03/22/19 Recitation Notes

17-355/17-665/17-819: Program Analysis (Spring 2019)

Jenna Wise*
jlwise@andrew.cmu.edu

1 Reminders

- Homework 7 is due next **Tuesday, March 26, 2019 at 11:59pm**. Instructions can be found on the course website

2 SMT Solvers: Z3

There are a variety of different SMT solver available for use. Some of the popular ones used for Hoare-style verification are:

- Alt-Ergo
- CVC3
- CVC4
- Z3

The Wikipedia article on Satisfiability Modulo Theories has a more comprehensive list of SMT solvers and their corresponding features.

This document discusses Z3, as it will be the solver that you will use to complete homework 8.

2.1 Introduction to Z3

Z3 is a state-of-the art theorem prover from Microsoft Research. It can be used to check the satisfiability of logical formulas over the following theories (non-comprehensive list):

- Uninterpreted Functions and Constants
- Linear Arithmetic
- Non-linear Arithmetic
- Bitvectors
- Arrays
- Datatypes

*These recitation notes were developed together with Rijnard van Tonder, Jonathan Aldrich, and Claire Le Goues

- Quantifiers
- Strings

In these notes, we will cover the theories of Bitvectors and Uninterpreted Functions and Constants (EUF), as they will be helpful for completing homework 8. The Rise4fun Z3 Guide/Tutorial and Python Z3 Tutorial both describe the theory of Bitvectors, Uninterpreted Functions and Constants, and some of the other theories listed above in more detail and should be consulted as necessary. Links to these and other helpful Z3 resources can be found in Section 2.3 of these notes.[1]

You can use Z3 programmatically with various language specific APIs or online at https://rise4fun.com/Z3/. The code syntax is different in each of these settings, but the features remain largely the same. We present code snippets in the following sections in both the Rise4fun online syntax (SMT-LIB) and Python Z3 syntax (Z3Py). For using Z3 programmatically we recommend using the Python Z3 API, and instructions on how to set-up Python Z3 on your computer can be found in Section 2.2.

2.1.1 Basic Solving Commands

When we construct a formula or constraint that we want Z3 to solve, we must tell Z3 what it is and that we want it to determine the formula or constraint's satisfiability.

Listing 1: Solving in SMT-LIB

```
; Declares a constant of the given type Int named x
(declare-const x Int)

; Tells Z3 that we want to make sure x > 10 and x < 100 when we check
   satisfiability
; These two lines are equivalent to (assert (and (> x 10) (< x 100)))
(assert (> x 10))
(assert (< x 100))

; Tells Z3 to check for satisfiability of the asserted constraints
(check-sat)

; Used to retrieve an interpretation that makes all asserted
   constraints true when constraints are satisfiable
(get-model)
```

Listing 2: Solving in Z3Py

```
# Declares a constant of the given type Int named x
x = Int('x')

# Tells Z3 to solve the system of constraints, ie. check satisfiability
    and produce an interpretation that makes all of the constraints
   true
# Produces unsat if constraints are unsatisfiable
solve(x > 10, x < 100)
```

[1]The information in this section comes largely from the Rise4fun Z3 Guide/Tutorial and Python Z3 Tutorial.

```
# Alternatively, we could write
solve(And(x > 10, x < 100))
```

2.1.2 Bitvectors

Modern CPUs and main-stream programming languages use arithmetic over fixed-size bit-vectors. They implement the precise semantics of unsigned and of signed two-complements arithmetic. Z3 supports a number of functions and relations over bit-vectors, and we will cover the main ones in these notes.

Before presenting the functions and relations over bit-vectors, we first demonstrate how to create bit-vector variables and constants of various length.

Listing 3: Bit-vector creation in SMT-LIB

```
; (_ BitVec n) is the general syntax for creating bit-vector variables
   and constants of arbitrary length n

; Creates bit-vector with the value of the constant, #b0100, in binary
   format and with size 4 inferred from the number of characters (bit-
   vectors created from constants in binary or hexadecimal format will
   have their sizes inferred)
(display #b0100)

; Creates a bit-vector with hexadecimal value #x0a and size 8
(display #x0a)

; Creates a bit-vector with decimal value of 20 and size 32 (bit-
   vectors created from constants in decimal format must have their
   sizes provided)
(display (_ bv20 32))

; Bit-vectors are displayed in hexadecimal format if the bit-vector
   size is a multiple of 4, and in binary otherwise by default
; This command can be used to force Z3 to display bit-vector constants
   in decimal format
(set-option :pp.bv-literals false)

; Creates a bit-vector variable named x with 4 bits (of size 4)
(declare-const x (_ BitVec 4))
```

Listing 4: Bit-vector creation in Z3Py

```
# Creates a bit-vector variable in Z3 named x with 16 bits
x = BitVec('x', 16)

# Creates a bit-vector of size 32 containing the value 10
x = BitVecVal(10, 32)
```

Basic Bit-vector Arithmetic:

Listing 5: Bit-vector arithmetic in SMT-LIB

```
(simplify (bvadd #x07 #x03))  ; addition
(simplify (bvsub #x07 #x03))  ; subtraction
(simplify (bvneg #x07))       ; unary minus
(simplify (bvmul #x07 #x03))  ; multiplication
(simplify (bvurem #x07 #x03)) ; unsigned remainder
(simplify (bvsrem #x07 #x03)) ; signed remainder
(simplify (bvsmod #x07 #x03)) ; signed modulo
(simplify (bvshl #x07 #x03))  ; shift left
(simplify (bvlshr #xf0 #x03)) ; unsigned (logical) shift right
(simplify (bvashr #xf0 #x03)) ; signed (arithmetical) shift right
```

