Introduction to SMT with Z3

Satisfiability Modulo Theories

- Decision problem for logical formulas similar to SAT
- Additionally supports theory fragments:
  - arithmetic, integers, reals
  - bit-vectors, uninterpreted functions, arrays
  - quantifiers, custom types
- Z3 is a solver for SMT formulas
  - It provides a great API for prototyping

First Steps with Z3 any Python

```
from z3 import Solver, Bools, Or, Not
from z3 import sat as SAT

tie, shirt = Bools('tie shirt')    # create boolean variables

solver = Solver()                  # create a solver
solver.add(Or(tie, shirt))         #   assert  tie or shirt
solver.add(Or(Not(tie), shirt))    #   assert  !tie or shirt
solver.add(Or(tie, Not(shirt)))    #   assert  tie or !shirt

if solver.check() == SAT:          # check if satisfiable
    print(solver.model())          # print the solution
```

Working with Integers and Arithmetic

```python
from z3 import Solver, Int
from z3 import sat as SAT

x, y = Int('x'), Int('y')           # create integer variables

solver = Solver()                   # create a solver
solver.add(x + y == 42)             #   assert  x + y = 42
solver.add(x < 6 * y)               #   assert  x < 6y
solver.add(x % 2 == 1)              #   assert  x == 1 mod 2

if solver.check() == SAT:           # check if satisfiable
    m = solver.model()              # retrieve the solution
    print(m[x] + m[y])              # print symbolic sum

# hint: use m[x].as_long() to get python integers
```

Bit-vectors as a Model for Memory

```python
from z3 import Solver, BitVec, Extract
from z3 import sat as SAT

x = BitVec('x', 8)                         # create 8-bit variable

# Overflow and underflow semantics

solver = Solver()                          # create a solver
solver.add(x + 5 < x - 10)                 #   assert  x + 5 < x - 10

if solver.check() != SAT: exit(1)   # check if satisfiable
m = solver.model()                         # retrieve the solution
print(m[x])                                # print result
for i in range(8):
    print(m.eval(Extract(i, i, x))) # print all bits
```

Signedness and Size Matter

```
from z3 import Solver, BitVec, ULE, ZeroExt, LShR
from z3 import sat as SAT

x = BitVec('x', 8)                        # create 8-bit variable
y = BitVec('y', 16)                       # create 16-bit variable

# Signedness and size semantics

solver = Solver()                         # create a solver
solver.add(ULE(x + 5, x - 10))            #   assert  x + 5 < x - 10
x0 = LShR(ZeroExt(8, x), 5)               #   x0 = x << 5
solver.add(y > x0)                        #   assert  y > (x << 5)

if solver.check() != SAT: exit(1)         # check if satisfiable
m = solver.model()                        # retrieve the solution
print(m[x], m[y])                         # print result
```

Quantifiers and Bounded Variables

```
from z3 import Solver, ForAll, Exists
from z3 import Int, Implies, And
from z3 import sat as SAT

x, y = Int('x'), Int('y')                    # create integers

solver = Solver()                            # create a solver
f = ForAll([x], Implies(x > 5, x > 0))
solver.add(f)                                #   assert  ∀ x > 5. x > 0
e = Exists([x, y], And(y > 10, x < y))
solver.add(e)                                #   assert  ∃ x,y. y > 10 && x <
y
if solver.check() != SAT: exit(1)            # check if satisfiable

# There is no model, all variables are bounded
```

We can Mix Bounded and Unbounded Variables

```
from z3 import Solver, ForAll, Exists
from z3 import Int, Implies, And
from z3 import sat as SAT

x, y = Int('x'), Int('y')                 # create integers

solver = Solver()                         # create a solver
solver.add(And(x > 42, x < 56))           #   assert x > 42 && x < 56
f = ForAll([y], Implies(y + x < 100, y < 50))
solver.add(f)                             #   assert ∀ y + x < 100. y < 50
if solver.check() != SAT: exit(1)         # check if satisfiable
print(solver.model())                     # print the solution
```

Custom Datatypes

```python
from z3 import Solver, Datatype, Const
from z3 import sat as SAT

C = Datatype("Colour")                      # declare the datatype
for c in ["red", "green", "blue"]:
    C.declare(c)                            # declare constructors
CSort = C.create()                          # create the sort
green = CSort.constructor(1)()              # fetch a constructor
solver = Solver()                           # create a solver
x = Const("x", CSort)                       # create a colour variable
solver.add(x != green)                      #   assert x != green
solver.add(x != CSort.red)                  #   assert x != red
if solver.check() != SAT: exit(1)           # check if satisfiable
print(solver.model())                       # print the solution
```

Uninterpreted Functions

- Generally, functions look like this:
  - $\boldsymbol{f} : \mathbf{A_0} \times \ldots \times \mathbf{A_n} \rightarrow \mathbf{B}$
  - Inputs are in sets $\mathbf{A_i}$
  - ***f*** maps them to an output in **B**
- Uninterpreted functions are just *'unknown'*
  - The solver has to decide them
  - You can think of them as a lookup-table
  - E.g. if $a_0 = 0$ and $a_1 = 5$, set output to 7

Uninterpreted Functions in Z3

```
from z3 import Solver, Function, Int, IntSort
from z3 import sat as SAT

x, y = Int('x'), Int('y')                    # create integers
f = Function("f", IntSort(), IntSort())  # f : N -> N

solver = Solver()                          # create a solver
solver.add(f(f(x)) == x)                   #   assert f(f(x)) = x
solver.add(f(x) == y)                      #   assert f(x) = y
solver.add(x != y)                         #   assert x != y
if solver.check() != SAT: exit(1)          # check if satisfiable
m = solver.model()                         # get the solution
print(m.eval(f(5)))                        # evaluate f(5)
```

Uninterpreted Functions as Relations

```
from z3 import Solver, Int, IntSort, BoolSort, Function
from z3 import ForAll, And, Implies, sat as SAT

# create an integer and a function
x, f = Int("x"), Function("f", IntSort(), IntSort(), BoolSort())
solver = Solver()                                       # create a solver
fa = ForAll([x], Implies(And(x > 0, x < 4), f(x, x)))
solver.add(fa)                                          #   assert ∀ x in (0,4). f(x,x) = 1
if solver.check() != SAT: exit(1)                       # check if satisfiable

m = solver.model()                                      # get the solution
for i in range(0,5):                                    # print the relation table
    vs = [int(bool(m.eval(f(i, j)))) for j in range(0,5)]
    print(("%d " * 5) % tuple(vs))                      # print one row
```