Z3 Guide in Python

By: Lev Naiman

Last Modified: Saturday, Jan. 23, 2016

Based on: `http://www.cs.tau.ac.il/~msagiv/courses/asv/z3py/guide-examples.htm`

1 Introduction

Z3 is a high performance theorem prover and satisfiability modulo theories solver (SMT) developed at Microsoft Research. Z3 is used in many applications such as: software/hardware verification and testing, constraint solving, analysis of hybrid systems, security, biology (in silico analysis), and geometrical problems.

This guide will show you how to use the Python API for Z3, as well as a range of useful functions, common errors to avoid, and some more complex problems. It is recommended to try the examples while reading the guide.

2 Installation

1. Download the Z3 package from here `http://z3.codeplex.com/releases/view/101913` (32 bit version) or here `http://z3.codeplex.com/releases/view/101911` (64 bit version). Make sure you have the version that corresponds to your machine.

2. Decompress the file and find the path of the Z3 binaries. Lets assume you renamed the folder containing Z3 to "z3". Then the relative path to the binaries will be at `z3/bin`.

3. Add the absolute path to the Z3 binaries to both your PATH and PYTHONPATH variables. For example, my absolute path is `/Downloads/z3/bin`. To add this path to the path variables mentioned above, type in your shell (with your absolute path of course):

```
export PATH=\$PATH:~/Downloads/z3/bin
export PYTHONPATH=$PYTHONPATH:~/Downloads/z3/bin
```

4. Add the header `from z3 import *` to your python code. Note that if you were to restart your computer, the changes to the path variables will be lost, and you would have to redo step 2. This is why you may optionally install Z3 more permanenly by moving the binaries to a place like `/usr/bin`, or changing your start-up scripts so that the path variables include the binaries of Z3 where-ever you placed them.

We recommend using Python 2.7 with Z3 since some versions of Z3 don't support Python 3.3 yet. Make sure that you use the same architecture for Z3 as Python, e.g., 32-bit Z3 for 32-bit Python. For problems with Z3 installation and set up, first make sure that PYTHONPATH is set, e.g.,

```
export PYTHONPATH=$PYTHONPATH:~/Downloads/z3/bin
```

Sometimes you will encounter the error message

```
Z3Exception("init(Z3_LIBRARY_PATH) must be invoked before using Z3-python
```

Then you will need to include Z3 shared objects to your search path, or invoke init with the full path to the Z3 shared object (`*.so` Linux, `*.dylib` OSX, or `*.dll` Windows).

To invoke Z3 within python code add the header `from z3 import *`.

3 Getting Started

The following is a simple example that illustrates all the steps required in general to use Z3 as a solver:

```
from z3 import *

x = Int('x')
y = Int('y')

s = Solver()
s.add(x > 2, y < 10, x + 2*y == 7)

print s.check()
print s.model()
```

The function `Int('x')` creates an integer variable in Z3 named `x`. We then create a solver instance using `Solver()`. We add to the solver a system of constraints using `add`. The example above uses two variables `x` and `y`, and three constraints. The operators <, ≤, >, ≥, == and != are used for comparison. In the example above, the expression `x + 2*y == 7` is a Z3 constraint. Finally, to get the solution from Z3 we can ask if the constraints are satisfiable by calling `check()`. If the answer is `sat`, then we can print the model using `model()`.

The result from `check()` is either `sat` if the formula is satisfiable, `unsat` if the formula is unsatisfiable, and the third option is `unknown`. This happens in the case that a formula was not decidable and Z3 could not find a solution.

A simple command that allows avoiding the creation of an explicity solver and then invoking the model is `solve()`. This command takes a set of constraints and prints the model if it exists. For example, `solve(x+1=1)` prints `[x = 0]`.

