



Scala上の制約プログラミング用ドメイン特化言語 Coprismについて

田村, 直之
丹生, 智也
番原, 睦則

(Citation)

コンピュータソフトウェア, 29(4):114-129

(Issue Date)

2012

(Resource Type)

journal article

(Version)

Version of Record

(Rights)

ここに掲載した著作物の利用に関する注意 本著作物の著作権は日本ソフトウェア科学会に帰属します。本著作物は著作権者である日本ソフトウェア科学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」に従うことをお願いいたします。

Notice for the use of this material: The copyright of this material is retained by t...

(URL)

<https://hdl.handle.net/20.500.14094/90002833>



Scala 上の制約プログラミング用ドメイン特化言語 Copris について

田村 直之 丹生 智也 番原 睦則

本論文では、オブジェクト指向言語と関数型言語が融合された Scala 言語上に、制約プログラミング用のドメイン特化言語 (DSL; Domain-Specific Language) として開発した Copris (Constraint Programming in Scala) について述べる。Copris は、Scala のプログラマが簡潔に制約充足問題 (CSP) や制約最適化問題 (COP) を記述できることを目的に設計したものであり、既存の Java 上の制約プログラミング用 API である JSR-331 等よりも高い記述力を提供している。バックエンドの制約ソルバーとしては、国際制約ソルバー競技会で 2 年連続グローバル制約部門で優勝した SAT 型制約ソルバー Sugar を用いており、制約ソルバーとしての性能も高い。本論文では、Copris の設計について説明した後、いくつかのプログラミング例を紹介することで、Copris を用いたソフトウェア開発の利点を明らかにする。

In this paper, we explain Copris (Constraint Programming in Scala) system which is developed as a Domain-Specific Language (DSL) for constraint programming embedded in Scala programming language. Copris is designed to help Scala programmers to be able to easily solve Constraint Satisfaction Problems (CSP) and Constraint Optimization Problems (COP), and offers richer description power than existing CSP languages, such as JSR-331, a standardized constraint programming API for Java. Copris also provides a high performance constraint solving since constraint solver Sugar is used as its backend which won at global constraint categories of the international CSP solver competitions in two consecutive years. In this paper, we explain the effectiveness of using Copris by showing some example programs after describing the design overview of Copris DSL.

1 はじめに

制約充足問題 (CSP; Constraint Satisfaction Problem) は与えられた制約を満たす解を探索する問題であり、**制約最適化問題** (COP; Constraint Optimization Problem) は制約を満たす解のうち与えられた目的関数に関する最適解を探索する問題である [30] [40] [3] [7]。人工知能分野等で生じる多くの組合せ問題は、制約充足問題あるいは制約最適化問題として定式化できる。

Domain-Specific Language Copris for Constraint Programming in Scala.

Naoyuki Tamura, Mutsunori Banbara, 神戸大学情報基盤センター, Information Science and Technology Center, Kobe University.

Tomoya Tanjo, 新領域融合研究センター, Transdisciplinary Research Integration Center.

コンピュータソフトウェア, Vol.29, No.4 (2012), pp.114–129. [ソフトウェア論文] 2011 年 11 月 18 日受付。

なお、以下では特別に区別する必要がない限り両者をまとめて CSP と呼ぶ。

制約プログラミング (constraint programming) システムは CSP を取り扱うシステムであり、CSP を記述するための**制約モデリング言語** (constraint modeling language) と、CSP の解を探索する**制約ソルバー** (constraint solver) とからなる。

制約プログラミングシステムには、専用の制約モデリング言語で CSP を記述するタイプのものと、Prolog, C++, Java 等の汎用プログラミング言語の構文で CSP を記述するタイプのものがある。前者には OPL [42], MiniZinc [26], XCSP^{†1} などがあり、後者には Prolog III (Prolog) [9], clp(FD) (Prolog) [8], MPC (Haskell) [31], IBM ILOG CP

^{†1} <http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html>

(C++, Java)^{†2}, Gecode (C++)^{†3}, Choco (Java) [39], Cream (Java)^{†4}, Numberjack (Python) [14] などがある。

専用の制約モデリング言語を用いる場合、高機能的な構文の導入により、より簡潔に CSP を記述できる可能性がある。また MiniZinc や XCSP のように、複数の制約ソルバーが入力言語として対応している場合、使用する制約ソルバーを切換えることができ利便性が高い。特に XML 構文による XCSP は国際制約ソルバー競技会 [21] で採用され、多くの制約ソルバーに対する入力言語として利用できる。しかし、汎用プログラミング言語による記述ではないため、GUI などの他プログラムとの連携を行いにくいという欠点がある。

一方、汎用プログラミング言語中で CSP を記述する場合、ライブラリの利用などを含め既存言語の機能をすべて利用でき、柔軟な処理が可能になる。また、インターフェイスを適切に設計すれば複数の制約ソルバーを切換えることも可能である。しかし、既存言語の構文の枠内での記述であり、言語によっては簡潔に CSP を記述できない。例として、Java 上での制約プログラミング API として提案されている JSR-331 (Java Specification Request 331)^{†5} での CSP 記述例を図 1 に示す。JSR-331 は、Java Community Process において 2010 年度の “The Most Innovative JSR” を受賞しており、Java 上での制約プログラミングの試みとして評価されているが、比較演算子を文字列で指定しているなど、自然な記述を実現できていない。

しかし、柔軟かつ拡張可能な構文を持ったプログラミング言語を採用すれば、記述性・可読性の高い制約モデリング言語を汎用プログラミング言語内に埋め込んだ**ドメイン特化言語** (DSL; Domain-Specific Language) [22] として実現できると考えられる。そのような DSL を実現可能な言語としては Ruby, Python, Scala 等が候補となる [12]。これらの言語を用いれば、

```
1: Var x = variable("X", 1, 10);
2: Var y = variable("Y", 1, 10);
3: Var z = variable("Z", 1, 10);
4: Var r = variable("R", 1, 10);
5: Var[] vars = { x, y, z, r };
6: post(x,"<",y);
7: post(z,">",4);
8: post(x.plus(y),"=",z);
9: postAllDifferent(vars);
10: int[] coef1 = { 3, 4, -5, 2 };
11: post(coef1,vars,">",0);
12: post(vars,">=",15);
13: int[] coef2 = { 2, -4, 5, -1 };
14: post(coef2,vars,">",x.multiply(y));
```

図 1 Java での制約プログラミング例 (JSR-331)

```
1: val x = int('x', 1, 10)
2: val y = int('y', 1, 10)
3: val z = int('z', 1, 10)
4: val r = int('r', 1, 10)
5: val vars = Vec(x, y, z, r)
6: add(x < y)
7: add(z > 4)
8: add(x + y === z)
9: add(Alldifferent(vars))
10: add(Vec(3,4,-5,2) :* vars > 0)
11: add(vars.sum >= 15)
12: add(Vec(2,-4,5,-1) :* vars > x*y)
```

図 2 提案する Copris DSL での制約プログラミング例

元の言語の構文等を拡張することなく、制約モデリング言語に適した構文や機能を持たせた**埋込み DSL** (embedded DSL) が実現可能である。本論文では、静的型付けにより型安全なプログラムを作成できる点、JVM (Java Virtual Machine) 上での効率の良い実行が可能な点から Scala 言語 [28][27] に着目し、Scala 上の制約プログラミング用 DSL である Copris (Constraint Programming in Scala) を提案する。提案 DSL を用いて前述の CSP を記述した例は図 2 のようになり、図 1 と比較して記述性・可読性が向上している (:* はベクトルの内積を表す演算子である)。

制約ソルバーとしては、SAT 型制約ソルバーである Sugar^{†6} を用いる [37][36]。Sugar は、XCSP および Lisp 風の制約モデリング言語で記述した CSP を入力とし、それを命題論理の充足可能性判定問題であ

