Z3Python

March 11, 2021

1 Breve introdução à utilização do Z3 em Python

Um tutorial do Z3Py, a biblioteca Python de interface para o popular solver Z3 da Microsoft, pode ser encontrado em https://ericpony.github.io/z3py-tutorial/guide-examples.htm.

Começamos por importar o módulo do Z3.

[1]:
```python
from z3 import *
```

As funções `Int()`, `Real()`, `Bool()` criam uma variável no Z3 do tipo correspondente. A função `solve` resolve um sistema de restrições. Por exemplo, para encontrar uma solução para o sistemas equações $x > 2$, $y < 10$ e $x + 2 \times y = 7$ podemos utilizar o seguinte programa.

[2]:
```python
x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

```
[y = 0, x = 7]
```

Mais um exemplo, agora com a teoria de reais, para encontrar a solução para o sistema de equações $x^2 + y^2 > 3$ e $x^3 + y < 5$.

[3]:
```python
x = Real('x')
y = Real('y')
solve(x**2 + y**2 > 3, x**3 + y < 5)
```

```
[y = 2, x = 1/8]
```

O Z3 também pode ser usado como SAT solver. Para tal basta usar variáveis do tipo Bool e fórmulas proposicionais. Por exemplo, o programa seguinte verifica se a conjunção das fórmulas $p \to q$, $r \leftrightarrow \neg q$, e $\neg p \vee r$ é satisfazível.

[4]:
```python
p = Bool('p')
q = Bool('q')
r = Bool('r')
solve(Implies(p, q), r == Not(q), Or(Not(p), r))
```

```
[q = True, p = False, r = False]
```

Também podemos usar o Z3 para simplificar expressões.

[5]:
```
p = Bool('p')
q = Bool('q')
print (And(p, q, True))
print (simplify(And(p, q, True)))
print (simplify(And(p, False)))
```

```
And(p, q, True)
And(p, q)
False
```

[6]:
```
p = Bool('p')
x = Real('x')
solve(Or(x < 5, x > 10), Or(p, x**2 == 2), Not(p))
```

```
[x = -1.4142135623?, p = False]
```

É possível controlar a precisão com que são apresentados os números reais alterando a opção `precision`.

O Z3 também permite resolver conjuntos de restrições envolvendo variáveis de vários tipos.

[7]:
```
set_option(precision=30)
solve(Or(x < 5, x > 10), Or(p, x**2 == 2), Not(p))
```

```
[x = -1.414213562373095048801688724209?, p = False]
```

O comando `Solver()` cria um solucionador de propósito geral. Inicialmente não tem restrições. Está vazio.

[8]:
```
x = Int('x')
y = Int('y')

s = Solver()
print(s)
```

```
[]
```

As restrições podem ser adicionadas usando o método `add`. O método `check` resolve as restrições declaradas. O resultado é `sat` se uma solução for encontrada.

[9]:
```
s.add(x > 10, y == x + 2)
print(s)
print("Solving constraints in the solver s ...")
print(s.check())
```

```
[x > 10, y == x + 2]
Solving constraints in the solver s ...
sat
```

O resultado é `unsat` se não houver solução.

Em algumas aplicações, queremos explorar vários problemas semelhantes que compartilham várias restrições. Podemos usar os métodos `push` e `pop` para fazer isso. Cada *solver* mantém uma pilha de asserções (restrições). O método `push` cria um novo escopo, salvando o tamanho atual da pilha. O método `pop` remove qualquer asserção acrescentada entre ele e o `push` correspondente. O método `check` opera sobre o conjunto de asserções que estão no topo da pilha.

[10]:
```
print("Create a new scope...")
s.push()
s.add(y < 11)
print(s)
print("Solving updated set of constraints...")
print(s.check())
```

```
Create a new scope...
[x > 10, y == x + 2, y < 11]
Solving updated set of constraints...
unsat
```

[11]:
```
print("Restoring state...")
s.pop()
print(s)
print("Solving restored set of constraints...")
print(s.check())
```

```
Restoring state...
[x > 10, y == x + 2]
Solving restored set of constraints...
sat
```

Finalmente, um *solver* pode não ser capaz de se pronunciar quanto à satisfazibilidade de um conjunto de restrições. Nesse caso devolve `unknown`.

[12]:
```
x = Real('x')
s = Solver()
s.add(2**x == 3)
print(s.check())
```