Z3 contains within it many solvers, tactics, and pre-processes. It can do more than just solve for satisfiability. For example, you can use it to simplify formulas:

```
from z3 import *

x = Int('x')
y = Int('y')
print simplify(x + y + 2*x + 3)
print simplify(x < y + x + 2)
print simplify(And(x + 1 >= 3, x**2 + x**2 + y**2 + 2 >= 5))
```

4 Variable Declaration

Variables in Z3 can be declared with a variety of commands. Here are a few methods:

```
from z3 import *

x = Int('x')
r = Real('r')
a,b = Ints('a b')
y = IntVector('y', 5)
z = [ Int('z%s' % i) for i in range(5) ]
# Create a 3x3 "matrix" (list of lists) of integer variables
X = [ [ Int("x_%s_%s" % (i, j)) for j in range(3) ]
      for i in range(3) ]

print x.sort(),r.sort(),a.sort()
print y
print z
print X
pp(X)
```

A single variable is usually declared using a command that takes a string for a name and reflects its type. For example, `Int('x')` will create an integer Z3 variable called `x`. Multiple variables can be declared at once, such as with the function `Ints()`. An `IntVector` is a list of variables. Note that this is not some list object in Z3, but rather a Python list of Z3 variables. The expression `"x%s"% i` returns a string where `%s` is replaced with the value of `i`. A two dimensional list can be created in the same way, by making a list of lists in Python. Z3 also includes a pretty printer for its formulas called `pp()`.

5 Booleans

Z3 supports Boolean operators: `And, Or, Not, Implies` (implication), `If` (if-then-else). Bi-implications are represented using equality `==`. There are a number of other operators for other theories such as integers that also have boolean values, such as ≤. The following example shows how to solve a simple set of Boolean constraints.

```
from z3 import *

p = Bool('p')
q = Bool('q')
r = Bool('r')
x = Real('x')

solve(Implies(p, q), r == Not(q), Or(Not(p), r))
solve(Or(x < 5, x > 10), Or(p, x**2 == 2), Not(p))
```

Boolean functions such as `And` and `Or` can take a variable number of arguments. They can also take a list of formulas, creating a conjunction/disjunction of all the list items. Note that for the function `If` only the first argument needs to be boolean, and the rest can be any other type (but they must be the same type).

6 Arithmetic

Z3 supports real and integer variables. They can be mixed in a single problem. Z3 will automatically add coercions converting integer expressions to real ones when needed. You can also cast between types, such as with the command `ToReal()`. The following example demonstrates different ways to declare and use integer and real variables.

```
from z3 import *

x,y = Reals('x y')
a,b,c = Ints('a b c')

solve(a > b + 2,
      a == 2*c + 10,
      c + b <= 1000)

q = Q(1,3)
print x + c + 1
print ToReal(c) + x
print q, q.sort()
```

There is no datatype for rational numbers; instead their type will be real. To create a constant of rational value you can use the `Q()` function. For example, `Q(1,3)` is the value one-third, and its type is `Real`. Z3 also supports arbitrarily large numbers.

7 Arrays

Unlike lists of variables in Python, arrays are a datatype that is implemented in Z3. In Z3 arrays have simple operations: the expression `Select(a,i)` returns the value stored at position `i` of the array `a`; and `Store(a,i,v)` returns a new array identical to `a`, but on position `i` it contains the value `v`. A further useful command is `Update(a,i,v)` which returns an array that is the same as `a` except that at index `i` there is `v`.

An array is treated as a mapping between two types. An array is declared with a name, domain, and range: `a = Array('a',IntSort(),IntSort())`. Here we have declared an array that maps integers to integers. In other words, it is indexed by integers and the elements of the array are of integer type. Because arrays are treated as a mapping, they do not have a size in Z3; they can be thought of as infinite in length. A sample array is the mapping `[0 -> 1, else -> 0]`, which means that index 0 maps to 1, and all other integers map to 0. Two arrays are equal if all their elements agree. Note that arrays are datatypes, and there is no such thing as array assignment in Z3 as it would be in a programming language. Instead the values of the array should be described using select, store, and update. Here is an example of the use of arrays:

```
from z3 import *

i,j = Ints('i j')
a = Array('a',IntSort(), IntSort())
b = Update(a,3,1)

print a
print b
prove(Implies(j!=3, a[j]==a[j]))
prove(b[3]==1)
prove(Select(Store(a,0,5),0) == 5)
print eq(Select(a,0), a[0])
solve(a[0]==0)
```

For ease of use, Z3 arrays can be indexed in Python using square brackets. This is simply a syntactic abbreviation; for example `a[0]==5` is the same as writing `Store(a,0,5)`. Usually array operation are slower to solve for, as opposed to linear arithmetic for example. For this reason it is often more efficient to create a list of variables instead of an array in whenever possible .