^{†2} <http://www.ilog.com/products/cp/>

^{†3} <http://www.gecode.org>

^{†4} <http://bach.istc.kobe-u.ac.jp/cream/>

^{†5} <http://jcp.org/en/jsr/detail?id=331>

^{†6} <http://bach.istc.kobe-u.ac.jp/sugar/>

る SAT に符号化した後、外部の SAT ソルバーを用いて元の CSP の解を求めるシステムである。国際制約ソルバー競技会の複数部門で優勝するなど、非常に高性能な制約ソルバーであるが、独立して動作するシステムとして設計されており、他のプログラムから利用することが難しかった。また、ループなどの制御構造や、言語中から操作できる配列等のデータ構造がなく、大規模な CSP を直接記述することは実質的に不可能だった。

本論文で提案する Copris を利用すれば、自然な制約の記述、全解の列挙、動的な制約の追加、他プログラムとの連携などが簡単に行え、Sugar を利用したプログラムの開発が容易になることが期待できる。なお Copris という名称は、Scala 上の埋込み DSL に対して、埋込み DSL を実装したシステムとの双方について用いる。特に DSL を指す場合は Copris DSL と呼ぶ。

以下では、第 2 節で SAT 型制約ソルバー Sugar について述べ、第 3 節で Scala 言語の特徴、特に DSL を実装するための機能について紹介する。第 4 節では、本論文で提案する制約プログラミング用 DSL である Copris の各クラスと API について説明し、第 5 節で、Copris の Sugar 用インターフェイスについて述べる。また付録において、Copris を用いたプログラム例を紹介する。なお Copris は以下の Web ページでオープンソースソフトウェアとして公開している。

<http://bach.istc.kobe-u.ac.jp/copris/>

SAT ソルバーや SAT 型システムなどの SAT 技術に関する説明は本論文の範囲外である。文献[6][17][25][38][18][13][15][23][2][41]などを参照されたい。

2 SAT 型制約ソルバー Sugar

2.1 Sugar の概要

Sugar は、入力として与えられた整数の有限領域上の CSP を SAT に符号化した後、外部の SAT ソルバーを用いて SAT の解を求め、それを元の CSP の解に変換する SAT 型システムである (図 3 参照)。2008 年および 2009 年の国際制約ソルバー競技会[21]の全グローバル制約部門で優勝するなど、優れた性能を示している[37][36]。

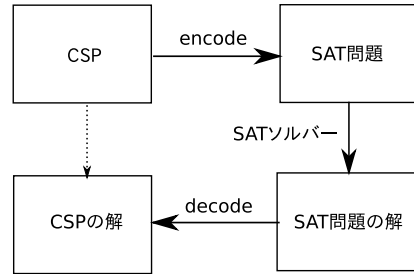


図 3 SAT 型制約ソルバー Sugar

SAT 符号化には **順序符号化** (order encoding) と呼ばれる新しい方法を用いている[35]。順序符号化は、各整数変数 x とそのドメインの各整数値 i に対し $x \leq i$ を意味する命題変数 $p(x \leq i)$ を割当て、これらの命題変数を用いて制約を符号化する方法である。1994 年に Crawford と Baker によってジョブショップ・スケジューリング問題に利用され[10]、論文[34][35]で、より一般の制約に適用できるよう拡張された。順序符号化は、広範囲の問題に対して良い性能を示し、オープンショップ・スケジューリング問題[35][33]、ジョブショップ・スケジューリング問題[20][19]、2次元ストリップパッキング問題[32]、組合せテストケース生成[1]などで未解決だった問題の解決に成功している。また、CSP の SAT 符号化法として、理論的にも他の符号化より優れていることが示されている[29]。

Sugar のシステムは以下の構成要素から成る。

- (1) 与えられた CSP を SAT に符号化するプログラム。Java で記述されている。
- (2) 符号化した SAT を解く (外部の) SAT ソルバー。MiniSat [11], PicoSAT [5], GlueMiniSat [24] などが利用できる。
- (3) SAT の解を CSP の解に変換するプログラム。Java で記述されている。
- (4) 上記のプログラムを順に呼出し、与えられた CSP の解を出力するプログラム。Perl で記述され、制約最適化問題への対応なども行っている。

2.2 Sugar の制約モデリング言語

Sugar で用いている制約モデリング言語は、XCSP

の略記法 (abridged notation) に準拠しているが、XML ではなく Lisp 風の記法を採用している。ただし、XCSP から Sugar の CSP への変換プログラムも用意しており、XML 形式と Lisp 形式のどちらの入力にも対応可能である。

Sugar で利用できる制約には、算術・比較・論理演算で構成されている**内包的制約** (intensional constraint), 制約を満足する (あるいは違反する) 点集合が陽に与えられている**外延的制約** (extensional constraint), そして alldifferent 等に代表される**グローバル制約** (global constraint) がある。

内包的制約では、論理演算として and (論理積), or (論理和), xor (排他的論理和), iff (同値), not (否定), true (真), false (偽) が使用でき、比較演算として eq (=), ne ($\neq$), ge ($\geq$), gt ($>$), le ($\leq$), lt ($<$) が、算術演算として add (加算), sub (減算), neg (マイナス), mul (乗算), div (除算), mod (剰余), pow (べき乗), abs (絶対値), min (最小値), max (最大値), if (if 式) が利用できる。

また、グローバル制約としては、与えられたすべての変数が互いに異なることを表す alldifferent 制約、資源制約を表す cumulative 制約、リスト要素を表す element 制約、線形和を表す weightedsum 制約のほか、disjunctive, global_cardinality, global_cardinality_with_costs, nvalue, count, lex_less, lex_lesseq などが利用できる。各グローバル制約の定義については数百種類のグローバル制約が収集されている Global Constraint Catalog の Web サイト^{†7} を参照されたい。

2.3 Sugar の制約モデリング例

図 4 に、Sugar の制約モデリング言語で図 1 の CSP を記述した例を示す。ベクトルの内積などの演算がサポートされていないため直接的ではあるが、自然な記述となっている。

この CSP を記述したファイル file.csp を入力とし、Sugar プログラムを実行した例を下に示す。

```
$ sugar file.csp
s SATISFIABLE
```

```
1: (int x 1 10)
2: (int y 1 10)
3: (int z 1 10)
4: (int r 1 10)
5: (< x y)
6: (> z 4)
7: (= (+ x y) z)
8: (alldifferent x y z r)
9: (> (+ (* 3 x) (* 4 y) (* -5 z) (* 2 r)) 0)
10: (>= (+ x y z r) 15)
11: (> (+ (* 2 x) (* -4 y) (* 5 z) (* -1 r))
12:      (* x y))
```

図 4 Sugar での制約プログラミング例

```
a x      4
a y      6
a z     10
a r      9
a
```

1 行目が実行コマンドの入力、2 行目以降が Sugar の出力である。2 行目は与えられた CSP が充足可能であることを表し、3 行目以降は各変数への値割当て結果を表す。最後の a のみの行は値割当て出力の最後を示す。

このように Sugar は独立したプログラムとして設計されており、他のプログラムから Sugar を利用するためには、CSP ファイルの作成、Sugar プログラムのプロセス起動、出力の解析などが必要となり、簡便な利用は難しかった。

また、ループなどの制御構造や、言語中から操作できる配列等のデータ構造がなく、大規模な CSP を直接的に記述するのは実際上不可能だった。このため大規模な CSP を記述する際には、CSP を生成するプログラムを作成し実行するしかなく、この点も Sugar の利用を難しくしていた。

3 Scala の概要と特徴

Scala^{†8} は、Martin Odersky により設計された比較的新しいプログラミング言語である [28][27]。言語上の特徴としては、関数型言語とオブジェクト指向言語の融合、強力な型推論、高階関数、不変コレクション、アクターによる並行計算などが挙げられる。

