Introduction to Constraint Programming in Copris

Naoyuki Tamura

2018-03-21

Contents

1 Introduction

This document describes an introduction to Constraint Programming by using Copris which is a DSL (Domain Specific Language) for constraint programming embedded in Scala language.

2 Installation

1. Install Java runtime version 1.8 or later
2. Install Scala version 2.11
3. Check the version of your Scala

```
$ scala -version
Scala code runner version 2.11.8 -- Copyright 2002-2016, LAMP/EPFL
```

   Copris can run on Scala version 2.11 (not on other versions).

4. Download Copris package
5. Unzip Copris package

3 First Step

3.1 Using REPL

Let us consider the following example.

> You have some of 1 cent, 5 cents, and 10 cents coins. The number of coins is 15 in total and at least one for each. The total sum of the coins is 90 cents. What are the numbers of each coin?

The problem can be formalized as the following CSP (Constraint Satisfaction Problem).

$$
\begin{aligned}
x &\in \{1\,..\,15\} \\
y &\in \{1\,..\,15\} \\
z &\in \{1\,..\,15\} \\
x + y + z &= 15 \\
x + 5y + 10z &= 90
\end{aligned}
$$

Let us solve the problem in Copris. Move to the "~build~" folder of Copris, and start Copris as follows.

```
$ scala -cp copris-all-v2-2-8.jar
```

3.1.1 Defining CSP

After you get the input prompt of Scala, you can directly define your CSP within Scala REPL.

```
scala> import jp.kobe_u.copris._
scala> import jp.kobe_u.copris.dsl._
scala> int('x, 1, 15)
scala> int('y, 1, 15)
scala> int('z, 1, 15)
scala> add('x + 'y + 'z === 15)
scala> add('x + 'y * 5 + 'z * 10 === 90)
```

- The first line imports the class names provided by Copris.
- The second line imports the methods provided by Copris DSL.
- The third line declares the integer variable x (its lower and upper bounds are 1 and 15 respectively). The notation starts from single quotation mark denotes a Symbol object in Scala, and it is implicitly converted to the integer variable object (Var object) of Copris by Copris DSL.
- The fourth and fifth lines declare the integer variables y and z in the same way.

- The sixth line adds the constraint $x + y + z = 15$. Note that `===` is used for the equality. The method `add` adds the constraint to the CSP object.
- The seventh line adds the constraint $x + 5y + 10z = 90$. You can not use the notation `5 * 'y`. Use `Num(5) * 'y` when you want to write an integer constant prior.

The variable 'csp' refers the defined CSP object.

```
scala> csp
res: CSP = CSP(Vector(x, y, z),Vector(),Map(...),Vector(...))
```

The CSP object consists of a sequence of integer variables 'variables', a sequence of Boolean variables 'bools', a map of integer variable domains 'dom', and a sequence of constraints 'constraints'.

```
scala> csp.variables
res: IndexedSeq[Var] = Vector(x, y, z)

scala> csp.bools
res: IndexedSeq[Bool] = Vector()

scala> csp.dom
res: Map[Var,Domain] = Map(x -> Domain(1,15), y -> Domain(1,15), z -> Domain(1,15))

scala> csp.constraints
res: IndexedSeq[Constraint] = Vector(Eq(Add(Add(x,y),z),15), Eq(Add(Add(x,Mul(y,5)),Mul(z,10)),90))
```

The CSP can be displayed by 'show' command.

```
scala> show
int(x,1,15)
int(y,1,15)
int(z,1,15)
Eq(Add(Add(x,y),z),15)
Eq(Add(Add(x,Mul(y,5)),Mul(z,10)),90)
```

- Here, 'Add' means addition, 'Mul' means multiplication, and 'Eq' means equality.

CSP object is implemented as a mutable object for which additions of variables and constraints are allowed.

3.1.2 Searching solutions

The 'find' method searches the first solution of the defined CSP.

```
scala> find
res: Boolean = true
```

The result 'true' means there exists a solution. The CSP solution of CSP is stored in the 'solution' variable.

```
scala> solution
res: Solution = Solution(Map(x -> 5, y -> 3, z -> 7),Map())
```

A Solution object consists of a map representing an assignment for integer variables (Var objects) and a map representing an assignment for Boolean variables (Bool objects).

```
scala> solution.intValues
res: Map[Var,Int] = Map(x -> 5, y -> 3, z -> 7)

scala> solution.boolValues
res: Map[Bool,Boolean] = Map()
```

A value assigned for each variable can be obtained by 'solution' method.

```
scala> solution('x)
res: Int = 5

scala> solution('y)
res: Int = 3

scala> solution('z)
res: Int = 7
```

The 'findNext' method searches the next solution.

```
scala> findNext
res: Boolean = false
```

The result 'false' means there exist no more solutions. And the 'solution' has null value.

```
scala> solution
res: Solution = null
```