```
unknown
```

1.0.1 Exemplo

O Cryptarithms é um jogo que consiste numa equação matemática entre números desconhecidos, cujos dígitos são representados por letras. Cada letra deve representar um dígito diferente e o dígito inicial de um número com vários dígitos não deve ser zero.

Queremos saber os dígitos a que correspondem as letras envolvidas na seguinte equação:

```
TWO + TWO = FOUR
```

Podemos modelar o problema numa teoria de inteiros. Cada letra dá origem a uma variável inteira (T,W,O,F,U, e R) e para representar a equação acima representamos cada parcela por uma expressão aritmética onde cada letra é multiplicada pelo seu "peso específico" (em base 10).

Resolver este problema equivale a resolver o seguinte sistema de equações:

$$\begin{cases} 0 \leq T \leq 9 \\ \ldots \\ 0 \leq R \leq 9 \\ T \neq W \neq O \neq F \neq U \neq R \\ T \neq 0 \\ F \neq 0 \\ (100 \times T + 10 \times W + O) + (100 \times T + 10 \times W + O) = 1000 \times F + 100 \times O + 10 \times U + R \end{cases}$$

Em Z3 este sistema pode ser resolvido da seguinte forma.

[13]:
```
T, W, O, F, U, R = Ints('T W O F U R')
s = Solver()

s.add(And(0<=T,T<=9))
s.add(And(0<=W,W<=9))
s.add(And(0<=O,O<=9))
s.add(And(0<=F,F<=9))
s.add(And(0<=U,U<=9))
s.add(And(0<=R,R<=9))

s.add(Distinct(T, W, O, F, U, R))

s.add(Not(T==0))
s.add(F!=0)

s.add((T*100+W*10+O)+(T*100+W*10+O)==F*1000+O*100+U*10+R)

r = s.check()
if r==sat :
    m = s.model()
    print(m)
else:
    print("Não tem solução.")
```

```
[O = 4, W = 3, T = 7, F = 1, U = 6, R = 8]
```

Podemos consultar o conjunto de restrições que temos no solver `s`, usando o método `assertions`.

[14]:
```
for c in s.assertions():
    print(c)
```

```
And(T >= 0, T <= 9)
And(W >= 0, W <= 9)
And(O >= 0, O <= 9)
```

```
And(F >= 0, F <= 9)
And(U >= 0, U <= 9)
And(R >= 0, R <= 9)
Distinct(T, W, O, F, U, R)
Not(T == 0)
F != 0
T*100 + W*10 + O + T*100 + W*10 + O ==
F*1000 + O*100 + U*10 + R
```

Podemos consultar o modelo `m` gerado. No programa seguinte, `decls` é um método que devolve as variáveis atribuídas no modelo, `name` devolve o nome de uma variável atribuída no modelo, e `m[d]` o valor atribuído a `d` no modelo `m`. Atenção que este valor não é um tipo primitivo do Python. Por exemplo, para o converter para um inteiro do Python é necessário usar o método `as_long`. Para mais informações sobre estes métodos de conversão ver o seguinte post no Stack Overflow: https://stackoverflow.com/questions/12598408/z3-python-getting-python-values-from-model/12600208

[15]:
```
for d in m.decls():
    print("%s = %d" % (d.name(), m[d].as_long()))
```

```
O = 4
W = 3
T = 7
F = 1
U = 6
R = 8
```

Como podemos saber se existem outras soluções para este quebra-cabeças? Podemos acrescentar restrições de forma a excluir a solução apresentada pelo *solver*, e testar novamente.

[16]:
```
vs = [T, W, O, F, U, R]
while s.check() == sat:
    m = s.model()
    print(m)
    s.add(Or([x != m[x] for x in vs]))  # para excluir as mesmas atribuições␣
 ↪usadas no modelo anterior
```

```
[O = 8, W = 2, T = 9, F = 1, U = 5, R = 6]
[R = 2, O = 6, W = 3, T = 8, F = 1, U = 7]
[R = 0, O = 5, W = 6, T = 7, F = 1, U = 3]
[R = 4, O = 7, W = 6, T = 8, F = 1, U = 3]
[R = 2, O = 6, W = 4, T = 8, F = 1, U = 9]
[R = 8, O = 4, W = 3, T = 7, F = 1, U = 6]
[R = 6, O = 8, W = 3, T = 9, F = 1, U = 7]
```

1.0.2 Exercício 1

Defina uma função `prove` que verifique se uma fórmula proposicional é válida e use essa função para provar lei de Morgan $A \wedge B = \neg(\neg A \vee \neg B)$.

[17]:
```python
def prove(f):
    # completar

