Scala API for JaCoP Solver

Krzysztof Kuchcinski
Krzysztof.Kuchcinski@cs.lth.se

Department of Computer Science
Lund University
Sweden

Outline

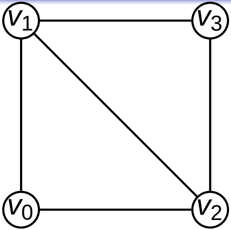
JaCoP library

- constraint programming paradigm implemented in Java.
- provides different type of constraints
  - *primitive constraints*, such as arithmetical constraints (+, *, div, mod, etc.), equality (=) and inequalities (<, >, =<, >=, !=).
  - *logical, reified and conditional constraints*
  - *global constraints*, such as `alldifferent`, `circuit`, `cumulative` and `diff2`.
  - *set constraints*, such as =, $\bigcup$, $\bigcap$.
- can be used from Java (JaCoP, JSR-331), Scala, MiniZinc
- http://www.jacop.eu
- http://sourceforge.net/projects/jacop-solver/

Motivation

- Scala API for Java library
- Hides JaCoP library details
- Offers high-level constraint programming constructs
- Simple and intuitive use without confusing Scala constructs and CP constructs

Example in Java




```
import JaCoP.core.*;
import JaCoP.constraints.*;
import JaCoP.search.*;

public class Main {
    static Main m = new Main ();

    public static void main (String[] args) {
        Store store = new Store();  // define FD store
        int size = 4;
        // define finite domain variables
        IntVar[] v = new IntVar[size];
        for (int i=0; i<size; i++)
            v[i] = new IntVar(store, "v"+i, 1, size);
        // define constraints
        store.impose( new XneqY(v[0], v[1]) );
        store.impose( new XneqY(v[0], v[2]) );
        store.impose( new XneqY(v[1], v[2]) );
        store.impose( new XneqY(v[1], v[3]) );
        store.impose( new XneqY(v[2], v[3]) );

        // search for a solution and print results
        Search<IntVar> search = new DepthFirstSearch<IntVar>();
        SelectChoicePoint<IntVar> select = new InputOrderSelect<IntVar>(store, v,
                                                                new IndomainMin<IntVar>());
        boolean result = search.labeling(store, select);

        if ( result )
            System.out.println("Solution: " + v[0]+", "+v[1] +", "+v[2] +", "+v[3]);
        else System.out.println("*** No");
    }
}
```

Example (cont'd)

The program produces the following output indicating that vertices v0, v1 and v3 get different colors (1, 2 and 3) while vertex v3 gets color 1.

```
Solution: v0=1, v1=2, v2=3, v3=1
```

Example in MiniZinc

```
array [0..3] of var 1..4: v;

constraint
   v[0] != v[1] /\
   v[0] != v[2] /\
   v[1] != v[2] /\
   v[1] != v[3] /\
   v[2] != v[3];
solve :: int_search(v, input_order, indomain_min, complete) satisfy;

output[ show(v) ];
```

Example in JaCoP/Scala

```
import JaCoP.scala._

object Color extends App with jacop {

  val size = 4

  val v = Array.tabulate(size)(i => new IntVar("v"+i, 1, size))

  v(0) #\= v(1)
  v(0) #\= v(2)
  v(1) #\= v(2)
  v(1) #\= v(3)
  v(2) #\= v(3)

  val result = satisfy( search(v, input_order, indomain_min))

  if (result) {
    print("Solution : ")
    for (i <- 0 until size) print(v(i) + " ")
    println
  }
    else println("No solution")
}
```

Approach

- Objects
- Operator overloading
- Implicit conversion functions
- Package object
- First-class functions (passed as values)

Operator overloading

```
object Model extends JaCoP.core.Store {
....
}
```

```
class IntVar(name: String, min: Int, max: Int) extends
          JaCoP.core.IntVar(Model, name, min, max) with jacop {
....

  def #\=(that: JaCoP.core.IntVar) = {
    val c = new XneqY(this, that)
    Model.constr += c
    c
  }
....
}
```

Implicit functions

```
trait jacop {
/**
 * Converts integer to IntVar.
 *
 * @param i intger to be converted.
 */
  implicit def intToIntVar(i: Int): IntVar = {
    val v = new IntVar(i, i)
    v
  }

....

}
```

Package Object

- Each package is allowed to have one package object
- Definitions in the package object are members of package itself
- contains “global definitions”

Package Object

```
package object scala {
....
  def alldifferent(x: Array[IntVar]) : Unit = {
    val c = new Alldiff( x.asInstanceOf[Array[JaCoP.core.IntVar]] )
    if (trace) println(c)
    Model.impose( c )
  }

....
  def satisfy[T <: JaCoP.core.Var](select: SelectChoicePoint[T],
                                    printSolutions: () => Unit*)
                                    (implicit m: ClassManifest[T]): Boolean = {
    ....
  }
....
}
```

Functions as Parameters

```
....
val result = minimize( search(t, smallest_min, indomain_min), end, printSol )
statistics

def printSol() : Unit = {
  println("\nSolution with cost: " + end.value + "\n=======================")
  println(t)
}
....
```

Nonvariance and Covariance

```
IntVar <: Var
SetVar <: Var
BooleanVar <: Var
```

Nonvariance and Covariance

```
IntVar <: Var
SetVar <: Var
BooleanVar <: Var
```

but

Array[IntVar] is not a subtype of Array[Var]

Nonvariance and Covariance

```
IntVar <: Var
SetVar <: Var
BooleanVar <: Var
```

but
Array[IntVar] is not a subtype of Array[Var]

BUT
List[IntVar] IS a subtype of List[Var]

Nonvariance and Covariance

```
IntVar <: Var
SetVar <: Var
BooleanVar <: Var
```

but
Array[IntVar] is not a subtype of Array[Var]

BUT
List[IntVar] IS a subtype of List[Var]

```
def search[T <: JaCoP.core.Var](vars: List[T],
     heuristic: ComparatorVariable[T], indom: Indomain[T])
     (implicit m: ClassManifest[T]) :
               SelectChoicePoint[T] = {
  new SimpleSelect[T](vars.toArray, heuristic, indom)
}
```

Reification

$b \Leftrightarrow (X = Y)$ or $(X = Y) \Leftrightarrow b$

Reification

$b \Leftrightarrow (X = Y)$ or $(X = Y) \Leftrightarrow b$

Scala code:

```
b <=> (X #= Y)      but not (C #= Y) <=> B
```

Reification

$b \Leftrightarrow (X = Y)$ or $(X = Y) \Leftrightarrow b$

Scala code:

`b <=> (X #= Y)` but not `(C #= Y) <=> B`

- we do not want to extend **each** primitive constraint
- also the reason why operator overloading does not create directly a new constraint (return constraint)

Element constraint

$$vector[index] = value$$

Element constraint

$$vector[index] = value$$

$$element(index, vector, value)$$

Element constraint

$$vector[index] = value$$

$$element(index, vector, value)$$

- cannot “override” vector access to array of Int or IntVar with index IntVar

Conclusions

- More information and discussion
  - `http://sourceforge.net/projects/jacop-solver/`
  - Håkan's “My constraint Programming blog”