8 Quantifiers

Quantifiers ∀ and ∃ can be used in Z3. If they are used, then it is no longer a decision procedure in general. That is, it is no longer the case that in finite time Z3 can decide whether a formula is satisfiable or unsatisfiable. However, in many cases Z3 can handle formulas that have quantifiers. A quantifier needs to be declared with a local variable and a body. Optionally, you may abbreviate quantification over multiple variables by using a list of local variables instead. For example:

```
from z3 import *

i,j = Ints('i j')
def prime_nat(n):
   return Not(Exists([i,j],And(i>1,j>1, n==i*j) ))

prove(ForAll(i, Exists(j, j>i)))
prove(prime_nat(5))
```

There are however very simple proofs, such as those that would require induction, that Z3 would not be able to handle.

9 Validity

A formula F is **valid** if F always evaluates to true for any assignment of appropriate values to its uninterpreted symbols. A formula/constraint F is **satisfiable** if there is some assignment of appropriate values to its uninterpreted symbols under which F evaluates to true. Validity is about finding a proof of a statement; satisfiability is about finding a solution to a set of constraints. Consider a formula F containing a and b. We can ask whether F is valid, that is whether it is always true for any combination of values for a and b. If F is always true, then `Not(F)` is always false, and then `Not(F)` will not have any satisfying assignment (i.e., solution); that is, `Not(F)` is unsatisfiable. That is, F is valid precisely when `Not(F)` is not satisfiable (is unsatisfiable). Alternately, F is satisfiable if and only if `Not(F)` is not valid (is invalid). The following example proves the deMorgan's law.

```
from z3 import *

p, q = Bools('p q')
demorgan = And(p, q) == Not(Or(Not(p), Not(q)))
prove(demorgan)
```

10 Functions

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.

In Z3, functions 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.

Here is an example that uses uninterpreted functions:

```
from z3 import *

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

These uninterpreted functions are not abbreviations, and nor can they be recursive like a function in Python. If you desire to have a function as an abbreviation of another formula, write a method in Python whose return value is a Z3 formula. For example, here is how to create a prime function:

```
from z3 import *

def prime_nat(n):
   return Not(Exists([i,j],And(i>1,j>1, n==i*j) ))
```

.

A function in Python can of course be recursive, and you can build any desired formula. While outright recursive functions are not included in Z3, one of its advanced features is support for fixed-points and relations.

11 Model

Once the solver has finished and determined that a formula is satisfiable, we often want get the values that made it so. A model is a mapping of variables to values that we obtain from a solver using `model()`. We can query the model in the same way we query a Python dictionary. Once we have a model we can also use it to evaluate expressions:

```
from z3 import *

x = Int('x')
y = Int('y')
```

```
a = Array('a', IntSort(),IntSort())

s = Solver()
s.add(x > 0, a[x]==y, a[x+1]==y+1)

if s.check() == sat:
    m = s.model()
    print m[x]
    print m[a]
    print a[0]
    print m.eval(x+y)
    print [ m.eval(a[j]) for j in range(3) ]
else:
    print "failed to solve"

solve(Distinct(x,y))
```

Notice that if we try to index a Z3 array we do not get a value back. Remember that indexing a Z3 array with square brackets is just a syntactic abbreviation. The result is a Z3 object. Instead if we wish to get the value at some index we should use **eval()**.

12 Avoiding Errors

A common error is failing to distinguish Z3 objects and Python values. For example, here we declare some variables

```
v = Int('v')
X = [Int('x_\%i' \%i) for i in range(n)]
```

and then do indexing as follows

```
X[v]
```

This results in an error saying a Z3 object cannot be used for indexing. The problem is that `X` is a Z3 array, as opposed to a Python list of Z3 integer variables. A Python list must be indexed with Python integers. A useful function to see whether something is a Z3 object is `is_ast()` which returns true if the input is a Z3 abstract syntax tree (Z3 object). The Python function **type()** is also useful.

Another subtle issue is that the Python value `True` is not the same as the Z3 object that represents true in Z3. The first is a boolean value in Python, and the latter an object. Often we can write formulas such as `Or(a,True)` without care for such a distinction, because Z3 will coerce the correct values. Sometimes this is not possible. For example, the following will result in an error

