Logic and Computability

Lecture 5

SCIENCE
PASSION
TECHNOLOGY

Introduction to Z3

Bettina Könighofer
bettina.koenighofer@iaik.tugraz.at

Stefan Pranger
stefan.pranger@iaik.tugraz.at

What is Z3?

- Solver for Satisfiability Modulo Theories

What is Z3?

- Solver for **Satisfiability** *Modulo Theories*
  - we know how to check satisfiability ✓
  - … until now: Only propositional logic!

- Z3 allows us to efficiently answer decision problems including
  - Integers, Reals, Arithmetic
  - BitVectors, uninterpreted Functions, Arrays,
  - etc.

- More on Theories starting from next week
- Today: Basics Principles of Z3 and First Problems

Background

- Developed by Microsoft Research
  - https://github.com/Z3Prover/z3

Christoph Wintersteiger

Lev Nachmanson

Leonardo de Moura

Nikolaj Bjørner

- SMT-LIB2 – A standardized language for Problems in SMT

Principles

- Is $\neg a \wedge (a \vee b)$ satisfiable?
- What do we need to describe a problem for the solver?
  - Variables (of a specific Sort),

```
(declare-const a Bool)
(declare-const b Bool)
```

  - Constraints, and

```
(assert (not a) )
(assert (or a b) )
```

  - Checking for Satisfiability

```
(check-sat)
```

A Simple Example in SMT-LIB2

```
(declare-const a Bool)
(declare-const b Bool)
(assert (not a) )
(assert (or a b) )
(check-sat)
(get-model)
```

Background

- Developed by Microsoft Research
  - https://github.com/Z3Prover/z3

Christoph Wintersteiger

Lev Nachmanson

Leonardo de Moura

Nikolaj Bjørner

- `SMT-LIB2` – A standardized language for Problems in SMT
- API for `C++`, `Python`, `Julia`, etc.

Installing

- We will use the `Python API`:
  - `pip install z3-solver`

- Optionally, you may install z3 natively:
  - `sudo apt-get install z3` (Via aptitude for Ubuntu, etc.)
  - https://www.nuget.org/packages/Microsoft.Z3/ (Windows)
  - https://jfmc.github.io/z3-play (online)

Python API

- User-friendly interface for `SMT-LIB2`
- Used in the Programming Assignment

- Variables (of a specific Sort),

```
(declare-const a Bool)          →    a = Bool("a")
(declare-const b Bool)               b = Bool("b")
```

- Constraints, and

```
                                     solver = Solver()
(assert (not a) )               →    solver.add(Not(a))
(assert (or a b) )                   solver.add(Or(a,b))
```

- Checking for Satisfiability

```
(check-sat)                     →    solver.check()
```

Python API

```
from z3 import *

a, b = Bools("a b")

solver = Solver()
solver.add(Not(b))
solver.add(Or(a,b))

print(solver.sexpr())
result = solver.check()
model = solver.model()
print(result)
print(model)
```

Python API

- Constraints

```
(assert (not a) )        →    solver.add(Not(a))
(assert (or a b) )       →    solver.add(Or(a,b))
```

- Provides Methods for Connectives:
  - **`And()`**, **`Or()`**, **`Not()`**, **`Implies()`**, ==, ^, etc.
- Method to check whether two statements can be distinct:
  - **`Distinct(a,b)`**
- Operator overloading:
  - +, -, >>, <<, etc.

- Reference: https://z3prover.github.io/api/html/namespacez3py.html

A First Example

- We want to show that the following statements are equal:
  - $p \rightarrow q$
  - $\neg p \vee q$

A First Example

- $p \rightarrow q == \neg p \vee q$?

```
from z3 import *

solver = Solver()
a, b = Bools("a b")
l, r = Bools("l r")

solver.add(l == Implies(a, b))
solver.add(r == Or(Not(a), b))
solver.add(Distinct(r,l) )

result = solver.check()
print(result)
```