# completar

if prove(demorgan):
    print("De Morgan is valid!")
```

```
  File "<ipython-input-17-97fffdf82fce>", line 6
    if prove(demorgan):
     ^
IndentationError: expected an indented block
```

1.1 Modelação em Lógica Proposicional

Considere o seguinte problema:

```
- Maria cannot meet on Wednesday.
- Peter can only meet either on Monday, Wednesday or Thursday.
- Anne cannot meet on Friday.
- Mike cannot meet neither on Tuesday nor on Thursday

When can the meeting take place?
```

Vamos usar o Z3 para encontrar a solução.

1. Vamos modelar o problema em Lógica Proposicional, criando uma variável proposicional para cada dia da semana (*Mon*,*Tue*,*Wed*,*Thu*, e *Fri*), com a seguinte semântica: se a variável for `True` é porque a reunião se pode fazer nesse dia, caso contrário será `False`.

2. De seguida, teremos que modelar cada uma das restrições, acrescentando as fórmulas lógicas correspondentes.

$$\neg Wed$$
$$Mon \lor Wed \lor Thu$$
$$\neg Fri$$
$$\neg Tue \land \neg Thu$$

3. Finalmente testamos se o conjunto de restrições é satisfazível e extraimos a solução calculada.

[ ]:
```python
Mon, Tue, Wed, Thu, Fri = Bools('Monday Tuesday Wednesday Thursday Friday')
s = Solver()
s.add(Not(Wed))
s.add(Or(Mon,Wed,Thu))
```

```
s.add(Not(Fri),And(Not(Tue),Not(Thu))) # Também é possível passar várias restrições ao solver de uma vez só

if s.check() == sat:
    m = s.model()
    print(m)
else:
    print("The meeting cannot take place!")
```

1.1.1 Exercício 2

Altere o código acima por forma a imprimir apenas o dia em que deverá ocorrer a reunião (em vez de imprimir todo o modelo).

[ ]:
```
# completar
```

1.1.2 Exercício 3

Considere o seguinte enigma:

```
- If the unicorn is mythical, then it is immortal.
- If the unicorn is not mythical, then it is a mortal mammal.
- If the unicorn is either immortal or a mammal, then it is horned.
- The unicorn is magical if it is horned.

Given these constraints:
- Is the unicorn magical?
- Is it horned?
- Is it mythical?
```

Modele o problema em Lógica Proposicional e use o Z3 para o resolver.

Sugestão: Resolva o problema com o auxílio de 5 variáveis proposicionais, correspondentes às 5 propriedades dos unicórnios. Relembre que a afirmação $A_1, \ldots, A_n \models B$ é válida se e só se o conjunto de restrições $\{A_1, \ldots, A_n, \neg B\}$ é inconsistente. Tire proveito dos métodos `push` e `pop` para responder às várias questões usando de forma incremental o mesmo solver.

[ ]:
```
# completar
```

1.1.3 Exercício 4

Considere o seguinte problema.

```
Temos 3 cadeiras em linha (esquerda, meio, e direita) e precisamos de sentar 3 convidados: a A

- A Ana não quer ficar sentada à beira do Pedro.
- A Ana não quer ficar na cadeira da esquerda.
- A Susana não se quer sentar à esquerda do Pedro.

Será possível sentar os convidados? Como?
```

Modele o problema em Lógica Proposicional e use o Z3 para o resolver. Não se esqueça que todas as pessoas devem ficar sentadas e que só é possível sentar uma pessoa por cadeira.

Sugestão: Crie uma variável proposicional (com nome sugestivo) para cada par (p, c), onde p é uma pessoa e c uma cadeira. Se a pessoa p ficar sentada na cadeira c o valor da variável respectiva será `True`, caso contrário será `False`. Em alternativa, pode também criar um dicionário `v` de variáveis proposicionais de tal forma que `v[p][c]` corresponde à variável do par (p, c).

[ ]:
```
# completar
```