Z3

Adam Krasuski

Plan

- SAT, SMT – definicje
- Z3 – opis
- Zastosowania oraz przykłady

SAT

- Problem spełnialności
- Formuła logiczna (**Boole'owska**)
- Wersja decyzyjna oraz pełna
- `(x1 ^ x2) <=> (x2 v x1)`
- NP-zupełny, nawet 3-SAT

SMT

- Satisfiability Modulo Theories
- Dowolna formuła rachunku predykatów pierwszego rzędu
- Dodatkowo kwantyfikatory, symbole funkcyjne:

$$\neg(\exists x)(S(x) \wedge (\forall y)(I(y) \implies C(y, x)))$$

Przykładowe teorie

- Bitvector
  - np. 32-bitowe zmienne, (+, -, *, /, %, ^, <<, ...)
- Arytmetyka liniowa
- Teoria tablic (array)
  - X = store(Y, index, value)
- Arytmetyka nieliniowa
  - sin, log, wielomiany, ...
- I wiele innych (teoria typów, napisów, ...)

Złożoność SMT

- Bitvector
  - jak SAT, NP - zupełny
- Arytmetyka nieliniowa
  - Nierozstrzygalna (uncomputable)!

Złożoność SMT

- Bitvector
  - jak SAT, NP - zupełny
- Arytmetyka nieliniowa
  - Nierozstrzygalna (uncomputable)!
  - Nawet na intach

Z3

- SMT Solver
- Microsoft Research
- Licencja MIT
- C++, bindingi

Działanie

- Input: formuła
- Output:
  - “sat”, x=1, y=true
  - “unsat”
  - “unknown”
    - OOM, TLE,
    - nierozstrzygalny,
    - niezaimplementowane

Format wejścia (SMT-LIB)

- Problem: `(x1 v x2) <=> (x1 ^ x2)`
- Wejście:

```
(declare-const x1 Bool)
(declare-const x2 Bool)
(assert (= (or x1 x2) (and x1 x2)))
(check-sat)
(get-model)
```

Format wyjścia

- **`$ z3 simple.z3`**

```
sat
(model
  (define-fun x2 () Bool
    true)
  (define-fun x1 () Bool
    true)
)
```

Programowanie

- Składnia Lispowa i RPN (or x y) niewygodne
- C++
- Bindingi

Python

```
from z3 import *
x1 = Bool('x1')
x2 = Bool('x2')
solve(Or(x1, x2) == And(x1, x2))
```

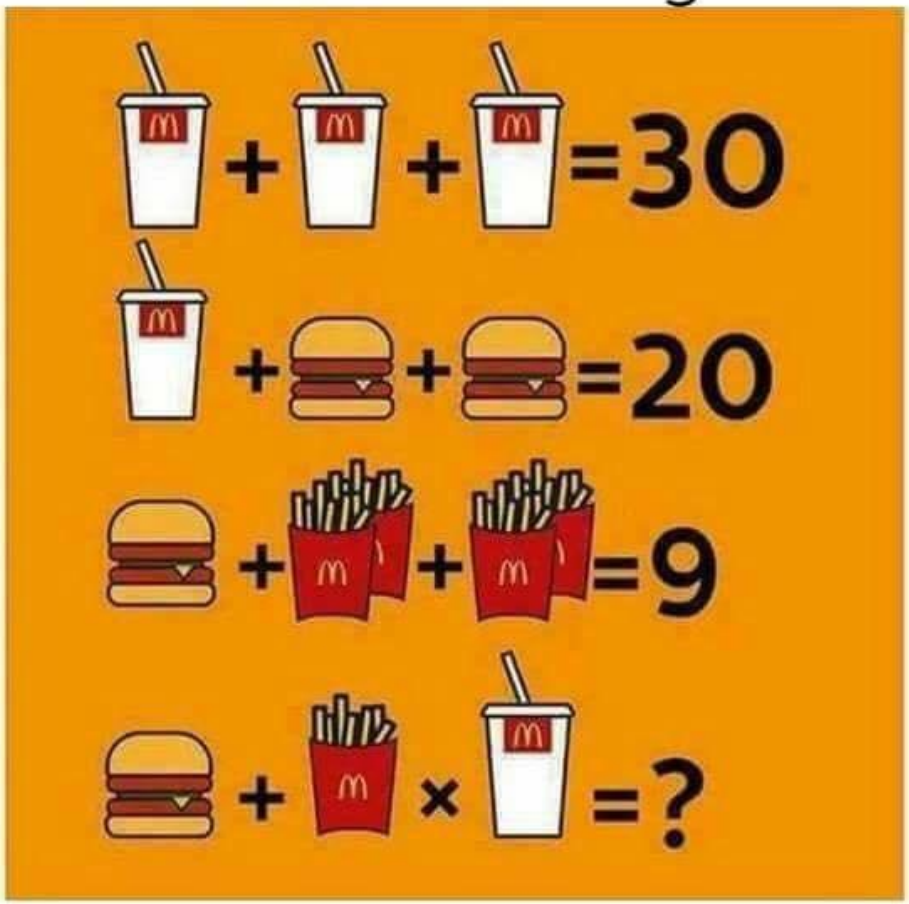
```
$ python simple.py
[x2 = True, x1 = True]
```