^{†7} <http://www.emn.fr/z-info/sdemasse/gccat/>

^{†8} <http://www.scala-lang.org>

処理系としては、JVM (Java Virtual Machine) へのコンパイラとインタラクティブな実行環境 (REPL; Read Eval Print Loop) が用意されている。Java との親和性は高く、Java のクラス・ライブラリをそのまま利用できる。Twitter の分散 DB フレームワークに用いられているなど、実用的なソフトウェアの開発にも利用されており、現在注目を集めているプログラミング言語のひとつである。

さらに Scala は、関数型言語およびオブジェクト指向言語としての高度な記述能力、リスト、マップ、集合等の豊富なコレクションフレームワーク、演算子の多重定義や柔軟な構文、オブジェクトのメソッドのインポート機能など、埋込み DSL を実装するために適した機能を数多く備えている [12]。以下では、DSL 実装に適した機能のいくつかを取り上げ簡単に紹介する。

ケースクラス (case class) は、`equals`, `hashCode`, `toString` およびファクトリクラスによるオブジェクト生成メソッドなどが自動的に定義され、またパターンマッチに利用できるなど、CSP で利用するデータ構造を定義するのに非常に適している。たとえば、抽象クラス `Term` の子クラスとしてケースクラス `Num`, `Add` を定義し、整数と加算からなる再帰的なデータ構造を表現している例を次に示す。

```
abstract class Term
case class Num(value: Int) extends Term
case class Add(x0: Term, x1: Term) extends Term
```

以下に Scala の REPL での実行例を示す。簡潔に $(1 + 2) + 3$ に相当するデータ構造を表現できていることがわかる。

```
scala> Add(Add(Num(1), Num(2)), Num(3))
res0: Add = Add(Add(Num(1),Num(2)),Num(3))
```

さらに**演算子の多重定義**を導入すれば、より自然な表記を実現できる。たとえば、以下のように抽象クラス `Term` にメソッド `+` を定義する。

```
abstract class Term {
  def + (that: Term) = Add(this, that)
}
```

この定義により挿入演算子 `+` が利用可能になる。

```
scala> (Num(1) + Num(2)) + Num(3)
res1: Add = Add(Add(Num(1),Num(2)),Num(3))
```

このように表記できるのは、`obj.method(arg)` とい

うメソッド呼出しを `obj method arg` のようにピリオドおよびカッコを省略して記述できることによる。この場合のメソッド適用の優先度は、メソッド名の最初の文字によって定まっており、数式記述での通常の優先度と一致している (たとえば $x + y * z$ は $x + (y * z)$ と解釈される)。なお、同一優先度のメソッドは基本的には左結合的に解釈される (ただし、コロンで終わるメソッド名は右結合的となる)。

また、あるデータ型を別のデータ型に変換する**暗黙変換** (implicit conversion) の定義が可能である。たとえば以下のように整数を `Num` オブジェクトに変換する関数を暗黙変換として定義する。

```
implicit def int2num(value: Int) = Num(value)
```

Scala コンパイラは、型推論で型誤りが生じる場合、暗黙変換で型誤りが解消できるならばその変換を自動的に適用する。たとえば、以下で整数 `1` は `Num` に暗黙変換されている。

```
scala> 1 + Num(2)
res2: Add = Add(Num(1),Num(2))
```

引数のデータ型を $\Rightarrow$ `Type` のように指定することで、値呼び (call-by-value) ではなく**名前呼び** (call-by-name) を実現できる。名前呼びの場合、引数の評価は関数内で行われるため、マクロ定義と同様に新しい制御構造の導入などを実現できる。

シングルトンオブジェクト (singleton object) と、その**オブジェクトからのインポート**は、オブジェクト中のメソッドを直接利用可能にする。たとえばオブジェクト `Obj` を次のように定義する。

```
object Obj {
  def double(x: Term) = Add(x, x)
}
```

`import Obj._` と記述すれば `Obj` オブジェクトのメソッドがインポートされ、`Obj.double(x)` ではなく単に `double(x)` と記述できる。

Scala の `DynamicVariable` クラスは、変数の動的なスコープを模倣するために用いることができる。たとえば以下のようにオブジェクト `dsl` を定義する。

```
object dsl {
  val dynVar = new DynamicVariable[CSP](null)
  def using(csp: CSP)(block: => Unit) =
    dynVar.withValue(csp) { block }
  def add(c: Constraint) = dynVar.value.add(c)
```

表 1 Copris の主要なクラス

クラス名等	内容	親クラス等
Term	項を表すクラス	
Vec	項のベクトルを表すクラス	
Matrix	項の行列を表すクラス	
Constraint	制約を表すクラス	
CSP	CSP を表すクラス	
AbstractSolver	CSP ソルバーを表す抽象クラス	
DefaultSolver	デフォルト CSP ソルバーを生成するファクトリーオブジェクト	
CoprisTrait	Copris DSL 用 API を定義したトレイト	
Copris	Copris DSL 用 API を実装したクラス	CoprisTrait
dsl	Copris DSL を実装したオブジェクト	CoprisTrait
sugar.SatSolver	Sugar ソルバーで利用する SAT ソルバーを表す抽象クラス	
sugar.Solver	Sugar ソルバーを実装したクラス	AbstractSolver
sugar.Sugar	Sugar DSL 用 API を実装したクラス	CoprisTrait
sugar.dsl	Sugar DSL を実装したオブジェクト	CoprisTrait

```

}

dsl オブジェクトのメソッドをインポートすること
で、以下のように動的スコープを導入した処理が可能
になる。

```

```

using(csp1) {
  add(...) // csp1.add(...) が実行される
}
using(csp2) {
  add(...) // csp2.add(...) が実行される
}

```

その他、可変長引数の使用、apply メソッドを用いたオブジェクト操作 (obj(arg) は obj.apply(arg) と解釈される)、代入演算のオーバーライドなど Scala 言語の様々な特徴を適切に利用することにより、開発者は自然な構文を導入した DSL を実装できる。

4 制約プログラミング用 DSL Copris

本節では、Scala に埋め込まれた制約プログラミング用 DSL である Copris について説明する。

Copris のクラスは、CSP を表現するためのクラス、CSP ソルバーを実現するためのクラス、DSL を実現するためのクラス、Sugar インターフェイスのためのクラスに分類される。以下に、それぞれの設計方針を述べる。

CSP 表現用クラス: CSP 記述に使用する算術・比較・論理演算およびグローバル制約については、XCSP で規定しているものを対象とする。これは、XCSP が国際制約ソルバー競技会で採用さ

れ、多くの CSP ソルバーが対応しているためである。ただし、最小化あるいは最大化する目的変数の指定などの拡張を行い、制約最適化問題も表現できるようにする。

CSP ソルバー用クラス: 利用する CSP ソルバーとしては Sugar を想定するが、Sugar だけでなくその他の様々な制約ソルバーを使用できるように設計する。特に JSR-331 に準拠した制約ソルバーが利用可能となるように考慮する。すなわち、最初の解の探索だけでなく、全解の探索や最適解の探索も可能なように設計する。

DSL 用クラス: DSL 用クラスでは、Scala 言語の特徴を利用し、プログラマが自然に CSP を記述し、制約ソルバーを利用できる機能を提供する。

Sugar 用クラス: CSP ソルバー用クラスを実装した jp.kobe.u.copris.sugar.Solver クラスを提供し、SAT ソルバーの切換えなど Sugar 特有の機能を DSL から利用可能にする。

表 1 に Copris の主要なクラスとその説明を示す。なお、各クラスは jp.kobe.u.copris パッケージに属するものとする。

4.1 CSP 表現用クラス

4.1.1 項と制約

CSP における項 ($x + y$ 等の数式) および制約 ($(x > 0) \vee (y > 0)$ 等の論理式) を Scala 中で表現するために、項と制約を表すクラスを図 5 のように

```

1: abstract class Expr
2: abstract class Term extends Expr
3: abstract class Constraint extends Expr
4: abstract class GlobalConstraint
5:   extends Constraint
6:
7: case class Num(value: Int) extends Term
8: case class Var(name: String, is: String*)
9:   extends Term {
10:   def apply(is1: Any*) =
11:     Var(name, is++is1.map(_.toString): _*)
12: }
13: case class Neg(x0: Term) extends Term
14: case class Add(xs: Term*) extends Term
15: object Add {
16:   def apply(xs: Iterable[Term]): Add =
17:     Add(xs.toSeq: _*)
18: }
19:
20: case class Bool(name: String, is: String*)
21:   extends Constraint
22: case class Not(c0: Constraint)
23:   extends Constraint
24: case class And(cs: Constraint*)
25:   extends Constraint
26: case class Alldifferent(xs: Term*)
27:   extends GlobalConstraint

