Advanced Topics

Please send feedback, comments and/or corrections to leonardo@microsoft.com. Your comments are very valuable.

Expressions, Sorts and Declarations

In Z3, expressions, sorts and declarations are called *ASTs*. ASTs are directed acyclic graphs. Every expression has a sort (aka type). The method `sort()` retrieves the sort of an expression.

```
x = Int('x')
y = Real('y')
print (x + 1).sort()
print (y + 1).sort()
print (x >= 2).sort()
```

The function `eq(n1, n2)` returns `True` if `n1` and `n2` are the same AST. This is a structural test.

```
x, y = Ints('x y')
print eq(x + y, x + y)
print eq(x + y, y + x)
n = x + y
print eq(n, x + y)
# x2 is eq to x
x2 = Int('x')
print eq(x, x2)
# the integer variable x is not equal to
# the real variable x
print eq(Int('x'), Real('x'))
```

The method `hash()` returns a hashcode for an AST node. If `eq(n1, n2)` returns `True`, then `n1.hash()` is equal to `n2.hash()`.

```
x = Int('x')
print (x + 1).hash()
print (1 + x).hash()
print x.sort().hash()
```

Z3 expressions can be divided in three basic groups: **applications**, **quantifiers** and **bounded/free variables**. Applications are all you need if your problems do not contain universal/existential quantifiers. Although we say `Int('x')` is an integer "variable", it is technically an integer constant, and internally is represented as a function application with `0` arguments. Every application is associated with a **declaration** and contains `0` or more arguments. The method `decl()` returns the declaration associated with an application. The method `num_args()` returns the number of arguments of an application, and `arg(i)` one of the arguments. The function `is_expr(n)` returns `True` if `n` is an expression. Similarly `is_app(n)` (`is_func_decl(n)`) returns `True` if `n` is an application (declaration).

```
x = Int('x')
print "is expression: ", is_expr(x)
n = x + 1
print "is application:", is_app(n)
print "decl:          ", n.decl()
print "num args:      ", n.num_args()
for i in range(n.num_args()):
    print "arg(", i, ") ->", n.arg(i)
```

Declarations have names, they are retrieved using the method `name()`. A (function) declaration has an

arity, a domain and range sorts.

```
x   = Int('x')
x_d = x.decl()
print "is_expr(x_d):      ", is_expr(x_d)
print "is_func_decl(x_d):", is_func_decl(x_d)
print "x_d.name():        ", x_d.name()
print "x_d.range():       ", x_d.range()
print "x_d.arity():       ", x_d.arity()
# x_d() creates an application with 0 arguments using x_d.
print "eq(x_d(), x):      ", eq(x_d(), x)
print "\n"
# f is a function from (Int, Real) to Bool
f   = Function('f', IntSort(), RealSort(), BoolSort())
print "f.name():          ", f.name()
print "f.range():         ", f.range()
print "f.arity():         ", f.arity()
for i in range(f.arity()):
    print "domain(", i, "): ", f.domain(i)
# f(x, x) creates an application with 2 arguments using f.
print f(x, x)
print eq(f(x, x).decl(), f)
```

The built-in declarations are identified using their **kind**. The kind is retrieved using the method `kind()`. The complete list of built-in declarations can be found in the file `z3consts.py` (`z3_api.h`) in the Z3 distribution.

```
x, y = Ints('x y')
print (x + y).decl().kind() == Z3_OP_ADD
print (x + y).decl().kind() == Z3_OP_SUB
```

The following example demonstrates how to substitute sub-expressions in Z3 expressions.

```
x, y = Ints('x y')
f    = Function('f', IntSort(), IntSort(), IntSort())
g    = Function('g', IntSort(), IntSort())
n    = f(f(g(x), g(g(x))), g(g(y)))
print n
# substitute g(g(x)) with y and g(y) with x + 1
print substitute(n, (g(g(x)), y), (g(y), x + 1))
```

The function `Const(name, sort)` declares a constant (aka variable) of the given sort. For example, the functions `Int(name)` and `Real(name)` are shorthands for `Const(name, IntSort())` and `Const(name, RealSort())`.

```
x = Const('x', IntSort())
print eq(x, Int('x'))

a, b = Consts('a b', BoolSort())
print And(a, b)
```

Arrays

As part of formulating a programme of a mathematical theory of computation McCarthy proposed a *basic* theory of arrays as characterized by the select-store axioms. 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`. In Z3Py, we can also write `Select(a, i)` as `a[i]`.

```
# Use I as an alias for IntSort()
I = IntSort()
```

```
# A is an array from integer to integer
A = Array('A', I, I)
x = Int('x')
print A[x]
print Select(A, x)
print Store(A, x, 10)
print simplify(Select(Store(A, 2, x+1), 2))
```

By default, Z3 assumes that arrays are extensional over select. In other words, Z3 also enforces that if two arrays agree on all positions, then the arrays are equal.

Z3 also contains various extensions for operations on arrays that remain decidable and amenable to efficient saturation procedures (here efficient means, with an NP-complete satisfiability complexity). We describe these extensions in the following using a collection of examples. Additional background on these extensions is available in the paper Generalized and Efficient Array Decision Procedures.

Arrays in Z3 are used to model unbounded or very large arrays. Arrays should not be used to model small finite collections of values. It is usually much more efficient to create different variables using list comprehensions.

```
# We want an array with 3 elements.
# 1. Bad solution
X = Array('x', IntSort(), IntSort())
# