Copris による制約プログラミング入門

田村直之

2018-03-21

目次

1 はじめに

この文書では，Scala 言語に埋めこまれた制約プログラミング用 DSL (Domain Specific Language) である Copris を用い，制約プログラミングの入門的な内容を説明する．

2 インストール

1. Java runtime version 1.8 以降をインストールする
2. Scala version 2.11 をインストールする
3. Scala のバージョンをチェックする
```
$ scala -version
Scala code runner version 2.11.8 -- Copyright 2002-2016, LAMP/EPFL
```
Copris は Scala version 2.11 で動作する (他のバージョンでは動作しない)．
4. Copris パッケージをダウンロード
5. Copris パッケージを展開

3 始めての利用

3.1 REPL からの利用

ここでは，以下の例題を考える．

> 1 円硬貨，5 円硬貨，10 円硬貨を合計で 15 枚，それぞれを 1 枚以上持っている．金額の合計は 90 円である．それぞれの硬貨を何枚持っているか?

この問題は，以下の CSP (制約充足問題) として定式化できる．

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

これを Copris を用いて解くことにする．Copris パッケージの build フォルダに移動し，以下のように入力して Copris を起動する．

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

3.1.1 CSP の定義

scala の入力プロンプトが表示されたら，以下のように入力する．

```
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)
```

- 1 行目は Copris 用クラス名の import 宣言である．
- 2 行目は Copris DSL 用のメソッドの import 宣言である．

- 3 行目は変数 x を宣言している (下限 1，上限 15)．シングルクォーテーションから始まる記述 `'x` は Scala における Symbol オブジェクトの記法であるが，Copris DSL により Copris の整数変数 (Var オブジェクト) に暗黙変換される．
- 4 行目と 5 行目は同様に変数 y, z を宣言している．
- 6 行目は制約 $x + y + z = 15$ を追加している．制約中での等号に `===` を用いる点に注意する．add は制約を CSP オブジェクトに追加するためのメソッドである．
- 7 行目は制約 $x + 5y + 10z = 90$ を追加している．`5 * 'y` のようには記述できない点に注意する．整数定数を前に記述したい場合は `Num(5) * 'y` のように書く．

定義した CSP オブジェクトは，変数 csp として参照できる．

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

CSP オブジェクトは，整数変数の列 variables, ブール変数の列 bools, 変数ドメインのマップ dom, 制約の列 constraints から成る．

```
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))
```

show メソッドでも表示できる．

```
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)
```

- ここで Add は加算，Mul は乗算，Eq は等号を表す．

CSP オブジェクトは，変数や制約の追加を行える mutable なオブジェクトとして実装されている．

3.1.2 解の探索

最初の解の探索は find で行う．

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

結果の true は，解が存在することを表す．CSP の解は，solution 変数に代入されている．

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

Solution オブジェクトは，整数変数 (Var オブジェクト) に対する値割当てを表すマップとブール変数 (Bool オブジェクト) に対する値割当てを表すマップから構成される．

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

scala> solution.boolValues
```

```
res: Map[Bool,Boolean] = Map()
```

解における各変数の値は solution メソッドで得ることができる．

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

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

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

次の解の探索は findNext で行う．

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

結果の false は，解が存在しないことを表す．この場合，変数 solution は null になっている．

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

show メソッドを実行すると，確かに条件が追加されていることが分かる．

```
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()))
```

追加された条件を取り消すには cancel を実行する．

```
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)
```

解の一覧は solutions で求めることができる．

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

一度 findNext を実行した後で，全解を求めたい場合，上記のように cancel で追加条件を削除する必要がある．

3.2 スクリプトファイルの作成

CSP を Scala のスクリプトファイルとして定義することもできる．
以下がスクリプトファイルである (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)
```

スクリプトファイルは以下のように :load コマンドを使用して読み込む．

```
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())
```

スクリプトファイルの内容を変更した後，再度読み込みたい場合には，:load の前に init を実行し，いったん CSP の定義を消去する必要がある．

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

以下で示すスクリプトファイルでは，先頭に init を記述し，何度でも読み込めるようにしておく．

3.2.1 練習問題

1. 合計が 105 円の場合を試してみよ．

4 簡単な例題

4.1 コインの問題

最初の問題は，コインによる支払いの問題である．

> 以下のユーロ硬貨を持っているとする (総和は 148 セントになる)．
>
> - 20 セント硬貨 4 枚
> - 10 セント硬貨 4 枚
> - 5 セント硬貨 4 枚
> - 2 セント硬貨 4 枚
>
> 合計がちょうど 93 セントになる組合せはあるだろうか?

この問題は，以下の CSP (制約充足問題) として定式化できる．変数 x_i は i セント硬貨の枚数を表している．

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

CSP を記述したファイルは以下のようになる (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)
```

Copris では 'x(20) のように，添字を用いることができる．添字には整数だけでなく，文字列など Scala の任意のオブジェクトを用いて良い．また 'x(1,"a") などのように複数の添字を用いることもできる．

4.1.1 練習問題

1. 上のコインの問題の解は何通り存在するか?
   - (解答例)

     以下のようにして求めれば良い.

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

2. 硬貨が各 3 枚の場合はどうなるか?
   - (解答例)

     以下のようにして求めれば良い.

```
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 部分和問題

集合 $\{2, 3, 5, 8, 13, 21, 34\}$ の部分集合で，和が 50 になるものはあるか?

この問題は **部分和問題** (Subset sum problem) として知られている問題の例である．部分和問題は NP-完全である (Wikipedia:部分和問題).

これは，以下の CSP (制約充足問題) として定式化できる．

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

CSP を記述したファイルは以下のようになる (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:  }
```

boolInt は 0-1 変数の宣言であり，`boolInt(x)` は `int(x, 0, 1)` と同一である.

また上記プログラムでは，直接 CSP を記述するのではなく，関数 `define(sum: Int)` で和を与えられるようにし

ている．この場合，利用方法は以下のようになる．

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

4.2.1 練習問題

1. 和が 40 の場合はどうなるか?
   - (解答例)

     以下のようにして求めれば良い．

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

4.3 魔方陣

1 から 9 の数字を 3×3 に配置し，各行，各列，各対角線の和がいずれも 15 になるようにせよ．

このような配置は **魔方陣** (Magic square) と呼ばれる (Wikipedia:魔方陣)．

以下のように CSP として定式化できる．

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

ここで Alldifferent は与えられた引数が互いに異なることを表す．すなわち Alldifferent$(x_1, x_2, \ldots, x_n)$ は $x_i \neq x_j$ (for all $i < j$) と同様である．

CSP を記述したファイルは以下のようになる (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 練習問題

1. 3×3 の魔方陣の解はいくつあるか?

- (解答例)

  以下のようにして求めれば良い.

```
scala> solutions.size
```

2. 回転，鏡像で対称な解を除くには，どのような制約条件を追加すれば良いか?
   - (解答例)

     以下のようにして求めれば良い.

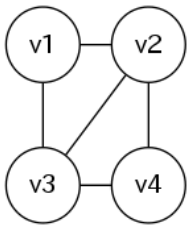
```
scala> cancel
scala> add('x(0,0) < 'x(0,2))
scala> add('x(0,0) < 'x(2,0))
scala> add('x(0,0) < 'x(2,2))
scala> add('x(0,2) < 'x(2,0))
scala> solutions.foreach(println)
```

3. $n \times n$ の魔方陣の CSP を Copris で記述せよ.
   - (解答例)

     MagicSquare-csp.scala 参照.

4.4 グラフ彩色問題

以下のグラフの各頂点に 1 から 3 の数を割り当て，隣接する頂点の異なるようにせよ.



このような問題は **グラフ彩色問題** (GCP; Graph Coloring Problem) と呼ばれる (Wikipedia:グラフ彩色).
上記のグラフは $v_1 = 1, v_2 = 2, v_3 = 3, v_4 = 1$ と彩色すれば良い．また 2 色では彩色できない.
これは，以下の CSP (制約充足問題) として定式化できる.

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

CSP を記述したファイルは以下のようになる (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 練習問題

1. 隣接する頂点の色が単に異なるだけでなく，値が 2 以上離れているようにするには，どのように CSP を記述すれば良いか? (ヒント: Copris では x の絶対値は Abs(x) と記述する)

4.5 数独

次は，数独の問題を解いてみよう (Wikipedia:数独).

CSP を記述したファイルは以下のようになる (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 練習問題

1. http://puzzle.gr.jp の問題を解いてみよう.

4.6 ビンパッキング問題

> 4 人で 8 個の荷物を運ぶとする．各荷物の重さは 3.3kg, 6.1kg, 5.8kg, 4.1kg, 5.0kg, 2.1kg, 6.0kg, 6.4kg である (合計は 38.8kg)．各自の運ぶ荷物の重さの合計が 11kg 以下になるように荷物を割り当てることはできるか?

このような問題は **ビンパッキング問題** と呼ばれる (Wikipedia:ビンパッキング問題).

CSP として定式化するために，各人を行に対応させ，各荷物を列に対応させた行列を考える．行列の各成分は 0 か 1 である．このような行列は **結合行列** (incidence matrix) と呼ばれる．

結合行列の各行，各列に対する制約条件を考え，それを記述する．なお Copris では実数は利用できないため，たとえば 100g 単位で制約条件を記述する必要がある．

4.6.1 練習問題

1. 各列に対する制約条件は何か?
2. 各行に対する制約条件は何か?
3. Copris の CSP として記述せよ．
   - (解答例)

     BinPacking-csp.scala を参照.
4. 10kg 以下の場合はどうなるか?
   - (解答例)

     以下のようにして求めれば良い．

```
scala> :load BinPacking-csp.scala
scala> ex1(max = 100)
```

4.7 ナップサック問題

> 5 つの品物の価格と価値が以下の表に示されるようになっているとする．価格の合計が 15 以下で，価値の合計が 19 以上になる場合はあるか?

品物	価格	価値
品物 0	3	4
品物 1	4	5
品物 2	5	6
品物 3	7	8
品物 4	9	10

Copris で記述した CSP は以下のようになる (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:  }
```

価格の合計の上限のみが与えられ，価値の合計が最大になる場合を探す問題は**ナップサック問題** (Knapsack problem) と呼ばれる (Wikipedia:ナップサック問題).

このような問題では，制約を満たす解のうち，指定された変数の値が最大 (あるいは最小) になるものを見つける必要がある．これらは **制約最適化問題** (COP; Constraint Optimization Problem) と呼ばれる．

Copris で COP を解く場合には，maximize (あるいは minimize) メソッドを用い，引数に最小化 (あるいは最大化) したい変数を指定する．最適解は findOpt メソッドで求められる．

上記の問題の場合，以下のようにプログラムを修正する．

```
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 練習問題

1. 上記で価値の合計が最大になる品物の組合せを求めよ．
   - (解答例)

     KnapsackOpt-csp.scala を参照．

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

4.8 4 クイーン問題

> 4×4 のボード上に，4 つのチェスのクイーンのコマを置く．4 つのクイーンが互いに取られないように配置することはできるか?

一般に $n \times n$ ボード上で n 個のクイーンを配置する場合，**n-クイーン問題** (n-Queens problem) と呼ばれる (Wikipedia:エイト・クイーン).

i 行 j 列にクイーンを配置することを表す 0-1 変数 x_{ij} を用いると，この 4-クイーン問題は以下のように定式化で

きる．

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

0-1 変数の和に関する制約は **ブール基数制約** (Boolean cardinality constraint) と呼ばれる．Copris ではブール基数制約の SAT 符号化を自動的に行うが，ここでは，プログラム内で SAT 符号化を実装する．ブール基数制約の SAT 符号化には様々な方法があるが，ここで実装する方法は最も単純なペア・ワイズ法である．

Copris で記述した CSP は以下のようになる (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:  }
```

ここで x.? は Ge(x,1) と同じである．

```
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(" "))
```

以下のような暗黙変換を加えたくなるかも知れない．

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

この場合，'?' メソッドを使用せずに以下のように制約を記述できる．

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

しかし，これらの暗黙変換により予想していない変換が行われる可能性がある．利用には注意が必要である．

5 クラスとメソッドの簡単なまとめ

ここでは Copris が提供するクラスとメソッドを簡単にまとめる．

5.1 整数変数オブジェクト (Var オブジェクト)

整数変数オブジェクトは Var で生成する．引数にはその名前を与える．

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

名前がない場合は，新しい匿名変数オブジェクトが生成される．

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

Var オブジェクトに添字を与えることで，新しい Var オブジェクトを生成できる．添字には整数や文字列を使用でき，また複数与えても良い．ただし，添字に Copris の整数変数を用いることはできない．

```
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)
```

Scala の Symbol は，Var オブジェクトに暗黙変換される．

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

以下のように日本語を用いても良い．

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

Var オブジェクトは後述の項 (Term) オブジェクトの一種である．

5.2 項オブジェクト (Term オブジェクト)

整数変数オブジェクトの他の項オブジェクトには以下のものがある．

- Num(*Int*)

  Num オブジェクトは数を表す．

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

- Abs(*T*)

  Abs オブジェクトは絶対値を表す．

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

- Neg(*T*)

  Neg オブジェクトは単項マイナスを表す．- 演算子も利用できる．

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

- Add(*T1*, . . . , *Tn*), Add(Seq(*T1*, . . . , *Tn*))

  Add オブジェクトは加算を表す．+ 演算子も利用できる．

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

- Sub(*T1*, . . . , *Tn*), Sub(Seq(*T1*, . . . , *Tn*))

  Sub オブジェクトは減算を表す．- 演算子も利用できる．

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

- Mul(*T1*, . . . , *Tn*), Mul(Seq(*T1*, . . . , *Tn*))

  Mul オブジェクトは乗算を表す．乗数は整数でなければならない．* 演算子も利用できる．

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

- Div(*T1*, *T2*), Mod(*T1*, *T2*)

  Div, Mod オブジェクトは除算, 剰余を表す．除数は整数でなければならない．/, % 演算子も利用できる．

```
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, Min オブジェクトは最大値, 最小値を表す．

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

- If(*C*, *T1*, *T2*)

  If オブジェクトは if 式を表す．*C* が真の時 *T1*, 偽の時 *T2* の値を取る．

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

5.3 制約オブジェクト (Constraint オブジェクト)

- Eq(*T1*, *T2*), Ne(*T1*, *T2*), Le(*T1*, *T2*), Lt(*T1*, *T2*), Ge(*T1*, *T2*), Gt(*T1*, *T2*)

  Eq, Ne, Le, Lt, Ge, Gt オブジェクトは比較を表す．===, =/=, <=, <, >=, > の演算子も利用できる．

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

  x.? および x.! は，それぞれ x >= 1 および x <= 0 と同一である．

```
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 オブジェクトは *T1* から *Tn* が互いに異なることを表す．

- Not(*C*)

  Not オブジェクトは否定を表す．演算子 ! も利用できる．

- And(*C1*, . . . , *Cn*), And(Seq(*C1*, . . . , *Cn*))

  And オブジェクトは連言 (論理積) を表す．演算子 && も利用できる．

- Or(*C1*, . . . , *Cn*), Or(Seq(*C1*, . . . , *Cn*))

  Or オブジェクトは選言 (論理和) を表す．演算子 || も利用できる．

- Imp(*C1*, *C2*), Iff(*C1*, *C2*)

  Imp, Iff オブジェクトは含意, 同値を表す．演算子 ==>, <==> も利用できる．

- Xor(*C1*, *C2*)

  Xor オブジェクトは排他的論理和を表す．演算子 ^ も利用できる．

5.4 CSP オブジェクト

CSP オブジェクトは，制約充足問題を表すオブジェクトである．jp.kobe_u.copris.dsl._ を import した場合，デフォールトの CSP オブジェクトを変数 csp として参照できる．

5.4.1 整数変数の宣言

整数変数は int メソッドで宣言する．通常は，下限値と上限値を与える．

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

下限値と上限値が一致する場合は，1 引数で良い．

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

ドメインが連続値でない場合は，集合を与える．集合の要素の順序は任意で良い．

```
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 変数は boolInt で宣言する．boolInt(x) は int(x, 0, 1) と同一である．

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

変数のドメインは，csp.dom メソッドで確認できる．

```
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 制約の追加

制約の追加は add で行う．

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

現時点での変数宣言と制約は show で確認できる．

```
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)))
```

Sugar の CSP ファイルは，dump コマンドで作成できる．

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

DIMACS 形式の CNF ファイルを作成する場合は，dump コマンドに “cnf” オプションを指定する．

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

5.5 解の探索

最初の解の探索は find で行う．

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

結果が true なら解が存在し，false なら存在しない．見つかった解は solution で表示される．

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

変数を solution への引数として与えれば，値が得られる．

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

find メソッド中では，以下が実行されている．

- CSP オブジェクトを SAT 符号化し，CNF ファイルを作成
- 作成された CNF に対し，SAT ソルバーによる解探索を実行 (デフォールトでは Sat4j が用いられる)
- SAT ソルバーの発見した解を CSP の解に逆符号化 (復号化)

なお，Copris が生成する CNF ファイル名は info コマンドで調べることができる．このファイルは Scala の実行終了時に自動的に消去される．

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

次の解は findNext で求める．

```
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())
```

findNext メソッド中では，以下が実行されている．

- 現在の解の否定を表す条件を CNF ファイルに追加
- 追加した CNF に対し，SAT ソルバーによる解探索を実行
- SAT ソルバーの発見した解を CSP の解に逆符号化 (復号化)

全解を求めるには solutions を用いる．

```
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())
```

ある変数を最大化，最小化するには maximize, minimize で変数を指定し，findOpt で最適解を求める．

```
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 その他

5.6.1 SAT ソルバーの切り換え

SAT ソルバーは use で切り換える．

```
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")))
```

2 引数を取る SAT ソルバー (結果が別ファイル) には SatSolver2 を用い，1 引数を取る SAT ソルバー (結果が標準出力) には SatSolver1 を用いる．

5.6.2 PB ソルバーの利用

PB (Pseudo-Boolean) ソルバーを利用するには，以下のようにする．

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

5.6.3 SMT ソルバーの利用

SMT ソルバー (Z3 など QF_LIA 理論が利用可能なもの) を使用したい場合，use コマンドで切り換える．

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

SMT ファイルは dump コマンドで保存できる．

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

5.6.4 JSR-331 ソルバーの利用

JSR-331 ソルバー も利用できる．

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

利用するには，JSR-331 を実装した jar ファイル (jsr331.choco.jar および choco-solver-2.1.5-20120603-with-sources. など) を lib/jsr331/ ディレクトリにコピーする必要がある．

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

JSR-331 ソルバーの実装は http://openrules.com/jsr331/downloads.htm からダウンロードできる．

5.6.5 XCSP ファイルのインポート

CSP ソルバー競技会で利用された XCSP 形式のファイルをインポートするには，以下のようにする．

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

5.6.6 Sugar CSP ファイルのエクスポート・インポート

定義した CSP は Sugar の形式 で，エクスポート・インポートができる．

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

5.6.7 複数の CSP+Solver の利用

以下のようにして，複数の CSP+Solver を同時に利用できる．copris1 と copris2 の CSP と Solver は完全に独立している．

```
 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 Copris DSL を用いない

Copris DSL のメソッドを import したくなければ (つまり `import jp.kobe_u.copris.dsl._` をしたくなければ)，以下のように記述する (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:  }
```