```

図 5 項と制約クラスの定義 (一部)

定義する。Expr クラスは Term および Constraint クラスの親クラスであり、それぞれ項と制約を表す。GlobalConstraint は Alldifferent 等のグローバル制約のクラスである。

7 行目から 18 行目は項のクラスの定義である。例えば $x + 1$ に相当する項は以下のように記述できる (import jp.kobe_u.copris._ の実行を仮定している)。

```
scala> Add(Var("x"), Num(1))
res3: jp.kobe_u.copris.Add = Add(x,1)
```

命題変数を表すクラス Var は、変数名 name に加え文字列のリスト is を添字として指定可能である (8 行目)。また、すでにある Var オブジェクトに任意の添字を追加した新しい Var オブジェクトを作成できるように、apply メソッドを定義している (10 行目)。これは、Sugar で整数変数の配列が利用できない問題点を解消するために導入している。

```
scala> val x = Var("x")
x: jp.kobe_u.copris.Var = x
scala> x(1,2) // x.apply(1,2) と同じ
```

```
res4: jp.kobe_u.copris.Var = x(1,2)
```

加算を表すクラス Add は Scala の可変引数指定により複数引数を取ることができる (14 行目)。また、コンパニオンオブジェクト (companion object) 中に apply メソッドを定義しており、項のリスト等を引数とすることが可能である (16 行目)。

```
scala> Add(Var("x"), Var("y"), Var("z"))
res5: jp.kobe_u.copris.Add = Add(x,y,z)
scala> Add((1 to 3).map(i => Var("x")(i)))
res6: jp.kobe_u.copris.Add=Add(x(1),x(2),x(3))
```

図中の Neg (マイナス), Add (加算) 以外に, XCSP の各算術演算に対応し Abs (絶対値), Sub (減算), Mul (乗算), Div (除算), Mul (乗算), Div (除算), Max (最大値), Min (最小値), If (if 式) を用意している。

20 行目から 27 行目は制約クラスの定義である。図中の Not (否定), And (論理積) 以外に, XCSP の各論理演算・比較演算に対応し Or (論理和), Imp (含意), Xor (排他的論理和), Iff (同値), Eq (=), Ne (≠), Le (≤), Lt (<), Ge (≥), Gt (>) を用意している。また Bool クラスで添字を指定可能な点, And クラスで複数引数を指定可能な点は, それぞれ Var クラスおよび Add クラスと同様である。たとえば $x + 1 \leq y$ に相当する制約は以下のように記述できる。

```
scala> Le(Add(Var("x"), Num(1)), Var("y"))
res7: jp.kobe_u.copris.Le = Le(Add(x,1),y)
```

また, 26 行目の Alldifferent を含め, XCSP で利用可能なグローバル制約はすべて用意している。

なお, XCSP (および Sugar) では外延的制約も利用可能だが, 現状の Copris では対応していない。

4.1.2 演算子の定義

式の自然な記述を可能にするため, 項および制約に対する演算子を定義する。以下は $(x > 0) \vee (y > 0)$ を記述した例である。

```
scala> Var("x") > 0 || Var("y") > 0
res8: jp.kobe_u.copris.Or = Or(Gt(x,0),Gt(y,0))
```

ただし ==, !=, => などは Scala 言語で特別の意味を持つため, 等価には ===, 非等価には !==, 含意には ==>, 同値には <==> を用いることにする。

さらに以下のように暗黙変換を定義すれば, Scala のシンボル (’で始まる記号) を CSP の整数変数として利用でき, より簡潔な記述が可能となる。

```

1: case class CSP(
2:   var variables: Seq[Var] = Seq(),
3:   var bools: Seq[Bool] = Seq(),
4:   var dom: Map[Var, Domain] = Map(),
5:   var constraints: Seq[Constraint] = Seq()) {
6:   def int(x: Var, lo: Int, hi: Int): Var
7:   def add(c: Constraint): Unit
8:   def minimize(x: Var): Var
9: }

```

図 6 Domain と CSP クラスの定義 (一部)

```

scala> implicit def symbol2var(s: Symbol) =
      Var(s.name)
scala> 'x > 0 || 'y > 0
res9: jp.kobe_u.copris.Or = Or(Gt(x,0),Gt(y,0))
scala> 'x(1) + 1 <= 'x(2)
res10: jp.kobe_u.copris.Le = Le(Add(x(1),1),
x(2))

```

後述する `jp.kobe_u.copris.dsl` オブジェクトのメソッドをインポートすることで、この暗黙変換が利用可能になる。

4.1.3 ベクトルと行列の定義

図 1 の例にも見られるように、ベクトル演算や行列演算を利用したい場合がある。そのためベクトルを表す `Vec` クラスと行列を表す `Matrix` クラスを用意し、要素毎の演算や内積 (演算子名は `.*`)、行列の転置、行列積などのメソッドを定義する。

4.1.4 CSP 用クラス

図 6 に CSP 用クラスの定義の一部を示す。CSP クラスのメンバは 2 行目から 5 行目に示す通りであり、それぞれ整数変数の集合、命題変数の集合、整数変数のドメイン関数、制約の集合を表し、初期値は空である。6 行目は整数変数とそのドメイン定義を追加する `int` メソッド、7 行目は制約を追加する `add` メソッド、8 行目は最小化する目的変数を指定する `minimize` メソッドである。

以下に簡単な CSP の記述例を示す。

```

// CSP の生成
scala> val csp = CSP()
// 整数変数  $x, y \in \{1..3\}$  の追加
scala> val x = csp.int(Var("x"), 1, 3)
scala> val y = csp.int(Var("y"), 1, 3)
// 制約  $x \geq 2$  の追加
scala> csp.add(x >= 2)
// 制約  $x + y \leq 4$  の追加
scala> csp.add(x + y <= 4)

```

```

1: case class Solution(
2:   intValues: Map[Var, Int],
3:   boolValues: Map[Bool, Boolean])
4:
5: abstract class AbstractSolver(csp: CSP) {
6:   def find: Boolean
7:   def findNext: Boolean
8:   def findOpt: Boolean
9:   def solution: Solution
10: }
11: object DefaultSolver {
12:   def apply(csp: CSP): AbstractSolver =
13:     new sugar.Solver(csp)
14: }

```

図 7 Solver クラスの定義 (一部)

4.2 CSP ソルバー用クラス

図 7 に CSP ソルバー用クラスの一部を示す。1 行目から 3 行目は CSP の解を表す `Solution` クラスである。`Solution` は、整数変数からその値へのマップと、命題変数からその値へのマップとからなる。`Solution` クラスには `Term` や `Constraint` を引数とする `apply` メソッドが定義されており、与えられた式の値を返す。

5 行目以降は制約ソルバーを表す `AbstractSolver` 抽象クラスの定義である。`find` は与えられた CSP に対して最初の解を探索し、解が存在すれば真を返す。`findNext` は次の解を求めるメソッドである。`findOpt` は与えられた制約最適化問題の最適解を探索する。9 行目は、解を取得するメソッドである。

`DefaultSolver` オブジェクトは、デフォルトの制約ソルバーを生成するためのファクトリーメソッドを提供している。`DefaultSolver(csp)` によりデフォルトの制約ソルバーとして `jp.kobe_u.copris.sugar.Solver` が生成される。これは後述の `Sugar` を利用した制約ソルバーである。

