GSLetterNeo vol.148

2020 年 11 月

SMT ソルバーZ3 を使った問題解決入門

熊澤 努 kumazawa @ sra.co.jp

はじめに

今回は、Microsoft 社で開発されている Z3 という SMT ソルバーを紹介します。SMT ソルバーは、不等式や方程式、配列、真理値などを使って定式化された問題を自動的に解くツールです。例えば、「$x + y \leq 1$を満たすx, yを求めよ」という問題に対して、高速でxとyを求めます。また、問題の記述のしやすさにも配慮がなされているため、工学をはじめとするいろいろな分野において容易に活用することができます[1]。なお、SMT は背景理論付き充足可能性問題（Satisfiability Modulo Theories）の略語です（詳細は省略します）。

Z3 は、オープンソースソフトウェアとして無償で使用できます。また、いくつかのプログラミング言語（2020 年 10 月現在では C、C++、Java、OCaml、Python、Julia）で書いたプログラムで使用するための仕組みが提供されていて、プログラマにとっても大変便利です。問題が定義されていれば、プログラマは、(1) 問題をプログラムで記述する、(2) Z3 のソルバーに問題を追加する、(3) ソルバーを使って問題を解く、(4) ソルバーが返す結果を利用する、というステップに従ったプログラムを書くことで、Z3 を利用して問題解決を体験することができます。今回の記事では、Pythonプログラムを通じてZ3を使ってみます。

---

[1] 最近の SMT ソルバーの応用例には、国民年金法の検証事例があります。詳しくは、片山卓也：「国民年金法の述語論理による記述と検証 SMT ソルバーZ3Py を用いたケーススタディ」，コンピュータソフトウェア，Vol. 36，No.3，pp. 33-46，2019 (https://doi.org/10.11309/jssst.36.3_33)を参照してください.

Z3 を準備する

Z3 を入手する方法を説明します。Z3 を使う方法には、ソースコードをユーザが自分でコンパイルする方法と、コンパイル済みバイナリファイルをダウンロードする方法があります。今回は後者を試してみます。なお、最新の情報については、公式サイトである GitHub の公開レポジトリを確認してください。

https://github.com/z3prover/z3

2020 年 10 月現在、コンパイル済みのバイナリは、下記のサイトで公開されています。

https://github.com/Z3Prover/z3/releases

今回は筆者がアクセスした時点での最新安定バージョン 4.8.9 を使います。バイナリを一式圧縮したファイルがあるので、使っている環境に合ったものをダウンロードしてください。以降の手順は、Windows の場合について説明します。
Windows の場合、圧縮形式は zip です。ダウンロードした zip ファイルを適当なインストール用フォルダで解凍します。次に、解凍してできたフォルダには bin フォルダがあるので、この bin フォルダへのパスを環境変数 PATH に追加します。ただし、使ってみるだけなら環境変数 PATH を編集する作業は必須ではありません。とりあえず使ってみたい読者は、この作業は後回しにしても構いません。今回は環境変数の設定はしない場合を解説します。

Z3 の API を Python から呼び出すために、z3-solver パッケージを Python にインストールします。pip を使える場合、Python のコマンドライン環境で以下のコマンドを実行してください。

```
pip install z3-solver
```

インストール後には、Z3 を Python で使うための環境変数 PYTHONPATH を設定します。解凍した bin フォルダに python フォルダがあるので、そのパスを設定しましょう（正確には z3.py という Python ファイルのパスを設定します）。Z3 をとりあえず使う場合には PYTHONPATH の設定も不要なので、今回の記事では設定していません。

Z3 で問題を解いてみる

Z3 を使って簡単な問題を解くプログラムを作って動かしてみましょう。

次の Python プログラムを作り、ex1.py という名称で保存します。環境変数を設定していない場合、保存するフォルダを Z3 の bin フォルダとしてください（他の例のソースコードもこのフォルダに保存します）。

```
from z3 import *

x = Int('x')
y = Int('y')
solver = Solver()
solver.add(3 * x + 2 * y == 13, x > y, x > 1, y > 1)

print(solver.check())
print(solver.model())
```

これは、次の 4 つの式をすべて満たす整数x, yを求める問題を解くプログラムです。

$$3x + 2y = 13,$$
$$y < x,$$
$$1 < x,$$
$$1 < y.$$

xとyが実数であれば手計算で簡単に解けますが、この問題では、変数が整数の値しかとれないため、難しくなっています。Z3 を使って解くには、ex1.py を Python で実行します。

```
python ex1.py
```