You can confirm the extra condition by 'show' method.

```
scala> show
int(x,1,15)
int(y,1,15)
int(z,1,15)
Eq(Add(Add(x,y),z),15)
Eq(Add(Add(x,Mul(y,5)),Mul(z,10)),90)
Not(And(And(Eq(x,5),Eq(y,3),Eq(z,7)),And()))
```

The extra condition can be removed by 'cancel' method.

```
scala> cancel
scala> show
int(x,1,15)
int(y,1,15)
int(z,1,15)
Eq(Add(Add(x,y),z),15)
Eq(Add(Add(x,Mul(y,5)),Mul(z,10)),90)
```

A list of solutions can be obtained by 'solutions' method.

```
scala> solutions.foreach(println)
Solution(Map(x -> 5, y -> 3, z -> 7),Map())
```

You need to remove extra conditions by 'cancel' method when you want to get all solutions after 'findNext' is executed.

3.2 Writing script files

CSP can be defined as a script file of Scala.
The next shows the script file of the above example (Example-csp.scala).

```
1:  import jp.kobe_u.copris._
2:  import jp.kobe_u.copris.dsl._
3:
4:  int('x, 1, 15)
5:  int('y, 1, 15)
6:  int('z, 1, 15)
7:  add('x + 'y + 'z === 15)
8:  add('x + 'y * 5 + 'z * 10 === 90)
```

The script file can be executed by the ':load' command.

```
scala> :load Example-csp.scala
Loading Example-csp.scala...
.....
scala> find
res: Boolean = true
scala> solution
res: Solution = Solution(Map(x -> 5, y -> 3, z -> 7),Map())
```

If you want to reload the script file after you modify it, use 'init' method to initialize the CSP and the Solver before loading.

```
scala> init
scala> :load Example-csp.scala
```

We will use 'init' method in the head of the script hereafter.

3.2.1 Exercise

1. Try the example for 105 cents.

4 Small Examples

4.1 A Coin Problem

First example is a coin problem.

> You have the following Euro coins (148 cents in total):
>
> - four 20 cents coins,
> - four 10 cents coins,
> - four 5 cents coins, and
> - four 2 cents coins.
>
> Is there a combination for 93 cents in total?

The problem can be formalized as the following CSP (Constraint Satisfaction Problem). Each variable x_i represents the number of i cents coins.

$$
\begin{aligned}
&x_{20} \in \{0\,..\,4\} \\
&x_{10} \in \{0\,..\,4\} \\
&x_{5} \in \{0\,..\,4\} \\
&x_{2} \in \{0\,..\,4\} \\
&20x_{20} + 10x_{10} + 5x_5 + 2x_2 = 93
\end{aligned}
$$

The CSP file can be written as follows. (Coin-csp.scala).

```
1:  import jp.kobe_u.copris._
2:  import jp.kobe_u.copris.dsl._
3:
4:  init
5:  int('x(20), 0, 4)
6:  int('x(10), 0, 4)
7:  int('x(5), 0, 4)
8:  int('x(2), 0, 4)
9:  add('x(20) * 20 + 'x(10) * 10 + 'x(5) * 5 + 'x(2) * 2 === 93)
```

In Copris, you can use indices as used in `'x(20)`. Indices can be integers or character strings (or any Scala objects). It is also possible to use multiple indices, such as `'x(1,"a")`.

4.1.1 Exercise

1. How many solutions are there for the above problem?

   - **(Answer)**
     You can find it as follows.

     ```
     scala> cancel
     scala> solutions.size
     scala> cancel
     scala> solutions.foreach(println)
     ```

2. How many solutions are there when the number of coins is three?

   - **(Answer)**
     You can find it as follows.

     ```
     scala> :load Coin-csp.scala
     scala> add('x(20) <= 3)
     scala> add('x(10) <= 3)
     scala> add('x(5) <= 3)
     scala> add('x(2) <= 3)
     scala> find
     ```

4.2 Subset Sum Problem

Is there a subset of $\{2, 3, 5, 8, 13, 21, 34\}$ which sums to 50?

This problem is known as the **Subset Sum Problem** (Wikipedia:Subset sum problem). The subset sum problem is NP-complete.

This problem can be formalized as the following CSP (Constraint Satisfaction Problem).

$$
\begin{aligned}
& x_2 \in \{0, 1\} \\
& x_3 \in \{0, 1\} \\
& x_5 \in \{0, 1\} \\
& x_8 \in \{0, 1\} \\
& x_{13} \in \{0, 1\} \\
& x_{21} \in \{0, 1\} \\
& x_{34} \in \{0, 1\} \\
& 2x_2 + 3x_3 + 5x_5 + 8x_8 + 13x_{13} + 21x_{21} + 34x_{34} = 50
\end{aligned}
$$