以下に `DefaultSolver` の利用例を示す。

```

// DefaultSolver の生成
scala> val solver = DefaultSolver(csp)
// 最初の解の探索
scala> solver.find
res11: Boolean = true
// 解の表示
scala> solver.solution
res12: jp.kobe_u.copris.Solution =
      Solution(Map(x -> 3, y -> 1), Map())

```

```

1: val csp = CSP()
2: val x = csp.int(Var("x"), 0, 7)
3: val y = csp.int(Var("y"), 0, 7)
4: csp.add(x + y == 7)
5: csp.add(x * 2 + y * 4 == 20)
6: val solver = DefaultSolver(csp)
7: if (solver.find)
8:   println(solver.solution)

```

図 8 Copris による直接的な CSP の記述例

```

scala> solver.solution(x)
res13: Int = 3
scala> solver.solution(x + y)
res14: Int = 4

```

4.3 DSL 用クラス

以上で Scala から Sugar を利用するためのインターフェイスは実現できているが、プログラム例 (図 8) で示すように、制約モデリング言語としての記述力は十分とはいえない。

そこで本節では、Scala の機能を利用し、より自然な制約プログラミング用 DSL の実現を目指す。記述例としては以下の CSP を取り上げ、その他のプログラム例は付録で紹介する。

$$\begin{aligned}
 x &\in \{0, 1, 2, \dots, 7\} \\
 y &\in \{0, 1, 2, \dots, 7\} \\
 x + y &= 7 \\
 2x + 4y &= 20
 \end{aligned}$$

なお、以降は `import jp.kobe_u.copris._` によるインポートを常に仮定し、プログラム例中での記述は省略する。

4.3.1 直接的な記述

ここまでで定義したクラスと API を用いて例題の CSP を直接的に記述した例は図 8 のようになる。常に対象となる CSP オブジェクトや Solver オブジェクトを指定しており、記述が冗長である。

プログラム中、1 行目で CSP オブジェクトを生成し、2 行目と 3 行目で CSP の整数変数として x と y を生成し、CSP オブジェクトの `int` メソッドを用い、 x と y のドメインを指定して CSP に追加している。4 行目と 5 行目では、CSP オブジェクトの `add` メソッドを用い、 $x + y = 7$ と $2x + 4y = 20$ を表す

```

1: trait CoprisTrait {
2:   implicit def symbol2var(s: Symbol) =
3:     Var(s.name)
4:   def csp: CSP
5:   def solver: AbstractSolver
6:   def use(newSolver: AbstractSolver): Unit
7:   def int(x: Var, lo: Int, hi: Int) =
8:     csp.int(x, lo, hi)
9:   def add(cs: Constraint*) =
10:    csp.add(cs: _*)
11:   def find = solver.find
12:   def solution = solver.solution
13: }
14: class Copris(val csp: CSP,
15:              var solver: AbstractSolver)
16:   extends CoprisTrait {
17:   def this(csp: CSP) =
18:     this(csp, DefaultSolver(csp))
19:   def this() = this(CSP())
20:   def use(newSolver: AbstractSolver) =
21:     solver = newSolver
22: }

```

図 9 Copris DSL 用クラスの定義 (一部)

制約を CSP に追加している。作成した CSP に対して、6 行目で制約ソルバーを生成し、7 行目の `find` メソッドで解を探索し、8 行目で以下のような CSP の解を表示している。

```
Solution(Map(x -> 4, y -> 3), Map())
```

4.3.2 クラス継承による DSL の実装

ここでは、クラス継承により DSL を実装する方法を示す。

図 9 が DSL 用クラスである Copris の定義である。1 行目から 13 行目で Copris クラスのインターフェイスを CoprisTrait トレイト (実装を継承できるインターフェイス) として定義している。CoprisTrait では `int`, `add`, `find` などのメソッドを、`csp` や `solver` に実行を移譲する形で実装している。`csp`, `solver` および `use(newSolver)` は CoprisTrait を継承したクラス側で実装しなければならない (Java での抽象メソッド)。14 行目以降が Copris クラスの実装である。14, 15 行目で `csp`, `solver` を、20 行目で `use(newSolver)` を定義している。なお、Scala では引数なしのメソッドを変数で再定義することができ、ここではその特徴を利用している。この Copris クラスを継承すれば、CoprisTrait で定義されているメ

```

1: class Test1 extends Copris {
2:   def main(args: Array[String]) = {
3:     val x = int('x, 0, 7)
4:     val y = int('y, 0, 7)
5:     add(x + y === 7)
6:     add(x * 2 + y * 4 === 20)
7:     if (find)
8:       println(solution)
9:   }
10: }

```

図 10 クラス継承による Copris DSL のプログラム例

```

1: object dsl extends CoprisTrait {
2:   val coprisVar =
3:     new DynamicVariable[Copris](new Copris)
4:   def csp = coprisVar.value.csp
5:   def solver = coprisVar.value.solver
6:   def use(newSolver: AbstractSolver) =
7:     coprisVar.value.use(newSolver)
8:   def using(copris: Copris = new Copris)
9:     (block: => Unit) =
10:     coprisVar.withValue(copris) { block }
11: }

```

図 11 DynamicVariable クラスを用いた Copris DSL の実装

ソッドが直接的に利用可能になる。

図 10 にプログラム例を示す。Test1 クラスは Copris クラスを継承しており、int, add, find 等のメソッドを直接的に利用している。

応用プログラムで Copris クラスを継承することができない場合は、Copris クラスではなく CoprisTrait トレイトを継承し、csp, solver および use(newSolver) を定義すれば良い。

4.3.3 DynamicVariable クラスを用いた DSL の実装

前述のクラス継承を用いた方法では、単一の CSP オブジェクトに対する記述のみが可能であり、複数の CSP を必要とする場合に対応できない。ほとんどの場合、単一 CSP で十分と考えられるが、複数 CSP が必要な場合にも対応可能なように拡張を行う。

図 11 に Scala の DynamicVariable クラスを用いて CSP の動的スコープを実現した実装を示す。4-6 行目で csp, solver, use(newSolver) を定義している。図中の dsl オブジェクトのメソッドをインポー

```

1: import jp.kobe_u.copris.dsl._
2: using(new Copris) {
3:   val x = int('x, 0, 7)
4:   val y = int('y, 0, 7)
5:   add(x + y === 7)
6:   add(x * 2 + y * 4 === 20)
7:   if (find)
8:     println(solution)
9: }

```

図 12 DynamicVariable クラスを用いた Copris DSL のプログラム例

```

1: import jp.kobe_u.copris.dsl._
2: val x = int('x, 0, 7)
3: val y = int('y, 0, 7)
4: add(x + y === 7)
5: add(x * 2 + y * 4 === 20)
6: if (find)
7:   println(solution)

```

図 13 Copris DSL を用いた標準的なプログラム例

トすれば、複数 CSP の利用が可能になる。図 12 がそのプログラム例である。using(new Copris) により、新しく生成された CSP と制約ソルバーに対するスコープが作成され、それらに対する操作を記述している。

なお、DynamicVariable クラスによる束縛はスタックで管理されており、以下のように入れ子の記述も可能である。

```

using(new Copris) {
  ....
  using(new Copris) {
    ....
  }
  ....
}

```

using を省略した場合、プログラム開始時に生成された CSP と制約ソルバー (図 11 の 3 行目の new Copris) に対する操作となる。したがって図 13 に示す例が、Copris DSL を利用する際に最も標準的な記述となる。図 8 と比較して、簡潔な記述が実現できている。

5 Copris の Sugar インターフェイス

本節では Copris の Sugar インターフェイスについて述べる。Copris 中で Sugar に関するクラスは、`jp.kobe_u.copris.sugar` パッケージ中に定義されている。

5.1 SatSolver クラス

Sugar は SAT 型の制約ソルバーであるため、バックエンドとして利用する SAT ソルバーを切り替えて使用できることが望ましい。そこで、図 14 に示すように `SatSolver` 抽象クラスを定義する。`command` は SAT ソルバーコマンドのパス名、`opts` はオプションとして渡す引数である。SAT ファイル名などはオプション引数の後ろに追加されて呼出される。