結果は以下のように出力されます。

```
sat
[y = 2, x = 3]
```

sat は、「問題に解が存在する」ことを意味します（「充足可能」といいます）。解が次の行に表示されている$x = 3, y = 2$です。

次に、上の問題で不等式$y < x$の左辺と右辺を入れ替えて、$x < y$としてみましょう。つまり、以下の式をすべて満たす整数x, yを求めます。

$$3x + 2y = 13,$$
$$x < y,$$
$$1 < x,$$
$$1 < y.$$

問題を解く Python プログラムは以下のようになります。ex2.py という名前で保存して実行してみましょう。

```
from z3 import *

x = Int('x')
y = Int('y')
solver = Solver()
solver.add(3 * x + 2 * y == 13, y > x, x > 1, y > 1)

print(solver.check())
```

結果は次のようになります。unsat は「問題に解が存在しない」という意味です（「充足不能」といいます）。このように、Z3 は、自動で解の有無を判定します。

```
unsat
```

Z3 を使ったプログラムを読む

プログラムは次のような流れで書かれています。まず、変数を定義し、解きたい問題の数式を、定義した変数を使って書き下します。次に、解を求めるソルバーが数式を読み込んで求解し、結果を表示します。ex1.py を読んで、問題を解くプログラムの書き方を詳しく調べてみましょう。ex1.py を下に再掲します。

```
from z3 import *

x = Int('x')
y = Int('y')
solver = Solver()
solver.add(3 * x + 2 * y == 13, x > y, x > 1, y > 1)

print(solver.check())
print(solver.model())
```

1 行目は Z3 の API を利用するため、z3 のラッパモジュールをインポートしています。
3 行目と 4 行目は未知の変数x, yの定義です。文字列'x'と'y'は変数の名前で、結果を表示するときに使われます。
5 行目で問題を解くソルバーを生成します。続く 6 行目では、解きたい問題の条件を制約式として記述して、ソルバーに追加しています。6 行目の式は Python プログラムの計算式ではなく、問題を定式化した等式と不等式そのものである点に注意してください。
8 行目の check は問題に解が存在するか（充足可能性）を判定するメソッドです。解が存在すれば sat を、存在しなければ unsat を出力します。
最後の 9 行目は check メソッドの結果が sat ならば解を出力します。unsat の場合には例外を発行するので注意してください。
なお、z3 モジュールの組込み関数である solve 関数を使うと、下のプログラムのように、ソルバーの生成と画面表示のための print 関数の呼び出しを省略できます。

```
from z3 import *

x = Int('x')
y = Int('y')
solve(3 * x + 2 * y == 13, x > y, x > 1, y > 1)
```

実行結果は次の通りです。

```
[y = 2, x = 3]
```

Z3 でパズルを解く

今度は次のお絵描きロジック[2]というパズルを解いてみましょう。

	1 1 3	1 1 1	1 1 1	1 2 1	 3	3 2 3	 3 4	 1 1	1 2 1 2	 6 2
4 1 2										
3 1										
1 3 2										
1 1 1 2										
1 1 1 1										
1 1 1 1										
1 1 1 4										
1 3										
1 4 2										
2 2										

(この問題は Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli: Integer Programming, Springer, 2014. Exercise 2.30 (p.82)より引用しました)

この問題は、行と列の脇にある数字列（塗りつぶし数配列）にしたがってマス目を塗りつぶすパズルです。数字は、その行で連続して塗りつぶすマス目の数（塗りつぶし数）です。例えば、1 行目の塗りつぶし数配列「4 1 2」は、それぞれ 4 マス、1 マス、2 マス続けて塗りつぶしたマス目が 1 行目に存在し、それらが数字の並び順の通りに並ぶことを意味します。続けて塗りつぶしたマス目の両隣のマス目は塗りつぶしてはいけません。したがって、1 行

[2] お絵描きロジックは、下記のサイトで詳しく説明されています。
http://www.puzzler.ne.jp/solve/logic.html

目で４マス続けて塗りつぶしたら、その左右は塗りつぶさないようにします。以下は塗りつぶし方の例です。

ダメな例も載せます。下は、1 マスと 2 マスの塗りつぶしの間に塗りつぶさないマス目がありません。

また、下は、数字の順に塗りつぶしたマス目が並んでいないため、許されません。

この問題を解くプログラムを次のページに示します。少し長いので、コメントと制約を太文字にしました。

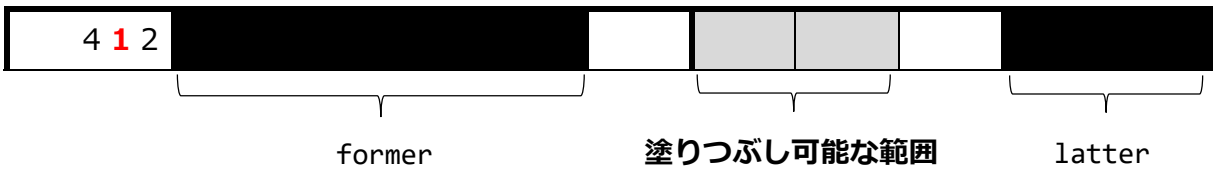
変数 squares はマスを表す変数の配列です。z3 で用意している整数変数の配列 IntVector を使って定義しています。引数'x'は配列名、10**2(=100)は配列のサイズです。下のように、配列の要素を 1 つのマス目に対応させます。