The CSP file can be written as follows. (SubsetSum-csp.scala).

```
 1:  import jp.kobe_u.copris._
 2:  import jp.kobe_u.copris.dsl._
 3:
 4:  def define(sum: Int) {
 5:    init
 6:    boolInt('x(2))
 7:    boolInt('x(3))
 8:    boolInt('x(5))
 9:    boolInt('x(8))
10:    boolInt('x(13))
11:    boolInt('x(21))
12:    boolInt('x(34))
13:    add('x(2)*2 + 'x(3)*3 + 'x(5)*5 + 'x(8)*8 + 'x(13)*13 + 'x(21)*21 + 'x(34)*34 === sum)
14:  }
```

The 'boolInt' method declares a 0-1 integer variable, and `boolInt(x)` is equivalent to `int(x, 0, 1)`.
In the above program, the function `define(sum: Int)` is declared to define the CSP for the given summation.

```
scala> :load SubsetSum-csp.scala
scala> define(50)
scala> find
```

4.2.1 Exercise

1. How the solution will be when the sum is 40?

- **(Answer)**
  You can find it as follows.

```
scala> define(40)
scala> find
```

4.3 Magic Square

Arrange the numbers from 1 to 9 in 3×3 matrix such that the sum of each row, each column, and each diagonal becomes 15.

This arrangement is called a **Magic Square** (Wikipedia:Magic square).
This problem can be formalized as the following CSP.

$$
\begin{aligned}
& x_{ij} \in \{1\,..\,9\} \qquad (i = 0\,..\,2,\ j = 0\,..\,2) \\
& \text{Alldifferent}(x_{00}, x_{01}, \ldots, x_{22}) \\
& x_{i0} + x_{i1} + x_{i2} = 15 \qquad (i = 0\,..\,2) \\
& x_{0j} + x_{1j} + x_{2j} = 15 \qquad (j = 0\,..\,2) \\
& x_{00} + x_{11} + x_{22} = 15 \\
& x_{02} + x_{11} + x_{20} = 15
\end{aligned}
$$

Here, 'Alldifferent' means a constraint where the given arguments are mutually different.
That is, Alldifferent$(x_1, x_2, \ldots, x_n)$ is equivalent to $x_i \neq x_j$ (for all $i < j$).
The CSP file can be written as follows. (MagicSquare3-csp.scala).

```
 1:  import jp.kobe_u.copris._
 2:  import jp.kobe_u.copris.dsl._
 3:
 4:  init
 5:  int('x(0,0), 1, 9); int('x(0,1), 1, 9); int('x(0,2), 1, 9)
 6:  int('x(1,0), 1, 9); int('x(1,1), 1, 9); int('x(1,2), 1, 9)
 7:  int('x(2,0), 1, 9); int('x(2,1), 1, 9); int('x(2,2), 1, 9)
 8:  add(Alldifferent(
 9:    'x(0,0), 'x(0,1), 'x(0,2),
10:    'x(1,0), 'x(1,1), 'x(1,2),
11:    'x(2,0), 'x(2,1), 'x(2,2)
12:  ))
13:  add('x(0,0) + 'x(0,1) + 'x(0,2) === 15)
14:  add('x(1,0) + 'x(1,1) + 'x(1,2) === 15)
15:  add('x(2,0) + 'x(2,1) + 'x(2,2) === 15)
16:  add('x(0,0) + 'x(1,0) + 'x(2,0) === 15)
17:  add('x(0,1) + 'x(1,1) + 'x(2,1) === 15)
18:  add('x(0,2) + 'x(1,2) + 'x(2,2) === 15)
19:  add('x(0,0) + 'x(1,1) + 'x(2,2) === 15)
20:  add('x(0,2) + 'x(1,1) + 'x(2,0) === 15)
```

4.3.1 Exercise

1. How many solutions of 3×3 magic square?

   - **(Answer)**
     You can find it as follows.

     ```
     scala> solutions.size
     ```

2. What constraints should be added to break symmetries of rotation and reflection?

   - **(Answer)**
     You can find it as follows.

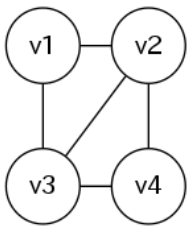
     ```
     scala> cancel
     scala> add('x(0,0) < 'x(0,2))
     scala> add('x(0,0) < 'x(2,0))
     scala> add('x(0,0) < 'x(2,2))
     scala> add('x(0,2) < 'x(2,0))
     scala> solutions.foreach(println)
     ```

3. Write a CSP of $n \times n$ magic square in Copris.

   - **(Answer)**
     See MagicSquare-csp.scala.

4.4 Graph Coloring Problem

For each vertex of the following graph, assign the numbers from 1 to 3 such that the adjacent vertices have different numbers.



This problem is known as a **Graph Coloring Problem** (GCP) (Wikipedia:Graph coloring).
In the above graph, you can color vertices as $v_1 = 1$, $v_2 = 2$, $v_3 = 3$, and $v_4 = 1$. In addition, there are no such coloring with two colors.

