JSetL: a Java library to support Constraint Logic Programming

Author: Francesco Vetere
e-mail: francesco.vetere@studenti.unipr.it

University of Parma

A.Y. 2019/2020

Logic Programming

- **Declarative programming** is a programming style in which, generally speaking, the programmer provides the properties that the desired solution should have, rather than specifying the actual sequence of operations needed in order to obtain that solution.
- **Logic programming** is a declarative programming paradigm based on formal logic: basically, any program is composed by a list of *facts* and *rules*.
  Given a certain *goal*, we want to demonstrate the truth of the goal using the *knowledge base* formed by facts and rules.

Unification

A key concept in logic programming is the one of unification.

- In logic, **unification** is the algorithmic procedure used in solving equations involving symbolic expressions.
- In other words, by replacing certain sub-expression variables with other expressions, unification tries to **identify two symbolic expressions**.
- We say that two terms unify if they are the same term or if they contain variables that can be uniformly instantiated with terms in such a way that the resulting terms are equal.
- In **Prolog**, for example, when we try to unify two terms, the interpreter performs all the necessary variable instantiations, so that the terms really are equal afterwards: if the unification succeeds, Prolog also gives us the value of the instantiated variables.

Logic programming in Prolog

Let's see a basic example in Prolog:

family.pl

```
father(a, b).   /* Fact   1 */
father(b, c).   /* Fact   2 */

/* Rule   1 */
brother(X, Y) :- father(Z, X), father(Z, Y), X \= Y.
/* Rule   2 */
grandfather(X, Y) :- father(X, Z), father(Z, Y).
```

Let's now query our program:

```
?- grandfather(a, c).
true.
?- grandfather(X, c).
X = a .
```

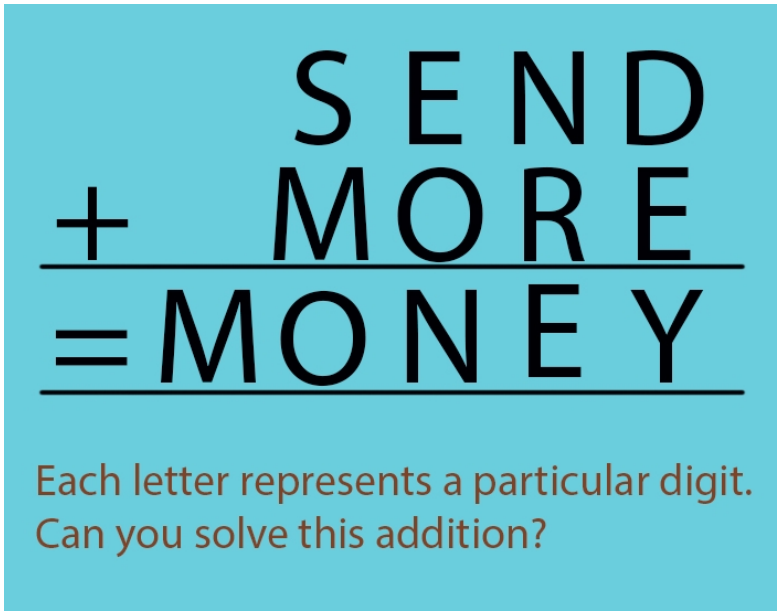
Constraint Logic Programming

- **Constraint Logic Programming** (*CLP*) is a form of logic programming in which the user can explicitly manipulate *constraints* (= relations over certain domains).

- Basically, it's an extension of the logic paradigm, in which we allow the user to use constraints into rules.
  These kind of problems are often called **Constraint Satisfaction Problems** (*CSP*), and Prolog supports them through different modules, like *clpfd*.

SEND + MORE = MONEY

- Let's analyze a classic CSP, the SEND + MORE = MONEY puzzle.
- It is a cryptarithmetic problem, in which we want to find numeric values for the letters S,E,N,D,M,O,R,Y, such that, if they are considered in decimal ordering, the equation holds.



SEND + MORE = MONEY in Prolog

Let's see the solution in Prolog:

money.pl

```
:- use_module(library(clpfd)).

money(S, E, N, D, M, O, R, Y) :-
    [S,E,N,D,M,O,R,Y] ins 0..9,
    all_different([S, E, N, D, M, O, R, Y]),
    S #>= 1, M #>= 1,
    1000*S + 100*E + 10*N + 1*D +
    1000*M + 100*O + 10*R + 1*E #=
    10000*M + 1000*O + 100*N + 10*E + 1*Y,
    label([S, E, N, D, M, O, R, Y]).
```

Let's now query our program:

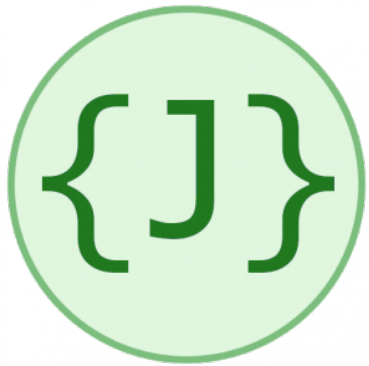
```
?- money(S, E, N, D, M, O, R, Y).
S = 9, E = 5, N = 6, D = 7,
M = 1, O = 0, R = 8, Y = 2.
```

SEND => 9567 +
MORE => 1085 =
MONEY => 10652

What is JSetL?

- **JSetL** is a Java library that has been developed at the University of Parma since 2002.
(`http://www.clpset.unipr.it/jsetl/`)

- It allows the user to use a **declarative programming style** inside Java, making it very easy to **declare and solve constraints** on logical objects.