Zastosowania

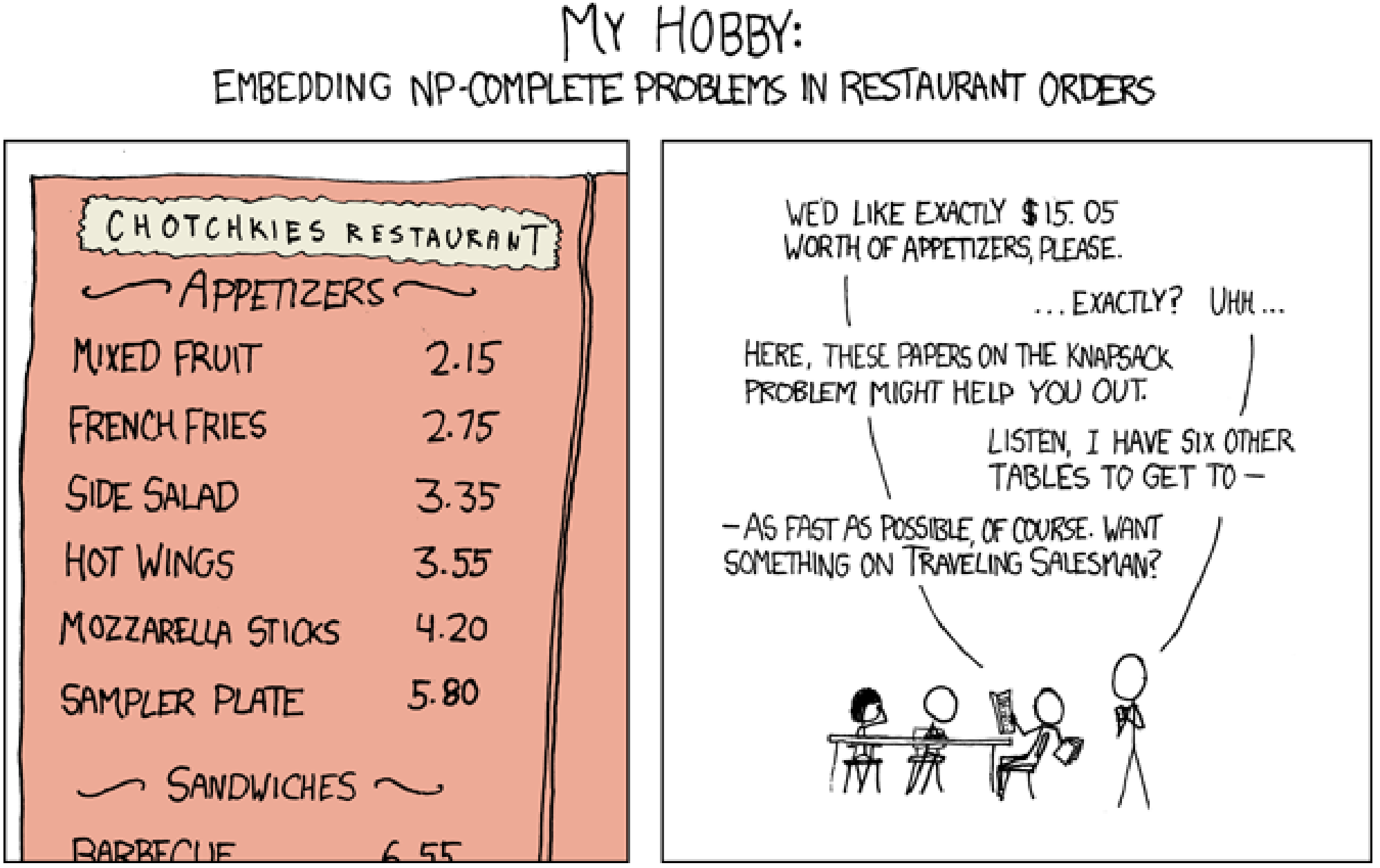
Proste problemy

```
rom z3 import *
h = Real("shake")
am = Real("hamburger")
r = Real("fries")
es = Real("result")
olve(
 sh + sh + sh == 30,
 sh + ham + ham == 20,
 ham + 2*fr + 2*fr == 9,
 ham + fr * sh == res)
 python mac.py
result = 15, fries = 1,
hamburger = 5, shake = 10]
```