Listing 6: Bit-vector arithmetic in Z3Py

```
x = BitVecVal(10, 32)
y = BitVecVal(3, 32)

x + y     # addition
x - y     # subtraction
-x        # unary minus
x * y     # multiplication
x / y     # division
UDiv(x,y) # unsigned division
URem(x,y) # unsigned remainder
SRem(x,y) # signed remainder
x % y     # signed modulo
x << y    # shift left
LShR(x,y) # unsigned (logical) shift right
x >> y    # signed (arithmetical) shift right
```

Bitwise Operations:

Listing 7: Bit-vector bitwise operations in SMT-LIB

```
(simplify (bvor #x6 #x3))   ; bitwise or
(simplify (bvand #x6 #x3))  ; bitwise and
(simplify (bvnot #x6))      ; bitwise not
(simplify (bvnand #x6 #x3)) ; bitwise nand
(simplify (bvnor #x6 #x3))  ; bitwise nor
(simplify (bvxnor #x6 #x3)) ; bitwise xnor
```

Listing 8: Bit-vector bitwise operations in Z3Py

```
x = BitVecVal(10, 32)
y = BitVecVal(3, 32)

x | y  # bitwise or
x & y  # bitwise and
~x     # bitwise not
```

Predicates over Bit-vectors:

Listing 9: Bit-vector predicates in SMT-LIB

```
(simplify (bvule #x0a #xf0))  ; unsigned less or equal
(simplify (bvult #x0a #xf0))  ; unsigned less than
(simplify (bvuge #x0a #xf0))  ; unsigned greater or equal
(simplify (bvugt #x0a #xf0))  ; unsigned greater than
(simplify (bvsle #x0a #xf0))  ; signed less or equal
(simplify (bvslt #x0a #xf0))  ; signed less than
(simplify (bvsge #x0a #xf0))  ; signed greater or equal
(simplify (bvsgt #x0a #xf0))  ; signed greater than
```

Listing 10: Bit-vector predicates in Z3Py

```
x = BitVecVal(10, 32)
y = BitVecVal(3, 32)

ULE(x,y) # unsigned less or equal
ULT(x,y) # unsigned less than
UGE(x,y) # unsigned greater or equal
UGT(x,y) # unsigned greater than
x <= y   # signed less or equal
x < y    # signed less than
x >= y   # signed greater or equal
x > y    # signed greater than
```

2.1.3 Uninterpreted Functions and Constants

Unlike programming languages, where functions have side-effects, can throw exceptions, or never return, functions in Z3 have no side-effects and are total. That is, they are defined on all input values. This includes functions, such as division. Z3 is based on first-order logic.

Given a constraints such as $x + y > 3$, we have been saying that x and y are variables. In many textbooks, x and y are called uninterpreted constants. That is, they allow any interpretation that is consistent with the constraint $x + y > 3$.

More precisely, function and constant symbols in pure first-order logic are uninterpreted or free, which means that no a priori interpretation is attached. This is in contrast to functions belonging to the signature of theories, such as arithmetic where the function + has a fixed standard interpretation (it adds two numbers). Uninterpreted functions and constants are maximally flexible; they allow any interpretation that is consistent with the constraints over the function or constant.

To illustrate uninterpreted functions and constants let us introduce the uninterpreted integer constants (aka variables) x, y. Finally let f be an uninterpreted function that takes one argument of type (aka sort) integer and results in an integer value. The example illustrates how one can force an interpretation where f applied twice to x results in x again, but f applied once to x is different from x.

Listing 11: EUF example in SMT-LIB

```
(declare-const x Int)
(declare-const y Int)
(declare-fun f (Int) Int)
```

```
(assert (= (f (f x)) x))
(assert (= (f x) y))
(assert (not (= x y)))
(check-sat)
(get-model)
```

Listing 12: EUF example in Z3Py

```
x = Int('x')
y = Int('y')
f = Function('f', IntSort(), IntSort())
solve(f(f(x)) == x, f(x) == y, x != y)
```

The solution (interpretation) for f should be read as f(0) is 1, f(1) is 0, and f(a) is 1 for all a different from 0 and 1.

2.2 Set-up for Python Z3

Make sure you have an installation of Python working on your computer. You can download Python from https://www.python.org/.

Now download Z3 for your platform from the releases page. Unzip it in a directory of your choosing. You may need to set up a library path so that you can use the Python API:

```
export LD_LIBRARY_PATH=/path/to/z3/bin/      // For Linux
export Z3_LIBRARY_PATH=/path/to/z3/bin       // For Mac
```

Check that it works by doing:

```
cd /path/to/z3/bin/python
python example.py
```

This should result in the following:

```
sat
[y = 3/2, x = 4]
```

2.3 Helpful Z3 & Python Z3 Resources

- Z3 Github Repository
- Rise4fun Z3 Guide/Tutorial
- Z3 Online
- Tutorial on Programming Z3
- Python Z3 Tutorial
- Python Z3 API