Main Features

JSetL combines the classical O-O paradigm of Java with some **typical concepts** of the **constraint logic paradigm**, such as:

- Logical variables
- Unification
- Constraints resolution
- Non-determinism

JSetL classes

JSetL offers a series of useful **Java classes**, in order to fully support the declarative style we discussed.
Some of these classes are:

- LVar
- LSet
- Constraint
- Solver

LVar

- The class `LVar` implements the concept of **logical variable**, as already seen in Prolog: a variable in a logic programming language is initially undefined (**unbound**), but may get **bound** to a value or another logic variable during unification of the containing clause with the current goal.
  The value bounded to the variable may contain other variables which may themselves be bound or unbound.

- Clearly, this is **quite different** from the concept of variable in the **imperative paradigm**!

LVar

Examples:

```
/* Creation of an unbounded LVar (no associated name) */
LVar A = new LVar();

/* Creation of a LVar bounded to the integer 1 (no
    associated name) */
LVar B = new LVar(1);

/* Creation of a LVar bounded to the integer 3, with
    associated name "C" */
LVar C = new LVar("C", 3);
```

A special subclass of `LVar` is `IntLVar`:

```
/* Creation of an unbounded IntLVar with domain [0..9], and
    associated name "X" */
IntLVar D = new IntLVar("X", 0, 9);
```

These kind of `LVar` objects are very important, because we can apply to them constraints like `sum`, `mul`, and many others.

LSet

- The class `LSet` implements the concept of a **logical object collection**, organized in the form of a **mathematical set**.
- A logical set can contain any other object (in particular, other `LSet` objects), ignoring any element repetition.
- Similarly to `LVar`, `LSet` object can be bound or unbound, but they can also be **completely** or **partially specified**!

LSet

Examples:

```
/* Creation of an unbounded LSet without any associated name
   (it represent a completely variable(unspecified) set) */
LSet A = new LSet();

/* Creation of a completely specified set, bounded and
    without any associated name
   (it represents the set {1,2,3}) */
LSet B = LSet.empty().ins(1).ins(2).ins(3);

/* Creation of a partially specified set, unbounded and with
     associated name "C"
   (it represents the set {_LV1, _LV2} ∪ C)
   (note that the first two LVar have no associated name) */
LSet C = new LSet("C").ins(new LVar()).ins(new LVar());
```

Constraint

- The class `Constraint` implements the concept of **logical constraint** in JSetL.
- A constraint is a *relation* that can be applied to a logical variable (`LVar`) or to a logical set (`LSet`).
- It can be either *atomic* or *composed*.

Constraint

- **Atomic constraints**, which can be:
  - The *empty* constraint (denoted as `[]`)
  - $e_0.op(e_1, ..., e_n)$ or $op(e_0, e_1, ..., e_n)$ with $n = 0, ..., 3$ where op is the constraint name, and $e_i (0 \leq i \leq 3)$ are expressions whose type depends on op.
- **Composed constraints**, which can be obtained by:
  - Conjuction $(c_1.and(c_2))$
  - Disjunction $(c_1.or(c_2))$
  - Implication $(c_1.impliesTest(c_2))$

Some constraints on LSet

Constraint	JSetL	Semantic
Equality	`A.eq(B)`	$A = B$
Disjunction	`A.disj(B)`	$A \cap B = \varnothing$
Union	`A.union(B,C)`	$C = A \cup B$
Intersection	`A.inters(B,C)`	$C = A \cap B$
Difference	`A.diff(B,C)`	$C = A \setminus B$

Solver

- The **constraint solver** is implemented by the class `Solver`.
- This class provides methods to **add, show** and **solve** constraints.
- Constraints are solved through **syntax-rewriting rules**, and the solving algorithm ends when there are no more rules to apply (or when a `Failure` exception is thrown, in the case of a non satisfiable constraint).

Solver class methods

These are the main methods provided by the `Solver` class:

- public void **add**(Constraint c)
  Adds the constraint `c` to the constraint store of the solver.
- public void **showStore**()
  Prints the conjuction of all the constraint in the constraint store which are still in a non-solved form.
- public void **solve**()
  Tries to solve the constraints in the constraint store: if they are not satisfiable, it throws a `Failure` exception.
- public boolean **check**()
  Like `solve()`, but no exceptions are thrown: it just returns `true` if the constraints in the constraint store are satisfiable, `false` otherwise.

SEND + MORE = MONEY with JSetL

Let's try to implement the previous SEND + MORE = MONEY puzzle seen before in Prolog, this time in Java using JSetL:

Money.java

```
IntLVar s = new IntLVar("S", 0, 9);
/*...*/
IntLVar money = new IntLVar("MONEY");
IntLVar[] letters = {s, e, n, d, m, o, r, y};
Solver solver = new Solver();

/* S >= 1 and M >= 1 */
solver.add(s.ge(1).and(m.ge(1)));

/* Each variable is different from each other */
solver.add(Constraint.allDifferent(
   (Object[]) letters)
);
```

SEND + MORE = MONEY with JSetL

```
/* SEND = S*1000 + E*100 + N*10 + D */
solver.add(send.eq(s.mul(1000).sum(e.mul(100).sum(n.
    mul(10).sum(d)))));

/* MORE = M*1000 + O*100 + R*10 + E */
solver.add(more.eq(m.mul(1000).sum(o.mul(100).sum(r.
    mul(10).sum(e)))));

/* MONEY = M*10000 + O*1000 + N*100 + E*10 + Y */
solver.add(money.eq(m.mul(10000).sum(o.mul(1000).sum(n
    .mul(100).sum(e.mul(10).sum(y))))));

/* MONEY = SEND + MORE */
solver.add(money.eq(send.sum(more)));

/* Labeling on the variables */
solver.add(s.label());
/*...*/

/* Try to find a solution */
solver.solve();
```

Thanks for your attention!