98% fails.
Share first
Only for geniuses.
Answer fast if u r a genius.



Trudne problemy



Trudne problemy (plecak)

```
from z3 import *
a, b, c, d, e, f = Ints('a b c d e f')
s = Solver()
s.add(
     215*a + 275*b + 335*c + 355*d + 420*e
  + 580*f == 1505,
     a>=0, b>=0, c>=0, d>=0, e>=0, f>=0,
     a<=10,b<=10,c<=10,d<=10,e<=10,f<=10)
print s.check()
print s.model()
$ python xkcd.py
[f = 0, b = 0, a = 7, c = 0, d = 0, e = 0]
```

Wiele rozwiązań

```
while s.check() == sat:
    m = s.model()
    print m
    s.add(Or([
        var() != m[var]
        for var in m]))
```

```
$ python xkcd_many.py
[f = 0, b = 0, a = 7, c = 0, d = 0, e = 0]
[f = 1, b = 0, a = 1, c = 0, d = 2, e = 0]
```

Przewidywanie LCG

- Windowsowy rand():

```
int rand() {
    uint32_t a = 1103515245;
    uint32_t c = 12345;
    state = a * state + c;
    return (state >> 16) & 0x7FFF;
}
int main(){
  int i;
  srand(time(NULL));
  for (i=0; i<10; i++)
    printf ("%d\n", rand()%100);
  };
  generate_keys();
}
```

Przewidywanie LCG

```
from z3 import *

gen = [74, 42, 34, 33, 34, 71, 0, 55,
  56, 41]
state = BitVec("state", 32)
s = Solver()
for i in range(10):
    state = 1103515245 * state + 12345
    s.add(((state >> 16) & 0x7FFF) % 100
  == gen[i])
print s.check()
print s.model()
```

- **$ python lcg_solve.py**

```
sat
[state = 1544128614]
```

Manipulacja LCG

```
t = time.time()
s.add(state > t, state < t + 60*60*24)
for i in range(4):
  state = 1103515245 * state + 12345
  s.add(((state >> 16) & 0x7FFF) % 10 == 9)
print s.check()
m = s.model()
print "Wait", m[initial].as_long() - t, "seconds"
$ python lcg_max.py
sat
Wait 26152.6894641 seconds
```

Dowodzenie poprawności

- Kwantyfikator ogólny:

$$\underset{x}{\forall}\, f(x) == g(x)$$

- Negacja ma szczególny:

$$\underset{x}{\exists}\, f(x) \neq g(x)$$

- Interpretacja: szukanie kontrprzykładu

Xor swap

```
from z3 import *

x0, y0 = BitVecs('x y', 32)

x, y = x0, y0
x = x ^ y
y = x ^ y
x = x ^ y

s = Solver()
s.add(Or(x != y0, y != x0))
print s.check()
$ python swap.py
unsat
```