```
ForAll(a, True)
```

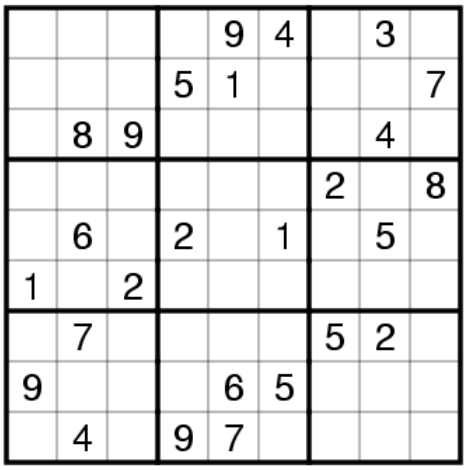
because Z3 expects the body to be a Z3 object and not a Python boolean. A simple work-around is to create a `True` value like so

```
ForAll(a, a==a)
```

13 Examples

13.1 Sudoku

The goal of the puzzle is to insert the numbers in the boxes to satisfy only one condition: each row, column and 3x3 box must contain the digits 1 through 9 exactly once.



The following example encodes the suduko problem in Z3. Different sukudo instances can be solved by modifying the matrix instance.

```
from z3 import *

# 9x9 matrix of integer variables
X = [ [ Int("x_%s_%s" % (i+1, j+1)) for j in range(9) ]
      for i in range(9) ]

# each cell contains a value in {1, ..., 9}
cells_c  = [ And(1 <= X[i][j], X[i][j] <= 9)
             for i in range(9) for j in range(9) ]

# each row contains a digit at most once
rows_c   = [ Distinct(X[i]) for i in range(9) ]

# each column contains a digit at most once
cols_c   = [ Distinct([ X[i][j] for i in range(9) ])
```

```
              for j in range(9) ]

# each 3x3 square contains a digit at most once
sq_c     = [ Distinct([ X[3*i0 + i][3*j0 + j]
                        for i in range(3) for j in range(3) ])
             for i0 in range(3) for j0 in range(3) ]

sudoku_c = cells_c + rows_c + cols_c + sq_c

# sudoku instance, we use '0' for empty cells
instance = ((0,0,0,0,9,4,0,3,0),
            (0,0,0,5,1,0,0,0,7),
            (0,8,9,0,0,0,0,4,0),
            (0,0,0,0,0,0,2,0,8),
            (0,6,0,2,0,1,0,5,0),
            (1,0,2,0,0,0,0,0,0),
            (0,7,0,0,0,0,5,2,0),
            (9,0,0,0,6,5,0,0,0),
            (0,4,0,9,7,0,0,0,0))

instance_c = [  X[i][j] == instance[i][j]
              for i in range(9) for j in range(9) if instance[i][j]!=0 ]

s = Solver()
s.add(sudoku_c + instance_c)
if s.check() == sat:
    m = s.model()
    r = [ [ m.evaluate(X[i][j]) for j in range(9) ]
          for i in range(9) ]
    print_matrix(r)
else:
    print "failed to solve"
```

13.2 Tiling

For this puzzle consider a string of characters from the alphabet {`A, B, C`}. There is a rule that says that three or more of the same consecutive character may be reduced to the empty string. The goal is to reduce the entire string of characters to the empty string as efficiently as possible. So if there are five consecutive A's, you must eliminate all of them at once, not over more than one step.

For example, the input string `ABBCBBBBCCCBAAAA` can be reduced as follows:

```
ABBC(BBBB)CCCBAAAA
ABB(CCCC)BAAAA
A(BBB)AAAA
(AAAAA)
nil
```

The solution to this puzzle is fairly involved and lengthy, but is included as part of this guide's package.

13.3 Useful Functions

A useful function to use when creating constrains is the built-in Python function `sum`. In the same way that `And()` can be applied to a list of formulas, so can sum be applied to a list of Z3 variables. The result of `sum([x,y])` is `0 + x + y`. Another supported arithmetic function is modulo `a%b`, although it often poses challenges for efficiency.

A further useful function is `Distinct()`, which takes a list of variables/formuals (or any number of arguments) and is satisfied when each item of the input is distinct. For example, a satisfying solution for `Distinct(x,y)` is `[y = 0, x = 1]`.

14 References and Further Reading

The Z3 Python implementation code: `http://code.metager.de/source/xref/microsoft/z3/src/api/python/z3.py`
Offical Z3 Guide (in SMT-LIB2): `http://rise4fun.com/Z3/tutorialcontent/guide#h26`