This problem can be formalized as the following CSP.

$$
\begin{aligned}
v_1 &\in \{1,2,3\} \\
v_2 &\in \{1,2,3\} \\
v_3 &\in \{1,2,3\} \\
v_4 &\in \{1,2,3\} \\
v_1 &\neq v_2 \\
v_1 &\neq v_3 \\
v_2 &\neq v_3 \\
v_2 &\neq v_4 \\
v_3 &\neq v_4
\end{aligned}
$$

The CSP file can be written as follows. (GCP-csp.scala).

```
 1:  import jp.kobe_u.copris._
 2:  import jp.kobe_u.copris.dsl._
 3:
 4:  def define(edges: Set[(Int,Int)], colors: Int) {
 5:    init
 6:    val nodes = for ((i,j) <- edges; k <- Seq(i,j)) yield k
 7:    for (i <- nodes)
 8:      int('v(i), 1, colors)
 9:    for ((i,j) <- edges)
10:      add('v(i) =/= 'v(j))
11:  }
12:
13:  val graph1 = Set((1,2), (1,3), (2,3), (2,4), (3,4))
```

```
scala> :load GCP-csp.scala
scala> define(graph1, 3)
scala> find
```

4.4.1 Exercise

1. How to define the CSP when the adjacent vertices have numbers at least two apart? (Hint: the absolute value of `x` is written as `Abs(x)` in Copris)

4.5 Sudoku

Next, we try to solve **Sudoku** (Wikipedia:Sudoku).
The CSP file can be written as follows. (Sudoku-csp.scala).

```
 1:  import jp.kobe_u.copris._
 2:  import jp.kobe_u.copris.dsl._
 3:
 4:  def define(board: Seq[Seq[Int]]) {
 5:    val n = 9; val m = 3
 6:    init
 7:    for (i <- 0 until n; j <- 0 until n)
 8:      int('x(i,j), 1, n)
 9:    for (i <- 0 until n)
10:      add(Alldifferent((0 until n).map(j => 'x(i,j))))
11:    for (j <- 0 until n)
12:      add(Alldifferent((0 until n).map(i => 'x(i,j))))
13:    for (i <- 0 until n by m; j <- 0 until n by m) {
14:      val xs = for (di <- 0 until m; dj <- 0 until m)
15:               yield 'x(i+di,j+dj)
16:      add(Alldifferent(xs))
17:    }
18:    for (i <- 0 until n; j <- 0 until n)
19:      if (board(i)(j) > 0)
20:        add('x(i,j) === board(i)(j))
21:  }
22:
23:  val sudoku_fujiwara = Seq(
24:    Seq(0, 0, 0, 0, 0, 0, 0, 0, 0),
25:    Seq(0, 4, 3, 0, 0, 0, 6, 7, 0),
26:    Seq(5, 0, 0, 4, 0, 2, 0, 0, 8),
```

```
27:    Seq(8, 0, 0, 0, 6, 0, 0, 0, 1),
28:    Seq(2, 0, 0, 0, 0, 0, 0, 0, 5),
29:    Seq(0, 5, 0, 0, 0, 0, 0, 4, 0),
30:    Seq(0, 0, 6, 0, 0, 0, 7, 0, 0),
31:    Seq(0, 0, 0, 5, 0, 1, 0, 0, 0),
32:    Seq(0, 0, 0, 0, 8, 0, 0, 0, 0)
33:  )
```

```
scala> :load Sudoku-csp.scala
scala> define(sudoku_fujiwara)
scala> find
```

4.5.1 Exercise

1. Try to solve problems in http://puzzle.gr.jp.

4.6 Bin Packing Problem

> Four persons will carry eight bags. Weight of each bag is 3.3kg, 6.1kg, 5.8kg, 4.1kg, 5.0kg, 2.1kg, 6.0kg, and 6.4kg (38.8kg in total). Can you assign bags to persons so that each person carries at most 11kg?

This problem is known as a **Bin Packing Problem** (Wikipedia:Bin packing problem).

To formalize this problem as CSP, we consider a matrix where each row corresponds to a person, and each column corresponds to a baggage. Each components of the matrix is eight 0 or 1. Such matrix is called an **incidence matrix**.

Consider the constraints for each row and column of the incidence matrix. Note that you need to specify the weights in 100g unit for example since real numbers are not allowed in Copris.

4.6.1 Exercise

1. What is the constraint for each row?
2. What is the constraint for each column?
3. Describe the CSP in Copris.
   - **(Answer)**
     See BinPacking-csp.scala.
4. How the solutions will be when the maximum weight is 10kg?
   - **(Answer)**
     You can find it as follows.

```
scala> :load BinPacking-csp.scala
scala> ex1(max = 110)
```

4.7 Knapsack Problem

> There are five items, and the price and the value for each item is given by the following table. Is there a combination of items where the total price is at most 15, and the total value is at least 19?