Sprawdzanie poprawności – bit twiddling hacks

```
return (v & (v - 1)) == 0;
```

Sprawdzanie poprawności – bit twiddling hacks

```
return (v & (v - 1)) == 0;
```

```
  111000
& 110111
--------
  110000

  100000
& 011111
--------
  000000
```

Sprawdzanie poprawności – bit twiddling hacks

```
(v & (v - 1)) == 0;
```

- Sprawdza czy v jest potęgą dwójki

```
from z3 import *
v, p = BitVecs('v p', 32)
s = Solver()
s.add(
   ((v & (v - 1)) == 0) !=
   Exists(p, And(p>=0, p<32,
      (BitVecVal(1, 32) << p) == v))
)

print s.check()
print s.model()
$ python bit.py
sat
[v = 0]
```

Sprawdzanie poprawności – bit twiddling hacks

```
(v & (v - 1)) == 0 && v;
```

- Sprawdza czy v jest potęgą dwójki

```
from z3 import *
v, p = BitVecs('v p', 32)
s = Solver()
s.add(
  And((v & (v - 1)) == 0, v != 0) !=
  Exists(p, And(p>=0, p<32,
    (BitVecVal(1, 32) << p) == v))
)
print s.check()
print s.model()
$ python bit_fixed.py
unsat
```

Wyrażenia regularne

- Spełnij: `\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}`
  - https://github.com/blukat29/regex-crossword-solver
  - `solve_regex("(U|O|I)*T[FRO]+", 5)`
  - `'IIITF'`
- Czy regex A == regex B?
  - Klasyczny algorytm z kompilacją do FSM
  - Ale regex z backref to nie regex

Optymalizacja

```
from z3 import *

opt = Optimize()
x, y, f = Ints('x y f')
opt.minimize((x-1)*(x-1) + y*y)
print opt.check()
print opt.model()
$ python opt.py
sat
[x = 1, y = 0]
```

Optymalizacja

```
from z3 import *

opt = Optimize()
x, y, f = Reals('x y f')
opt.minimize((x-1)*(x-1) + y*y)
print opt.check()
print opt.model()
$ python opt.py
unknown
[x = 0, y = 0]
```

Z3 - inne

- Tablice (array)
  - Bardziej mapy [K] ⟶ [V]

```
A = Array('A', IntSort(), IntSort())
x, y = Ints('x y')
solve(A[x] == x, Store(A, x, y) == A)
```

- Kwantyfikatory (ForAll, Exists)
- Funkcje – często unknown
- Napisy (string) oraz ogólne ciągi (sequences)
  - Python :(
  - Indexof, contains, concat, prefixof, ...

Inne

- Generacja unit testów
  - Coverage (if, for n razy)

Inne

- Generacja unit testów
  - Coverage (if, for n razy)
- Fuzzing

Inne

- Generacja unit testów
  - Coverage (if, for n razy)
- Fuzzing
- Obchodzenie zabezpieczeń (keygen)
  - angr
    - pamięć(t): array,
    - reg(t) – zmienne,
    - PC(t) == target – szukana formuła

Inne

- Generacja unit testów
  - Coverage (if, for n razy)
- Fuzzing
- Obchodzenie zabezpieczeń (keygen)
  - angr
    - pamięć(t): array,
    - reg(t) – zmienne,
    - PC(t) == target – szukana formuła
- Synteza programów
  - W następnym odcinku

Pytania

- Linki:
  - Z3 – źródła i dokumentacja: https://github.com/Z3Prover/z3
  - SAT SMT by example: https://github.com/DennisYurichev/SAT_SMT_by_example
  - Bit twiddling hacks: https://graphics.stanford.edu/~seander/bithacks.html
  - Moje kody: http://github.com/akrasuski1/presentation-z3