SAT ソルバーには、結果を標準出力に出力するものと、出力ファイルに書き出すものがあるため、各々を `SatSolver1` クラス (3 行目)、`SatSolver2` クラス (6 行目) として定義する。いずれも Scala 2.9 から導入された `scala.sys.process` パッケージを利用し、SAT ソルバーを別のプロセスとして呼出す方法で実装している。

しかし、SAT ソルバー自体が Java で実装されている場合、別プロセスとして呼出すのではなく、SAT ソルバーの API を直接利用すれば良い。本システムでは、Java で実装された SAT ソルバーの `Sat4j` [4] について、その方法で実装し、プロセス呼出しのオーバーヘッドを削減している。ただし、SAT 問題および解の受け渡しにファイルを用いているため、ファイル入出力のオーバーヘッドは残っている。

5.2 Sugar ソルバー用クラス

`jp.kobe_u.copris.sugar.Solver` クラスは、Sugar 制約ソルバーとのインターフェイスを実装するクラスである。このクラスでは以下の処理を行う。ただし (2) と (4) については、Java で記述された Sugar 制約ソルバーのプログラムを、Copris から呼び出すことにより実現する。

- (1) 与えられた CSP を Sugar の CSP オブジェクトへ変換する。

```

1: abstract class SatSolver(
2:   command: String, opts: Seq[String])
3: class SatSolver1(
4:   command: String, opts: Seq[String])
5:   extends SatSolver(command, opts)
6: class SatSolver2(
7:   command: String, opts: Seq[String])
8:   extends SatSolver(command, opts)

```

図 14 SatSolver クラスの定義 (一部)

```

1: case class Solver(csp: CSP,
2:   satSolver: SatSolver = Sat4j, ...)
3:   extends AbstractSolver(csp) {
4:   def find: Boolean =
5:     encoder.encode && satSolve
6:   def findNext: Boolean = {
7:     val cs1 = for (x <- csp.variables)
8:       yield Eq(x, Num(value(x)))
9:     val cs2 = for (p <- csp.bools)
10:       yield if (value(p)) p else Not(p)
11:     csp.add(Not(And(And(cs1), And(cs2))))
12:     addDelta
13:     commit
14:     satSolve
15:   }
16:   def findOpt: Boolean
17: }

```

図 15 Solver クラスの定義 (一部)

- (2) Sugar の CSP を SAT に符号化する。
- (3) SAT ソルバーを実行する。
- (4) SAT ソルバーの結果を CSP の解に変換する。

以上を行う `Solver` クラスの定義の一部を図 15 に示す。SAT ソルバーの指定が省略された場合は `Sat4j` を用いる。4 行目の `find` は最初の解を探索するメソッドである。`encoder.encode` で、(1) Sugar CSP オブジェクトへの変換と、(2) SAT への符号化を行い、その過程で充足不能でなければ `satSolve` により、(3) SAT ソルバーの実行と、(4) SAT の解の CSP の解への変換を行う。

6 行目の `findNext` は次の解を求めるメソッドである。7 行目から 11 行目で最後に得られた解の否定にあたる条件を CSP に追加し、`addDelta` と `commit` を呼出している。`addDelta` では追加された制約を符号化して SAT のファイルに追記し、`commit` でそれを確定させる。これにより、CSP 全体を何度も符号

```

1: class Sugar(val csp: CSP,
2:   var solver: Solver) extends CoprisTrait {
3:   def this(satSolver: SatSolver) =
4:     this(CSP(), satSolver)
5:   def use(satSolver: SatSolver): Unit =
6:     use(Solver(csp, satSolver))
7: }
8: object dsl extends CoprisTrait {
9:   val sugarVar =
10:    new DynamicVariable[Sugar](new Sugar)
11:   def use(satSolver: SatSolver) =
12:     sugarVar.value.use(satSolver)
13: }

```

図 16 Sugar 用 DSL の実装 (一部)

```

1: import jp.kobe_u.copris.sugar.dsl._
2: use(sugar.MiniSat)
3: val x = int('x, 0, 7)
4: val y = int('y, 0, 7)
5: add(x + y == 7)
6: add(x * 2 + y * 4 == 20)
7: if (find)
8:   println(solution)

```

図 17 Sugar 用 DSL での SAT ソルバーの指定

化することなく CSP の全解を列挙することが可能になり、符号化のオーバーヘッドが大幅に削減される。

16 行目の `findOpt` は制約最適化問題の最適解を探索するメソッドである (定義の記述は省略)。最適解の探索には 2 分法を用いている [16]。2 分法で最適値の範囲を狭めていく場合、最後に追加した制約を削除する処理が必要になる。本論文では説明を省略するが、CSP 全体を再符号化することなく最後に追加した制約の削除にも対応できるように実装しており、効率的な最適値探索を実現している。

5.3 Sugar ソルバー用 DSL

図 16 に Sugar 用 DSL の実装の一部を示す。SAT ソルバーを指定するための `use(satSolver)` メソッドが追加され、Sugar で使用する SAT ソルバーを切り換え可能なように拡張されている。

使用する SAT ソルバーとしては、あらかじめ用意してある `MiniSat`, `MiniSat2`, `MiniSat22`, `PicoSat`, `PrecoSat`, `Glucose`, `GlueMiniSat`, `Sat4j` オブジェクトのうちからひとつ選択するか、新しい `SatSolver`

オブジェクトを生成して使用する。図 17 に `MiniSat` を利用しているプログラム例を示す (2 行目で指定)。

6 おわりに

本論文では、オブジェクト指向言語と関数型言語が融合された Scala 言語上に、制約プログラミング用のドメイン特化言語 (DSL) として開発した `Copris` (Constraint Programming in Scala) について述べた。

`Copris` は、本文および付録中のプログラム例で示したように、Scala のプログラマが簡潔に制約充足問題 (CSP) や制約最適化問題 (COP) を記述できることを目的として設計され、既存の Java 上の制約プログラミング用 API である JSR-331 等よりも高い記述力を提供している。

制約ソルバーとしては、国際制約ソルバー競技会で 2 年連続グローバル制約部門で優勝した SAT 型制約ソルバー `Sugar` を用いており、制約ソルバーとしての性能も高い。`Sugar` がバックエンドとして利用する SAT ソルバーは、`Copris` 内から自由に切換えることができ、SAT ソルバーの性能が今後向上した場合にも簡単に対応ができる。特に `MiniSat`, `PicoSAT`, `GlueMiniSat` など、近年の SAT 競技会で好成績を収めた SAT ソルバーを利用することにより高性能な制約プログラミングが可能となる。また、性能的には劣るが JVM 上で動作する `Sat4j` を SAT ソルバーとして用いた場合、すべてが JVM 上で動作し、可搬性の高いシステムを実現できる点が特徴である。

今後の課題としては、`Sugar` だけでなく JSR-331 に準拠する制約ソルバー等、様々な制約ソルバーへの対応があげられる。

参考文献

- [1] Banbara, M., Matsunaka, H., Tamura, N. and Inoue, K.: Generating Combinatorial Test Cases by Efficient SAT Encodings Suitable for CDCL SAT Solvers, in *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-17)*, LNCS 6397, Springer, 2010, pp. 112–126.
- [2] 番原睦則, 田村直之: SAT によるシステム検証, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 122–129.
- [3] Barták, R.: On-line Guide to Constraint Programming, <http://kti.ms.mff.cuni.cz/~bartak/>