Item	Price	Value
Item0	3	4
Item1	4	5
Item2	5	6
Item3	7	8
Item4	9	10

The CSP file can be written as follows. (Knapsack-csp.scala).

```
1:  import jp.kobe_u.copris._
2:  import jp.kobe_u.copris.dsl._
3:
4:  def define(maxCost: Int, minValue: Int, costs: Seq[Int], values: Seq[Int]) {
```

```
 5:    init
 6:    val n = costs.size
 7:    for (i <- 0 until n)
 8:      boolInt('x(i))
 9:    val c = for (i <- 0 until n) yield 'x(i) * costs(i)
10:    add(Add(c) <= maxCost)
11:    val v = for (i <- 0 until n) yield 'x(i) * values(i)
12:    add(Add(v) >= minValue)
13:  }
14:
15:  def ex1(maxCost: Int = 15, minValue: Int = 19) {
16:    val costs = Seq(3, 4, 5, 7, 9)
17:    val values = Seq(4, 5, 6, 8, 10)
18:    define(maxCost, minValue, costs, values)
19:    if (find) {
20:      val items = (0 until costs.size).filter(i => solution('x(i)) == 1)
21:      println(items)
22:    }
23:  }
```

When the upper bound of the total price is given, the problem to find a combination of items with the maximum total value is called a **Knapsack Problem** (Wikipedia:Knapsack problem).

In these problems, we need to find a solution where the value of the specified variable is maximized (or minimized). Such problems are called a **Constraint Optimization Problem** (COP).

You can solve COP in Copris by specifying the objective variable by 'maximize' (or 'minimize') method. The optimum solution can be found by 'findOpt' method.

The CSP file can be modified as follows.

```
val v = for (i <- 0 until n) yield 'x(i) * values(i)
int('value, 0, values.sum)
add('value === Add(v))
maximize('value)
```

4.7.1 Exercise

1. Find a solution with maximum total value.

- **(Answer)**
  See KnapsackOpt-csp.scala.

```
scala> :load KnapsackOpt-csp.scala
scala> ex1()
```

4.8 4-Queens Problem

Place four chess Queens on 4×4 chessboard such that no two queens attack each other.

In general, the problem of placing n Queens on $n \times n$ chessboard is called an **n-Queens problem** (Wikipedia:Eight queens puzzle).

When we use 0-1 variable x_{ij} for each row i and column j, the 4-queens problem can be formalized as follows.

$$
\begin{aligned}
& x_{ij} \in \{0,1\} \qquad (i = 0\,..\,3,\ j = 0\,..\,3) \\
& x_{i0} + x_{i1} + x_{i2} + x_{i3} = 1 \qquad (i = 0\,..\,3) \\
& x_{0j} + x_{1j} + x_{2j} + x_{3j} = 1 \qquad (j = 0\,..\,3) \\
& x_{01} + x_{10} \leq 1 \\
& x_{02} + x_{11} + x_{20} \leq 1 \\
& x_{03} + x_{12} + x_{21} + x_{30} \leq 1 \\
& x_{13} + x_{22} + x_{31} \leq 1 \\
& x_{23} + x_{32} \leq 1 \\
& x_{02} + x_{13} \leq 1 \\
& x_{01} + x_{12} + x_{23} \leq 1 \\
& x_{00} + x_{11} + x_{22} + x_{33} \leq 1 \\
& x_{10} + x_{21} + x_{32} \leq 1 \\
& x_{20} + x_{31} \leq 1
\end{aligned}
$$

A constraint on the summation of 0-1 variables is called a **Boolean cardinality constraint**. Copris can automatically encode Boolean cardinality constraints to SAT. However, we try to implement the encoding explicitly in the program. There are a number of encodings of Boolean cardinality constraints proposed. Here, we use a pair-wise encoding which will be the simplest way.

The CSP file can be written as follows (QueensBC-csp.scala).

```
 1:  import jp.kobe_u.copris._
 2:  import jp.kobe_u.copris.dsl._
 3:
 4:  def addAtLeastOne(xs: Seq[Term]) {
 5:    add(Or(xs.map(x => x.?)))
 6:  }
 7:  def addAtMostOne(xs: Seq[Term]) {
 8:    for (Seq(x1,x2) <- xs.combinations(2))
 9:      add(! (x1.? && x2.?))
10:  }
11:  def addExactOne(xs: Seq[Term]) {
12:    addAtLeastOne(xs)
13:    addAtMostOne(xs)
14:  }
15:
16:  def define(n: Int) {
17:    init
18:    for (i <- 0 until n; j <- 0 until n)
19:      boolInt('x(i,j))
20:    for (i <- 0 until n) {
21:      val xs = for (j <- 0 until n) yield 'x(i,j)
22:      addExactOne(xs)
23:    }
24:    for (j <- 0 until n) {
25:      val xs = for (i <- 0 until n) yield 'x(i,j)
26:      addExactOne(xs)
27:    }
28:    for (s <- 1 to 2*n-3) {
29:      val xs = for (i <- 0 until n; j <- 0 until n; if s == i+j) yield 'x(i,j)
30:      addAtMostOne(xs)
31:    }
32:    for (s <- 2-n to n-2) {
33:      val xs = for (i <- 0 until n; j <- 0 until n; if s == i-j) yield 'x(i,j)
34:      addAtMostOne(xs)
35:    }
36:  }
```