	113	111	111	121	33	323	34	11	1212	62
４１２	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9
３１	x_{10}	x_{11}	x_{12}	x_{13}	x_{14}	x_{15}	x_{16}	x_{17}	x_{18}	x_{19}
１３２	x_{20}	x_{21}	x_{22}	x_{23}	x_{24}	x_{25}	x_{26}	x_{27}	x_{28}	x_{29}
１１１２	x_{30}	x_{31}	x_{32}	x_{33}	x_{34}	x_{35}	x_{36}	x_{37}	x_{38}	x_{39}
１１１１	x_{40}	x_{41}	x_{42}	x_{43}	x_{44}	x_{45}	x_{46}	x_{47}	x_{48}	x_{49}
１１１１	x_{50}	x_{51}	x_{52}	x_{53}	x_{54}	x_{55}	x_{56}	x_{57}	x_{58}	x_{59}
１１１４	x_{60}	x_{61}	x_{62}	x_{63}	x_{64}	x_{65}	x_{66}	x_{67}	x_{68}	x_{69}
１３	x_{70}	x_{71}	x_{72}	x_{73}	x_{74}	x_{75}	x_{76}	x_{77}	x_{78}	x_{79}
１４２	x_{80}	x_{81}	x_{82}	x_{83}	x_{84}	x_{85}	x_{86}	x_{87}	x_{88}	x_{89}
２２	x_{90}	x_{91}	x_{92}	x_{93}	x_{94}	x_{95}	x_{96}	x_{97}	x_{98}	x_{99}

```
from z3 import *

# 行と列の数
N = 10
# 各マス目に対応した整数変数
squares = IntVector('x', N ** 2)

solver = Solver()

# 1. マス目を塗りつぶすかどうか（塗らない:0、または、塗りつぶす:1）
solver.add([And(0 <= x, x <= 1) for x in squares])

# 2. 一行（列）の塗りつぶしルールを制約として追加する関数
def add_line_constraints(solver, line_squares, line_numbers):
    for j, num in enumerate(line_numbers):
        former = line_numbers[:j]
        latter = line_numbers[j+1:]
        constraints = []
        # 2.1. 可能な塗りつぶし方を計算する
        for k in range(sum(former)+len(former),N-(num-1)-sum(latter)-len(latter)):
            # 塗りつぶし一つ分の制約
            # 2.2. マス目を続けて塗りつぶす
            c = [Sum([line_squares[k + n] for n in range(num)]) == num]
            # 2.3. 続けて塗りつぶしたマス目の隣のマス目は塗りつぶさない
            if 0 < k:
                c.append(line_squares[k - 1] == 0)
            if k + num < N:
                c.append(line_squares[k + num] == 0)
            constraints.append(And(c))
        # 可能な塗りつぶし方の内どれかが成り立つ
        solver.add(Or(constraints))
    # 2.3. 塗りつぶすマス目の総数
    solver.add(Sum(line_squares) == sum(line_numbers))

# 各行の塗りつぶしの制約
rows = [[4, 1, 2], [3, 1], [1, 3, 2], [1, 1, 1,2], [1, 1, 1, 1],
    [1, 1, 1, 1], [1, 1, 1, 4], [1, 3], [1, 4, 2], [2, 2]]

for i, row in enumerate(rows):
    row_squares = squares[i * N: (i + 1) * N]
    add_line_constraints(solver, row_squares, row)

# 各列の塗りつぶしの制約
columns = [[1, 1, 3], [1, 1, 1], [1, 1, 1], [1, 2, 1], [3, 3],
    [3, 2, 3], [3, 4], [1, 1], [1, 2, 1, 2], [6, 2]]
for i, col in enumerate(columns):
    col_squares = [ squares[j * N + i] for j in range(N)]
    add_line_constraints(solver, col_squares, col)

# ソルバーで問題を解く
ret = solver.check()
print(ret)
if ret == sat:
    model = solver.model()
    for i in range(N):
        print([model[squares[N * i + j]] for j in range(N)])
```