- constraints/, 1998.
- [4] Berre, D. L. and Parrain, A.: The Sat4j Library, release 2.2, *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 7, No. 2-3(2010), pp. 59-64.
 - [5] Biere, A.: PicoSAT Essentials, *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 4, No. 2-4(2008), pp. 75-97.
 - [6] Biere, A., Heule, M., van Maaren, H. and Walsh, T.(eds.): *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications (FAIA), Vol. 185, IOS Press, 2009.
 - [7] Bordeaux, L., Hamadi, Y. and Zhang, L.: Propositional Satisfiability and Constraint Programming: A Comparative Survey, *ACM Computing Surveys*, Vol. 38, No. 4(2006).
 - [8] Codognet, P. and Diaz, D.: Compiling Constraints in clp(FD), *Journal of Logic Programming*, Vol. 27, No. 3(1996), pp. 185-226.
 - [9] Colmerauer, A.: An Introduction to Prolog III, *Communications of the ACM*, Vol. 33, No. 7(1990), pp. 69-90.
 - [10] Crawford, J. M. and Baker, A. B.: Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems, in *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI 1994)*, 1994, pp. 1092-1097.
 - [11] Eén, N. and Sörensson, N.: An Extensible SAT-solver, in *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, LNCS 2919, Springer, 2003, pp. 502-518.
 - [12] Günther, S.: Development of Internal Domain-Specific Languages: Design Principles and Design Patterns, in *Proceedings of the 18th Conference on Pattern Languages of Programs*, 2011.
 - [13] 長谷川隆三, 藤田博, 越村三幸: モデル列挙とモデル計数, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 96-104.
 - [14] Hebrard, E., O'Mahony, E. and O'Sullivan, B.: Constraint Programming and Combinatorial Optimisation in Numberjack, in *Proceedings of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR 2010)*, LNCS 6140, Springer, 2010, pp. 181-185.
 - [15] 平山勝敏, 横尾真: *-SAT: SAT の拡張, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 105-113.
 - [16] Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M. and Tamura, N.: A Competitive and Cooperative Approach to Propositional Satisfiability, *Discrete Applied Mathematics*, Vol. 154, No. 16(2006), pp. 2291-2306.
 - [17] 井上克巳, 田村直之: SAT ソルバーの基礎, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 57-67.
 - [18] 岩沼宏治, 鍋島英知: SMT: 個別理論を取り扱う SAT 技術, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 86-95.
 - [19] 越村三幸, 鍋島英知, 藤田博, 長谷川隆三: SAT 変換による未解決ジョブショップスケジューリング問題への挑戦, スケジューリング・シンポジウム 2009 講演論文集, 2009, pp. 209-213.
 - [20] Koshimura, M., Nabeshima, H., Fujita, H. and Hasegawa, R.: Solving Open Job-Shop Scheduling Problems by SAT Encoding, *IEICE TRANSACTIONS on Information and Systems*, Vol. E93-D, No. 8(2010), pp. 2316-2318.
 - [21] Lecoutre, C., Roussel, O. and van Dongen, M. R. C.: Promoting Robust Black-box Solvers Through Competitions, *Constraints*, Vol. 15, No. 3(2010), pp. 317-326.
 - [22] Mernik, M., Heering, J. and Sloane, A. M.: When and How to Develop Domain-Specific Languages, *ACM Computing Surveys*, Vol. 37, No. 4(2005), pp. 316-344.
 - [23] 鍋島英知: SAT によるプランニングとスケジューリング, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 114-121.
 - [24] 鍋島英知, 岩沼宏治: リテラルブロック距離に基づく良い学習節の評価と獲得による SAT ソルバの性能改善 (特集「AI の基本問題 SAT と応用技術」および一般), 人工知能基本問題研究会, Vol. 81(2011), pp. 1-6.
 - [25] 鍋島英知, 宋剛秀: 高速 SAT ソルバーの原理, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 68-76.
 - [26] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. and Tack, G.: MiniZinc: Towards a Standard CP Modelling Language, in *Proceedings of the 13th International Joint Conference on Principles and Practice of Constraint Programming (CP 2007)*, LNCS 4741, Springer, 2007, pp. 529-543.
 - [27] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E. and Zenger, M.: An Overview of the Scala Programming Language, Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, Switzerland, 2004.
 - [28] Odersky, M., Spoon, L. and Venners, B.: *Programming in Scala*, Second edition, Artima, Inc., 2010.
 - [29] Petke, J. and Jeavons, P.: The Order Encoding: From Tractable CSP to Tractable SAT, in *Proceedings of the 14th International Conference on Theory and Applications of Satisfiability Testing (SAT 2011)*, LNCS 6695, Springer, 2011, pp. 371-372.
 - [30] Rossi, F., van Beek, P. and Walsh, T.: *Handbook of Constraint Programming*, Elsevier, 2006.
 - [31] Schrijvers, T., Stuckey, P. J. and Wadler, P.: Monadic Constraint Programming, *Journal of Functional Programming*, Vol. 19, No. 6(2009), pp. 663-697.
 - [32] Soh, T., Inoue, K., Tamura, N., Banbara, M. and Nabeshima, H.: A SAT-based Method for Solving the Two-dimensional Strip Packing Problem, *Fundamenta Informaticae*, Vol. 102, No. 3-4(2010), pp. 467-487.
 - [33] 多賀明子, 田村直之, 北川哲, 番原睦則: グリッド計算環境上でのショップ・スケジューリング問題の SAT 変換による解法, スケジューリング・シンポジウム 2007

講演論文集, 2007, pp. 109–114.

- [34] Tamura, N., Taga, A., Kitagawa, S. and Banbara, M.: Compiling Finite Linear CSP into SAT, in *Proceedings of the 12th International Joint Conference on Principles and Practice of Constraint Programming (CP 2006)*, LNCS 4204, Springer, 2006, pp. 590–603.
- [35] Tamura, N., Taga, A., Kitagawa, S. and Banbara, M.: Compiling Finite Linear CSP into SAT, *Constraints*, Vol. 14, No. 2(2009), pp. 254–272.
- [36] Tamura, N., Tanjo, T. and Banbara, M.: System Description of a SAT-based CSP Solver Sugar, in *Proceedings of the 3rd International CSP Solver Competition*, 2008, pp. 71–75.
- [37] 田村直之, 丹生智也, 番原睦則: SAT 変換に基づく制約ソルバーとその性能評価, コンピュータソフトウェア, Vol. 27, No. 4(2010), pp. 183–196.
- [38] 田村直之, 丹生智也, 番原睦則: 制約最適化問題と SAT 符号化, 人工知能学会誌, Vol. 25, No. 1(2010), pp. 77–85.
- [39] The choco team: choco: an Open Source Java Constraint Programming Library, in *Proceedings of the 3rd International CSP Solver Competition*, 2008, pp. 7–13.
- [40] 上田和紀: 論理・制約プログラミングと並行計算, コンピュータソフトウェア, Vol. 25, No. 3(2008), pp. 49–54.
- [41] 梅村晃広: SAT ソルバ・SMT ソルバの技術と応用, コンピュータソフトウェア, Vol. 27, No. 3(2010), pp. 24–35.
- [42] van Hentenryck, P.: *The OPL Optimization Programming Language*, MIT Press, 1999.