Back to SMT-LIB2

- $p \rightarrow q == \neg p \land q$?

```
from z3 import *

solver = Solver()
a, b = Bools("a b")
l, r = Bools("l r")

solver.add(l == Implies(a, b))
solver.add(r == Or(Not(a), b))
solver.add(Distinct(r,l))
print(solver.sexpr())

result = solver.check()
print(result)
```

BitVectors

- Z3 allows us to use so-called *theories*
- We have a first look at bitvectors

- Syntax:
  - `bv = BitVector("bv", <size>)`

- `BitVector`s respect under-/overflow behaviour!
  - In contrast to Z3’s integers

Operations on BitVectors

- The BitVector Sort respects overloaded operators:
  - <,>, <=, +, -, <<, >>, /, etc.
  - Caution: These are signed interpretations
  - Use `ULT`, `UGT`, `ULE` for unsigned interpretations

Equivalence Checking for BitVectors

- We want to prove the equivalence of the following
  - `(((y & x)*-2) + (y + x))`
  - $x \oplus y$

Weird XOR

```
from z3 import *

x = BitVec('x', 32)
y = BitVec('y', 32)

output = BitVec('output ', 32)

s = Solver()
s.add(x^y==output)
s.add(Distinct(((y & x)* -2) + (y + x),output))

print(s.check())
```

Operations on BitVectors

- The BitVector Sort respects overloaded operators:
  - <,>, <=, +, -, etc.
  - Caution: These are signed interpretations
  - Use `ULT`, `UGT`, `ULE` for unsigned interpretations

- Overflow and Underflow
  - `BVAddNoOverflow, BVAddNoUnderflow`
  - `BVMulNoOverflow, BVMulNoUnderflow`

Overflow Behaviour

- We want to check whether the statement TODO
  - `(x + 1 < x - 1)`

Variables in a Satisfying Model

- Variables and Expressions are stored in z3-specific classes
- We can use `solver.model().decls()` to iterate through all declared variables
  - Use .as_long() to convert a BitVector to a Python Integer

```
model = solver.model()
for var in solver.model.decls():
    print(f"{var}: {model[var]}(:{type(model[var])})")
```

Overflow Behaviour

- We want to check whether the statement TODO
  - `(x + 1 < x - 1)`

- We need to add
  - `BVNoOverflow(x,1,True)`
  - `BVNoUnderflow(x,1,True)`

- Functions that evaluate to `False` when Over-/Underflow would occur in the model

Assignment Sheet

- 4 Exercises + 1 Bonus Exercise
- You are allowed to work in groups of 2
  - If you do so, please add your information into the README
- Deadline: 05. 06. 2024