Here, `x.?` is equivalent to `Ge(x,1)`.

```
scala> :load QueensBC-csp.scala
scala> val n = 4
scala> define(n)
scala> find
scala> for (i <- 0 until n) println((0 until n).map(j => solution('x(i,j))).mkString(" "))
```

You might want to add the following implicit conversions.

```
implicit def term2constraint(x: Term) = x.?
implicit def symbol2constraint(s: Symbol) = Var(s.name).?
```

Then, you can write constraints without '?' methods as follows.

```
boolInt('x)
boolInt('y)
add('x + 'y <= 1)
add('x || 'y)
```

However, these implicit conversions may be a cause of undesired (and unexpected) translation. Care should be taken to use them.

5 Short Summary of Classes and Methods

This section describes a short summary of classes and methods provided by Copris.

5.1 Integer Variable Objects (Var objects)

`Var` creates an integer variable object. The argument specifies its name.

```
scala> val x = Var("x")
x: jp.kobe_u.copris.Var = x
```

When no arguments are given, a new anonymous integer variable object is created.

```
scala> var z = Var()
z: jp.kobe_u.copris.Var = _I1
```

New `Var` object is created by applying an index to an exisiting `Var` object. Integers or character strings can be used as indices, and multiple indices can be given. However, integer variable objects of Copris can not be used as indices.

```
scala> x(1)
res: jp.kobe_u.copris.Var = x(1)
scala> x("a")
res: jp.kobe_u.copris.Var = x(a)
scala> x(1, "a")
res: jp.kobe_u.copris.Var = x(1,a)
```

Symbol of Scala is implicitly converted to a `Var` object.

```
scala> 'x(1)
res: jp.kobe_u.copris.Var = x(1)
```

`Var` is a subclass of `Term` class described later.

5.2 Term Objects

`Term` object is either a `Var` object or one of the following object.

- **Num(*Int*)**
  `Num` object represents a number.

  ```
  scala> Num(314)
  res: jp.kobe_u.copris.Num = 314
  ```

- **Abs(*T*)**
  `Abs` object represents an absolute value.

  ```
  scala> Abs('x)
  res: jp.kobe_u.copris.Abs = Abs(x)
  ```

- **Neg(*T*)**
  `Neg` object represents a unary minus. Unary operator `-` can be used also.

  ```
  scala> - 'x
  res: jp.kobe_u.copris.Neg = Neg(x)
  ```

- **Add(*T1*, ..., *Tn*), Add(Seq(*T1*, ..., *Tn*))**
  `Add` object represents an addition. Binary operator `+` can be used also.

  ```
  scala> 'x + 'y
  res: jp.kobe_u.copris.Add = Add(x,y)
  ```

- **Sub(*T1*, ..., *Tn*), Sub(Seq(*T1*, ..., *Tn*))**
  `Sub` object represents a subtraction. Binary operator `-` can be used also.

  ```
  scala> 'x - 'y
  res: jp.kobe_u.copris.Sub = Sub(x,y)
  ```

- **Mul(*T1*, ..., *Tn*), Mul(Seq(*T1*, ..., *Tn*))**
  `Mul` object represents a multiplication. The multiplier should be an integer constant. Binary operator `*` can be used also.

  ```
  scala> 'x * 2
  res: jp.kobe_u.copris.Mul = Mul(x,2)
  ```

- **Div(*T1*, *T2*), Mod(*T1*, *T2*)**
  `Div` and `Mod` operators represent a division and remainder operations respectively. The divisor should be an integer constant. Binary operators `/` and `%` can be used also.

```
scala> 'x / 3
res: jp.kobe_u.copris.Div = Div(x,3)
```

- **Max(*T1*, ..., *Tn*), Max(Seq(*T1*, ..., *Tn*)), Min(*T1*, ..., *Tn*), Min(Seq(*T1*, ..., *Tn*))**
  `Max` and `Min` object represent a maximum and minimum operations respectively.

```
scala> Max('x+1, 'y+2)
res: jp.kobe_u.copris.Max = Max(Add(x,1),Add(y,2))
```

- **If(*C*, *T1*, *T2*)**
  `If` object represetns an 'if' expression which is equal to *T1* when *C* is true, and equal to *T2* otherwise.

```
scala> If('x > 0, Num(1), Num(0))
res: jp.kobe_u.copris.If = If(Gt(x,0),1,0)
```