次に、問題が定める制約事項を一つ一つ不等式で定式化していきます。制約は大きく 2 つに分けられます（ソースコード中の番号付きコメントに対応しています）。

1. 変数 squares の各要素は 0 または 1 のどちらかしか取らないこととします。各要素に対応したマス目を塗りつぶさない場合は 0、塗りつぶす場合は 1 です。このことを不等式で追加します。z3 モジュールの組込み関数 And は、2 つの引数のどちらも成り立つことを表す論理式（0 以上かつ 1 以下である）を制約として生成します。なお、制約のリストを add メソッドに渡すと、リストのすべての要素を制約として追加します。
2. 各行（列）のマス目の塗りつぶしルールを定式化した関数 add_line_constraints です。ルールは、**指定された数のマス目を続けて塗りつぶすこと(2.1.)**、**連続で塗りつぶしたマス目の両隣のマス目は塗りつぶさないこと(2.2.)**、**塗りつぶすマス目の総数は塗りつぶし数配列の要素の合計値になること(2.3.)**の 3 つです。変数 former と latter は、塗りつぶし数配列の内、現在着目している塗りつぶし数の前後の情報を格納した配列です。これらは、塗りつぶすことが可能なマス目の範囲を絞り込むために使います。例えば、今 1 行目で 1 マス塗りつぶすことを考えます。このとき、former=[4]、latter=[2]となります。この情報から、1 マス塗りつぶしのできる範囲は下のように 2 マスのみであることが分かります。



2.1. 連続したマス目を塗りつぶすことを制約として加えます。そのために、塗りつぶし可能な範囲に渡って、塗りつぶすマス目を列挙します。このことを、塗りつぶすときは変数の値が 1 になることを利用して、変数の和で表現しています。
2.2. 続けて塗りつぶしたマス目の両隣のマス目を塗りつぶさないことを制約として加えています。端のマス目を塗りつぶした場合には、隣接するマス目は片側にしか存在しないことに注意します。
2.3. 塗りつぶすマス目の数は、塗りつぶし数配列の要素の合計値と一致することを制約に追加しています。例えば、1 行目で塗りつぶすマス目の数は、$4+1+2=7$マスです。

ここで挙げたプログラムでは、塗りつぶしが可能な組み合せを全て列挙しています。z3 の組込み関数である Or 制約を使って、列挙した組合せの内どれかが成り立つことを制約とします。最後に、ソルバーで問題を解き、適切な塗りつぶし方が存在すれば、それをリスト表示します。

このプログラムを実行した結果を以下に示します。

```
sat
[1, 1, 1, 1, 0, 1, 0, 1, 1, 0]
[0, 0, 0, 0, 1, 1, 1, 0, 0, 1]
[0, 0, 1, 0, 1, 1, 1, 0, 1, 1]
[0, 1, 0, 0, 1, 0, 1, 0, 1, 1]
[1, 0, 0, 1, 0, 1, 0, 0, 0, 1]
[0, 1, 0, 1, 0, 1, 0, 0, 0, 1]
[1, 0, 1, 0, 1, 0, 1, 1, 1, 1]
[1, 0, 0, 0, 1, 1, 1, 0, 0, 0]
[1, 0, 0, 1, 1, 1, 1, 0, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 0, 1, 1]
```

1が塗りつぶすマス目を、0 が塗りつぶさないマス目を表しています。この結果をもとのパズルで描くと下の通りになります。指定された塗りつぶし数の通りに塗られていることが分かります。

	1 1 3	1 1 1	1 1 1	1 2 1	3 3	3 2 3	3 4	1 1	1 2 1 2	6 2
4 1 2	■	■	■	■	■		■		■	■	
3 1					■	■	■			■
1 3 2			■		■	■	■		■	■
1 1 1 2		■			■		■		■	■
1 1 1 1	■			■		■				■
1 1 1 1		■		■		■				■
1 1 1 4	■		■		■		■	■	■	■
1 3	■				■	■	■			
1 4 2	■			■	■	■	■		■	■
2 2						■	■		■	■

おわりに

今回は、Microsoft 社が公開している問題解決ツール Z3 を紹介しました。Python から使用する方法や、使用事例をもっと詳しく知りたい読者は、下記の Z3Py のチュートリアルページを読んでみてください。

https://ericpony.github.io/z3py-tutorial/guide-examples.htm

SMT ソルバーについては、下記の解説記事で詳しく説明されています。

岩沼 宏治, 鍋島 英知：「SMT:個別理論を取り扱う SAT 技術(<特集>最近の SAT 技術の発展)」，人工知能学会誌，Vol. 25，No.1，pp. 86-95，2010
(https://doi.org/10.11517/jjsai.25.1_86).

GSLetterNeo Vol.148
2020 年 11 月 20 日発行
発行者　株式会社 SRA 先端技術研究所

編集者　土屋正人
バックナンバー http://www.sra.co.jp/gsletter
お問い合わせ　gsneo@sra.co.jp

株式会社SRA
〒171-8513 東京都豊島区南池袋 2-32-8