Outline – Part II

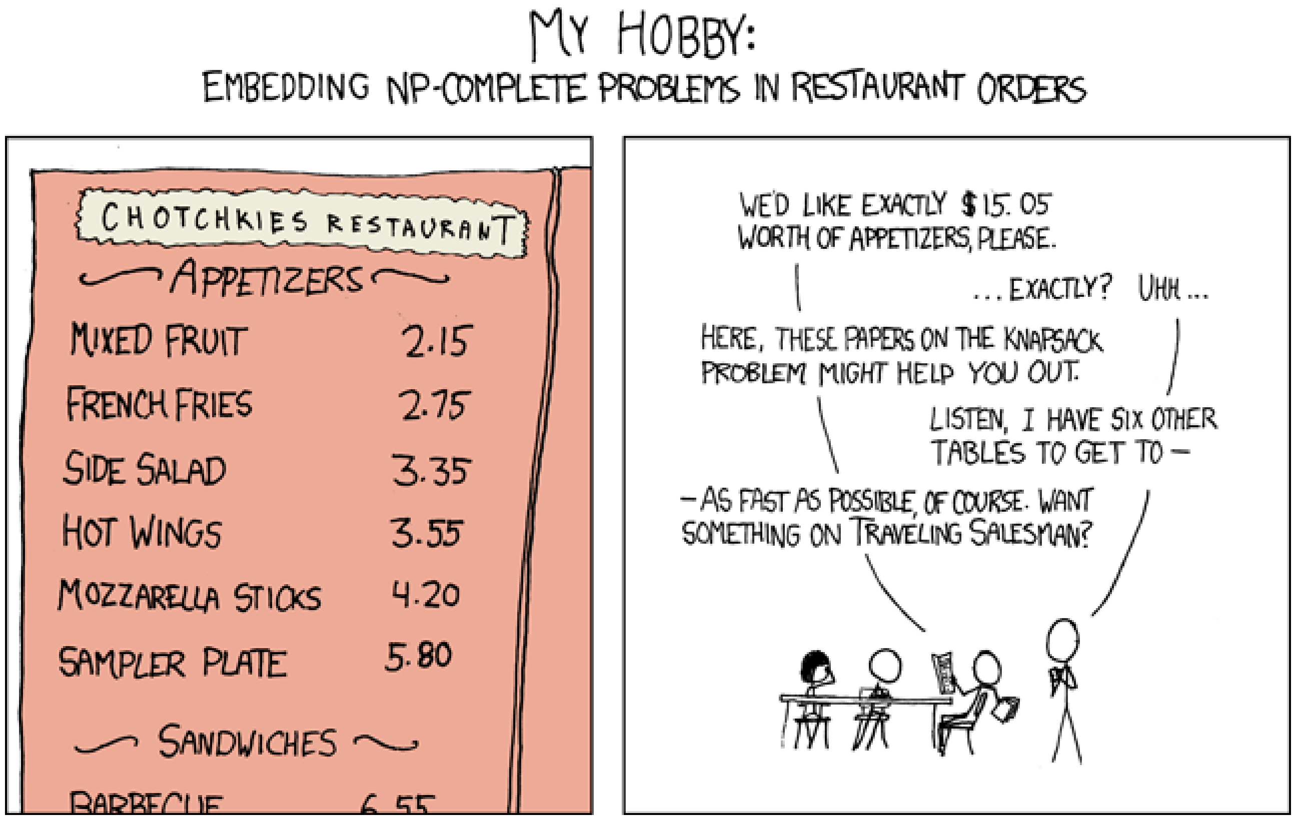
- `IntSort +` Z3 Built-in Sorts
- Quantifiers
- Custom Sorts
- Uninterpreted Functions

Working with Integers

- `IntSort`
  - <,>, <=, ==, +, -, etc.

Working with Integers

- IntSort
  - <,>, <=, ==, +, -, etc.



Example

```
#!/usr/bin/python3

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)
result = s.check()
if result == sat:
    print(s.model())
```

Variables in a Satisfying Model

- Variables and Expressions are stored in z3-specific classes
- We can use `solver.model().decls()` to iterate through all declared variables
  - Use `.as_long()` to convert a BitVector, Int, Real, etc. to a Python Integer

Example contd.

```
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print(m)
        results.append(m)
        block = [a != m[a].as_long(), b != m[b] .as_long(), c != m[c] .as_long(), d != 
m[d] .as_long(), e != m[e] .as_long(), f != m[f] .as_long()]
        """
        #Different approach: Iterate over all entries in the model
        block = []
        for d in m.decls():
            print(d, type(d), d(), type(d()), m[d], type(m[d]))
            c = d()
            block.append(c != m[d].as_long())
        """
        s.add(Or(block))
    else:
        print ("All results enumerated, total=", len(results))
        break
```

Z3 Built-in Sorts

- `BoolSort`, `BitVecSort`, `IntSort`, `RealSort`
- `Sequences, Strings`
- `Arrays`

Quantifiers

- Z3 offers `ForAll()` and `Exists()`
- Usage: `ForAll(<vars>, <formula>)`

Example

```
from z3 import *

x, y = Ints("x y")

solver = Solver()
solver.add(ForAll([x,y], Implies(And(x<0,y<0), x+y<0)))
#solver.add(ForAll([x,y], Implies(And(x<0,y<0), x+y>0)))

result = solver.check()
print(solver.sexpr())
print(result)
```

Example

▶ **15.** [*M26*] J. H. Quick noticed that $((x+2)\oplus 3)-2=((x-2)\oplus 3)+2$ for all x. Find all constants a and b such that $((x+a)\oplus b)-a=((x-a)\oplus b)+a$ is an identity.