5.3 Constraint Objects

- **Eq(*T1*, *T2*), Ne(*T1*, *T2*), Le(*T1*, *T2*), Lt(*T1*, *T2*), Ge(*T1*, *T2*), Gt(*T1*, *T2*)**
  `Eq`, `Ne`, `Le`, `Lt`, `Ge`, and `Gt` objects represent a comparison. Binary operators `===`, `=/=`, `<=`, `<`, `>=`, and `>` can be used also.

```
scala> 'x === 'y
res: jp.kobe_u.copris.Eq = Eq(x,y)
```

  Expression "˜x.?˜" and "˜x.!˜" are equivalent to "˜x ¿= 1˜" and "˜x ¡= 0˜" respectively.

```
scala> 'x.?
res: jp.kobe_u.copris.Eq = Ge(x,1)
scala> 'x.!
res: jp.kobe_u.copris.Le = Le(x,0)
```

- **Alldifferent(*T1*, ..., *Tn*), Alldifferent(Seq(*T1*, ..., *Tn*))**
  `Alldifferent` object represents an alldifferent constraint.

- **Not(*C*)**
  `Not` object represents a negation. Unary operator `!` can be used also.

- **And(*C1*, ..., *Cn*), And(Seq(*C1*, ..., *Cn*))**
  `And` object represents a conjunction (logical AND). Binary operator `&&` can be used also.

- **Or(*C1*, ..., *Cn*), Or(Seq(*C1*, ..., *Cn*))**
  `Or` object represents a disjunction (logical OR). Binary operator `||` can be used also.

- **Imp(*C1*, *C2*), Iff(*C1*, *C2*)**
  `Imp` and `Iff` objects represent an implication and equivalence respectively. Binary operators `==>` and `<==>` can be used also.

- **Xor(*C1*, *C2*)**
  `Xor` object represents an exclusive or. Binary operator `^` can be used also.

5.4 CSP Objects

`CSP` objects represents a Constraint Satisfaction Problem. When you import `jp.kobe_u.copris.dsl._`, the default `CSP` object can be referred by a variable `csp`.

5.4.1 Declaring Integer Variables

Integer variable can be declared by `int` method. Its domain can be given by specifying the lower and upper bounds.

```
scala> int('x, 1, 10)
res: jp.kobe_u.copris.Var = x
```

When it is a constant, one value is given.

```
scala> int('y, 5)
res: jp.kobe_u.copris.Var = y
```

When the domain is non-contiguous, a set can be specified as its domain.

```
scala> int('p(1), Set(2,3,5,7))
res: jp.kobe_u.copris.Var = p(1)
scala> int('p(2), Set(3,7,5,2))
res: jp.kobe_u.copris.Var = p(2)
```

0-1 variable is declared by `boolInt` method. `boolInt(x)` is equivalent to `int(x, 0, 1)`.

```
scala> boolInt('b)
res: jp.kobe_u.copris.Var = b
```

Domain of an integer variable can be shown by `csp.dom` method.

```
scala> csp.dom('x)
res: jp.kobe_u.copris.Domain = Domain(1,10)
scala> csp.dom('p(1))
res: jp.kobe_u.copris.Domain = Domain(TreeSet(2, 3, 5, 7))
```

5.4.2 Adding Constraints

Constraint can be added by `add` method.

```
scala> add('x === 'y * 2)
scala> add('x === 'p(1) + 'p(2))
scala> add('b.? <==> ('p(1) < 'p(2)))
```

Current integer variable declarations and constraints can be shown by `show` method.

```
scala> show
int(x,1,10)
int(y,5,5)
int(p(1),Domain(TreeSet(2, 3, 5, 7)))
int(p(2),Domain(TreeSet(2, 3, 5, 7)))
int(b,0,1)
Eq(x,Mul(y,2))
Eq(x,Add(p(1),p(2)))
Iff(Ge(b,1),Lt(p(1),p(2)))
```

CSP file in Sugar format can be created by `dump` method.

```
scala> dump("output.csp")
```

CNF file in DIMACS format can be created by `dump` method with "cnf" option.

```
scala> dump("output.cnf", "cnf")
```

5.5 Searching Solutions

The `find` method searches the first solution.

```
scala> find
res: Boolean = true
```

When the result is 'true', there exists a solution, and when it is 'false', there exists no solutions. The solution found is stored in `solution` variable.

```
scala> solution
res: jp.kobe_u.copris.Solution = Solution(Map(b -> 0, p(2) -> 5, y -> 5, x -> 10, p(1) -> 5),Map())
```

The 'find' method searches a solution as follows.

- CSP objected is encoded to SAT, and CNF file is created.
- A SAT solver (Sat4j is used as a default solver) is used to find a solution for the generated CNF file.
- A solution of the CNF (if any) is decoded to a solution of CSP.

