathbb{Z}$ defined by $n \geq 0$ intervals $I_1, I_2, \ldots, I_n \in \mathbb{Z}_\alpha \setminus \emptyset$ 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 $[a..b] \prec [c..d] \iff b < c - 1$.

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

Example 17 Examples of multi-intervals are:

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

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 $\mathcal{M}_\alpha = \mathcal{P}(\mathbb{Z}_\alpha)$; in other terms, every subset of $\mathbb{Z}_\alpha$ is uniquely identified by a multi-interval in $\mathcal{M}_\alpha$ (and viceversa).
The class `MultiInterval` allows to represent and manipulate all the multi-intervals $M \in M_\alpha$.

Note that, although an interval is a particular case of multi-interval (is trivial to prove that $I_\alpha \subset M_\alpha$), `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 $\alpha$ 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 $\|\{a\}\|_\alpha$.

`MultiInterval(Integer a, Integer b)`

Creates the multi-interval $\|[a..b]\|_\alpha$.

`MultiInterval(Set<Integer> s)`

Creates the multi-interval corresponding to $\|s\|_\alpha$.

`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 18 (Multi-interval constructors)**

- Create the multi-interval $\|\{10, 5, 8, \alpha + 1, -1, 9, 0\}\|_\alpha = \{-1..0, 5..8\}$
  ```java
  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 $[-2.4] \cup [3.5] \cup \varnothing \cup [10..20] = [-2.5, 10..20]$
  ```java
  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 \( \mathbb{Z}_\alpha \).

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

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

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

```java
MultiInterval union(MultiInterval M)
    Returns this \( \cup M \).
```

```java
MultiInterval intersect(MultiInterval M)
    Returns this \( \cap M \).
```

```java
MultiInterval diff(MultiInterval M)
    Returns this \( \setminus M \).
```

```java
MultiInterval sum(MultiInterval M)
    Returns this \( \oplus M \), where in general:
    \[
    A \oplus B \overset{def}{=} \| \{ c \in \mathbb{Z} : (\exists a \in A)(\exists b \in B) c = a + b \} \|_\alpha.
    \]
```

```java
MultiInterval sub(MultiInterval M)
    Returns this \( \ominus M \), where in general:
    \[
    A \ominus B \overset{def}{=} \| \{ c \in \mathbb{Z} : (\exists a \in A)(\exists b \in B) c = a - b \} \|_\alpha.
    \]
```

```java
MultiInterval opposite()
    Returns \( \ominus \text{this} \overset{def}{=} \{0\} \ominus M \).
```

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:

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

```java
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):

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

$A$ 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 $\mathbb{Z}_\beta \overset{def}{=} [-\beta..\beta]$. The set $\mathbb{S}_\beta$ of all the set-intervals whose bounds are subsets of $\mathbb{Z}_\beta$ is:

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

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 $\|\cdot\|_\beta : \mathcal{P}^2(\mathbb{Z}) \rightarrow \mathcal{P}^2(\mathbb{Z}_\beta)$ such that:

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

is needed.

- convex: a convex closure operator $\mathcal{C}H_\beta : \mathcal{P}^2(\mathbb{Z}) \rightarrow \mathbb{S}_\beta$ such that:

$$\mathcal{C}H_\beta(D) \overset{def}{=} \min \subseteq \{S \in \mathbb{S}_\beta : \|D\|_\beta \subseteq S\}$$

is needed.

Since for each set-interval $[A..B] \in \mathbb{S}_\beta$ we have that $\emptyset \subseteq A$ 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 $\beta$ is Interval.SUP / 2. Moreover, SetInterval provides the static method universe() which returns the universe $[\emptyset..\mathbb{Z}_\beta]$.  

Constructors

SetInterval()

Creates the empty set-interval $\emptyset$. 

42
SetInterval(MultiInterval A)
  Creates the set-interval \( \|A\|_\beta \).  

SetInterval(MultiInterval A, MultiInterval B)
  Creates the interval \( \|A..B\|_\beta \).  

SetInterval(Collection<MultiInterval> D)
  Creates the interval \( CH_\beta(D) \).  

**Example 19 (Set-interval constructors)**

- **Create an empty set-interval, since \( \|\{2..\beta + 1\}\|_\beta = \{2..\beta + 1\} \cap P(\mathbb{Z}_\beta) = \emptyset \)**
  
  MultiInterval m = new MultiInterval(2, SetInterval.SUP.getLub() + 1);
  SetInterval s = new SetInterval(m);

- **Create the set-interval \( CH_\beta(\{\{0\},\{1\}, [2..\beta + 1]\}) = \min_{\subseteq}\{S \in S_\beta : \{\{0\},\{1\}\} \subseteq S\} = [\emptyset..\{0,1\}] \)**
  
  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 \( M \in \text{this} \).

**boolean isEmpty()**
  Returns true iff this = \( \emptyset \).

**boolean isSingleton()**
  Returns true iff \(|\text{this}| = 1\).

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

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

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

**MultiInterval getLub()**
  Returns the LUB of this if this \( \neq \emptyset \), 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 $\emptyset \ldots Z_{\beta}$) 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 20 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 $P[v = x] = P[v = y] = P[v = z] = \frac{1}{3}$.$^4$

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 $D_x$.

$^4$The notation $P[E]$ indicates the probability that a given event $E$ occurs.
**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 $\{1, \ldots, n\}$.

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 21** 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 20. Let us see now how the value choice heuristics work on them, indicating with $\lambda(v)$ the selected value for each variable $v \in \{x, y, z\}$.

<table>
<thead>
<tr>
<th></th>
<th>$\lambda(x)$</th>
<th>$\lambda(y)$</th>
<th>$\lambda(z)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLB</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>LUB</td>
<td>30</td>
<td>1000</td>
<td>40</td>
</tr>
<tr>
<td>MID_MOST</td>
<td>5</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>7</td>
<td>26</td>
<td>20</td>
</tr>
</tbody>
</table>

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

**EQUI_RANDOM**
- $P[\lambda(x) = a] = \frac{1}{13}$
- $P[\lambda(y) = b] = \frac{1}{52}$
- $P[\lambda(z) = c] = \frac{1}{41}$
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 22** The statements

\[
\text{RANGE\_RANDOM: } \mathbb{P}[\lambda(x) = a] = \begin{cases} 
\frac{1}{20}, & \text{se } a \in [1..10] \\
\frac{1}{6}, & \text{se } a \in [28..30] \\
\frac{1}{150}, & \text{se } b \in [1..50] 
\end{cases}
\]

\[
\mathbb{P}[\lambda(y) = b] = \begin{cases} 
\frac{1}{3}, & \text{se } b = 100 \\
\frac{1}{3}, & \text{se } b = 1000 
\end{cases}
\]

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

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

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

\[
\lambda(z) = 20
\]
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 5.4 and 6.4).

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 \( k \in X \lor k \notin X \) 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 23  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.

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