Example

▶ **15.** [*M26*] J. H. Quick noticed that $((x+2)\oplus 3)-2=((x-2)\oplus 3)+2$ for all x. Find all constants a and b such that $((x+a)\oplus b)-a=((x-a)\oplus b)+a$ is an identity.

- We want to use Z3 to find all constants , s.t.
  - $\forall x\ \big((x+a)\oplus b\big)-a=\big((x-a)\oplus b\big)+a$

Example

- $\forall x\,\big((x+a)\oplus b\big)-a=\big((x-a)\oplus b\big)+a$

```
from z3 import *
s = Solver()
a, b = BitVecs('a b', 4)
x    = BitVec('x', 4)

s.push()
s.add(ForAll(x, ((x+a)^b)-a == ((x-a)^b)+a ))

results=[]
while True:
  if s.check() == sat:
    m = s.model(); results.append(m)
    block = [a != m[a].as_long(), b != m[b].as_long()]
    s.add(Or(block))
  else:
    print ("results total=", len(results))
    break
```

Example

- $\forall x \left((x + a) \oplus b \right) - a = \left((x - a) \oplus b \right) + a$
  - Let’s also use Z3 to find constants such that the equality does not hold

Example

- $\forall x\left(\left((x+a)\oplus b\right)-a=\left((x-a)\oplus b\right)+a\right)$
  - Let’s also use Z3 to find constants such that the equality does not hold

- Use `solver.push()` and `solver.pop()` to store and restore solver states

Example

- $\forall x\ \big((x+a) \oplus b\big) - a = \big((x-a) \oplus b\big) + a$
  - Let's also use Z3 to find constants such that the equality does not hold

```
from z3 import *
s = Solver()
a, b = BitVecs('a b', 4)
x    = BitVec('x', 4)

s.push()
s.add(ForAll(x, ((x+a)^b)-a == ((x-a)^b)+a ))
. . .
s.pop()
s.add(Exists(x, ((x+a)^b)-a != ((x-a)^b)+a ))
result = s.check()
print(result)
print(s.sexpr())
if result == sat:
    print(s.model())
```

Custom Sorts – Datatypes

- Beyond the built-in Sorts
- Datatypes allow us to define more complex data structures

Custom Sorts – Datatypes

- Beyond the built-in Sorts
- Datatypes allow us to define more complex data structures

- Simple Case: Enum

```
ColoursDatatype = Datatype("Colour")
ColoursDatatype.declare("RED")
ColoursDatatype.declare("GREEN")
ColoursDatatype.declare("BLUE")
ColoursDatatype.declare("MAGENTA")
ColoursSort = ColoursDatatype.create()

x = Const("x", ColoursSort)
```

Uninterpreted Functions

- Generally, we have:
  - $f: A_0 \times \ \ldots \times A_n \rightarrow B$
  - f maps values from $A_0 \times \ \ldots \times A_n$ to B

Uninterpreted Functions

- Generally, we have:
  - $f: A_0 \times \ \ldots \times A_n \rightarrow B$
  - f maps values from $A_0 \times \ \ldots \times A_n$ to B
- Uninterpreted Functions have no know “structure”
  - Z3 decides the output based on the constraint
  - f can be seen as a lookup-table

Uninterpreted Functions

- Generally, we have:
  - $f: A_0 \times \ \dots \times A_n \rightarrow B$
  - f maps values from $A_0 \times \ \dots \times A_n$ to B
- Uninterpreted Functions have no know “structure”
  - Z3 decides the output based on the constraint
  - f can be seen as a lookup-table

- `f  = Function('f', IntSort(), IntSort())`

Seating Arrangement Problem

- Problem Setting:
  - You have to arrange a set of guests on one large table
  - Some *need to* to be seated together
  - Some *must not* be seated together

Seating Arrangement Problem

- Problem Setting:
  - You have to arrange a set of guests on one large table
  - Some *need to* to be seated together
  - Some *must not* be seated together

- We can use an uninterpreted function as a mapping from guests to seats at the table!