The name of CNF file generated by Copris can be checked by 'info' command. The file is automatically deleted after exiting Scala.

```
scala> info
res: Map[String,String] = Map(satFile -> /tmp/sugar3235357395765834639.cnf)
```

The value of an integer variable can be obtained by `solution` method.

```
scala> solution('x)
res: Int = 10
scala> solution('x, 'p(1), 'p(2))
res: Seq[Int] = ArrayBuffer(10, 5, 5)
```

The `findNext` method searches the next solution.

```
scala> findNext
res: Boolean = true
scala> solution
res: jp.kobe_u.copris.Solution = Solution(Map(b -> 0, p(2) -> 3, y -> 5, x -> 10, p(1) -> 7),Map())
```

The 'findNext' method searches the next solution as follows.

- An extra condition of negating current solution is added to the CNF file.
- A SAT solver is used to find a solution.
- A solution of the CNF (if any) is decoded to a solution of CSP.

`solutions` method returns the iterator of all solutions.

```
scala> cancel
scala> solutions.foreach(println)
Solution(Map(b -> 0, p(2) -> 5, y -> 5, x -> 10, p(1) -> 5),Map())
Solution(Map(b -> 0, p(2) -> 3, y -> 5, x -> 10, p(1) -> 7),Map())
Solution(Map(b -> 1, p(2) -> 7, y -> 5, x -> 10, p(1) -> 3),Map())
```

To find an optimum solution, specify the objective variable by `maximize` (or `minimize`) method, and use `findOpt` method.

```
scala> cancel
scala> maximize('p(2))
scala> findOpt
res: Boolean = true
scala> solution
res: jp.kobe_u.copris.Solution = Solution(Map(b -> 1, p(2) -> 7, y -> 5, x -> 10, p(1) -> 3),Map())
```

5.6 Others

5.6.1 Switching SAT Solver

SAT solver can be switched by `use` method.

```
scala> use(new sugar.Solver(csp, new sugar.SatSolver2("/usr/local/bin/minisat")))
scala> use(new sugar.Solver(csp, new sugar.SatSolver1("/usr/local/bin/clasp")))
```

Use `SatSolver2` for a SAT solver taking two argument files (such as MiniSat), and use `SatSolver1` for a SAT solver taking one argument file (such as Clasp).

5.6.2 Using PB Solver

You can use PB (Pseudo-Boolean) solver as follows.

```
scala> use(new pb.Solver(csp, new pb.PbSolver("/usr/local/bin/clasp"), "binary"))
```

5.6.3 Using SMT Solver

You can use SMT solver supporting `QF_LIA` theory (such as Z3) by 'use' command.

```
scala> use(new smt.Solver(csp, new smt.SmtSolver("/usr/local/bin/z3")))
```

SMT file can be saved by 'dump' command.

```
scala> dump("file.smt2")
```

5.6.4 Using JSR-331 Solver

You can use JSR-331 solver by 'use' command.

```
scala> use(new jsr331.Solver(csp))
```

You need to put implementation jar files (e.g. `jsr331.choco.jar` and `choco-solver-2.1.5-20120603-with-sources.ja` in `lib/jsr331/` directory.

```
$ scala -cp build/copris-all-v2-2-8.jar:../lib/jsr331/"*"
```

JSR-331 solver implementations are available at http://openrules.com/jsr331/downloads.htm.

5.6.5 Importing XCSP

You can import XCSP file as follows.

```
scala> init
scala> (new loader.XCSPLoader(csp)).load("file.xml")
```

5.6.6 Exporting/Importing Sugar CSP

You can export/import CSP by Sugar CSP format.

```
scala> dump("file.csp")
scala> init
scala> (new loader.SugarLoader(csp)).load("file.csp")
```

5.6.7 Using Multiple CSPs+Solvers

Multiple CSPs (with multiple Solvers) can be simultaneously used in the following way. CSPs and Solvers of `copris1` and `copris2` are independent.

```
 1:  import jp.kobe_u.copris._
 2:  import jp.kobe_u.copris.dsl._
 3:
 4:  val copris1 = new sugar.Sugar()
 5:  val copris2 = new sugar.Sugar()
 6:  using(copris1) {
 7:    int('x, 1, 4)
 8:    add('x < 3)
 9:  }
10:  using(copris2) {
11:    int('x, 1, 4)
12:    add('x >= 3)
13:  }
14:  copris1.solutions.foreach(println)
15:  copris2.solutions.foreach(println)
```

5.6.8 Aviod Using Copris DSL

If you don't want to import Copris DSL methods (that is, to avoid `import jp.kobe_u.copris.dsl._`), you need to describe as follows (ExampleNoDSL-csp.scala).

```
 1:  import jp.kobe_u.copris._
 2:
 3:  val copris = new sugar.Sugar()
 4:  val x = copris.int(Var("x"), 1, 15)
 5:  val y = copris.int(Var("y"), 1, 15)
 6:  val z = copris.int(Var("z"), 1, 15)
 7:  copris.add(x + y + z === 15)
 8:  copris.add(x + y * 5 + z * 10 === 90)
 9:
10:  if (copris.find) {
11:    println(copris.solution)
12:  }
```