A 付録: プログラム例

以下に付録として Copris を用いたプログラム例を示す。Copris の利用には Sugar version 1.15.0, Scala version 2.9 以上が必要である。また Sat4j, MiniSat 等の SAT ソルバーも必要となる。Sat4j の場合は Java の jar ファイルが提供されているので、それを Scala のクラスパスに指定する。

A.1 Scala REPL からの利用

Scala にはインタラクティブな実行環境である REPL が用意されている。以下は REPL からの利用例である。まず、次のように Copris, Sugar, Sat4j の jar ファイルをクラスパスに指定して Scala を起動する。

```
$ scala -cp copris-v1-0-1.jar:sugar-v1-15-0.jar:org.sat4j.core.jar
```

次に、REPL 中で整数変数と制約を定義し、解を求め、表示している例を示す。

```
scala> import jp.kobe_u.copris._
```

```
1: val n = 8
2: for (i <- 1 to n) int('q(i), 1, n)
3: add(Alldifferent(
4:   (1 to n).map(i => 'q(i))))
5: add(Alldifferent(
6:   (1 to n).map(i => 'q(i)+i)))
7: add(Alldifferent(
8:   (1 to n).map(i => 'q(i)-i)))
9: if (find) {
10:  do {
11:    println(solution)
12:  } while (findNext)
13: }
```

図 18 Copris での N-Queens 問題のプログラム例

```
scala> import jp.kobe_u.copris.sugar.dsl._
scala> val x = int('x, 0, 7)
scala> val y = int('y, 0, 7)
scala> x + y === 7
scala> x * 2 + y * 4 === 20
scala> find
res2: Boolean = true
scala> solution
res3: jp.kobe_u.copris.Solution =
      Solution(Map(x -> 7, y -> 7), Map())
```

A.2 N-Queens 問題

N-Queens 問題は $N \times N$ のチェス盤に N 個のクイーンを互いに取られないように配置する問題である。図 18 に 8-Queens 問題の全解を表示するプログラム例を示す。

2 行目で CSP の整数変数 $q(1)$ から $q(8)$ を宣言し、それぞれの下限と上限を 1 および 8 と定義している。Sugar には整数変数の配列は存在していないが、Copris 上の整数変数は複数の添字を持てるようにしており、Sugar の整数変数への変換時に q_1 から q_8 という名前に置換される。

3 行目と 4 行目では $q(i)$ の値が互いに異なるという条件を `alldifferent` 制約を用いて表現している。同様に 5 行目から 8 行目では $q(i)+i$ および $q(i)-i$ が互いに異なるという条件を追加している。9 行目では解を探索し、解が存在すれば、10 行目から 12 行目で全解を表示している。

Intel Core i5 vPro 2.0GHz $\times$ 2 のマシンで、バックエンドの SAT ソルバーとして MiniSat 2.2 を用い

```

1: val size = 112
2: val s = List(2,4,6,7,8,9,11,15,16,17,
3:   18,19,24,25,27,29,33,35,37,42,50)
4: val n = s.size
5: for (i <- 0 to n-1) {
6:   int('x(i), 0, size - s(i))
7:   int('y(i), 0, size - s(i))
8: }
9: for (i <- 0 to n-1; j <- i+1 to n-1)
10:   add('x(i) + s(i) <= 'x(j) ||
11:     'x(j) + s(j) <= 'x(i) ||
12:     'y(i) + s(i) <= 'y(j) ||
13:     'y(j) + s(j) <= 'y(i))

```

図 19 Copris での完全正方形詰込み問題のプログラム例

た時、全解を求めるための CPU 時間は約 2.5 秒だった (表示時間を除き、SAT 符号化の時間を含む)。また同一のプログラムで 100-Queens の最初の解を求めるまでの CPU 時間は約 13 秒、200-Queens については約 30 秒だった。

A.3 完全正方形詰込み問題

完全正方形詰込みは、与えられた大きさの異なる正方形をより大きな正方形中に互いに重ならないよう隙間なく詰めて配置する問題である。図 19 に 21 個の正方形を 112×112 の正方形に詰込む問題に対するプログラム例を示す。Intel Core i5 vPro 2.0GHz $\times$ 2 のマシンで MiniSat 2.2 を用いた時、符号化を含む CPU 時間は約 80 秒だった。

A.4 ナンバーリンク

ナンバーリンクはニコリ^{†9} によるパズルである。図 20 に問題例を示す。ルールは以下の通りである。

- (1) 同じ数字どうしを線でつなげる。
- (2) 線はタテヨコに引き、マスの中央を通る。
- (3) 線は 1 マスに 1 本だけ通過できる。線をワクの外に出したり、交差や枝分かれさせてはいけない。また、線は数字が入っているマスを通してはいけない。

図 21 にナンバーリンクに対するプログラム例を示す。変数 `puzzle` に問題がリストのリストとして設定

					3	4
		1			5	
2				5		
		2			3	
1	4					

図 20 ナンバーリンクの問題 (ニコリ)

```

1: def degree(i: Int, j: Int) = {
2:   val vs = for (i0 <- i-1 to i;
3:     if 0 <= i0 && i0 < m - 1)
4:     yield 'v(i0, j)
5:   val hs = for (j0 <- j-1 to j;
6:     if 0 <= j0 && j0 < n - 1)
7:     yield 'h(i, j0)
8:   Add(vs ++ hs: _*)
9: }
10: for (i <- 0 until m; j <- 0 until n) {
11:   if (i < m - 1) int('v(i, j), 0, 1)
12:   if (j < n - 1) int('h(i, j), 0, 1)
13: }
14: for (i <- 0 until m; j <- 0 until n) {
15:   if (puzzle(i)(j) > 0)
16:     int('d(i, j), 1)
17:   else
18:     int('d(i, j), Set(0, 2))
19:   add('d(i, j) === degree(i, j))
20: }
21: for (i <- 0 until m; j <- 0 until n) {
22:   int('x(i, j), 1, k)
23:   if (puzzle(i)(j) > 0)
24:     add('x(i, j) === puzzle(i)(j))
25:   if (i < m - 1)
26:     add(('v(i, j) !== 0) ==>
27:       ('x(i, j) === 'x(i+1, j)))
28:   if (j < n - 1)
29:     add(('h(i, j) !== 0) ==>
30:       ('x(i, j) === 'x(i, j+1)))
31: }

```

図 21 Copris でのナンバーリンクのプログラム例

され、 m が行数、 n が列数、 k が問題中の数字の最大値を表しているとする。

CSP の変数 $v(i, j)$ はマス (i, j) から下に、 $h(i, j)$ はマス (i, j) から右に線があることを表す。各マスを頂点、隣接したマスへの線を辺としたグラフを考えると、数字マスの次数は 1、白マスの次数

^{†9} <http://www.nikoli.co.jp>

は 0 または 2 となる. 14 行目から 20 行目でこの制約を加えている. さらに, 同じ数字が線で結ばれる条件を表すために, 各マスにどの数字と結ばれているかを表す変数 $x(i, j)$ を割当てる. 辺で結ばれている場合, この値が同じでなければならない. この制約を 21 行目から 31 行目で加えている.

Intel Core i5 vPro 2.0GHz $\times$ 2 のマシンで MiniSat 2.2 を用いた時, 図 20 を解くのにかった CPU 時間は符号化を含め約 1.5 秒だった. 辺を有向辺にし, 盤面を 4 分割した部分領域での入次数・出次数に関する制約を追加すれば, さらに速度が改善され, 25×45 のサイズの問題でも約 50 秒で解くことができる.



田村直之

1980 年神戸大学理学部物理学科卒業. 1985 年同大学大学院自然科学研究科博士課程システム科学専攻修了. 学術博士. 1985 年日本 IBM 東京基礎研究所入社. 1988 年神戸大学工学部勤務. 2003 年より神戸大学情報基盤センター教授. 論理プログラミング, 制約プログラミング, SAT 技術等に興味をもつ.

日本ソフトウェア科学会, 情報処理学会会員.



丹生智也

2007 年神戸大学工学部情報知能工学科卒業. 2009 年同大学大学院工学研究科博士課程前期課程情報知能学専攻修了. 2012 年同研究科博士課程後期課程情報知能学専攻修了. 2012 年より新領域融合研究センター融合プロジェクト特任研究員. 制約プログラミング, 論理プログラミング, SAT 技術, 解集合プログラミング, 擬似ブール制約等に興味をもつ. 日本ソフトウェア科学会, 人工知能学会会員.



番原睦則

1994 年神戸大学理学部数学科卒業. 1996 年同大学大学院自然科学研究科博士課程前期数学専攻修了. 1996 年国立奈良工業高等専門学校助手. 1998 年同校講師. 2003 年より神戸大学情報基盤センター講師. 2007 年同校准教授. 博士 (工学). 論理プログラミング, 制約プログラミング, SAT 技術等に興味をもつ. 日本ソフトウェア科学会, 情報処理学会会員.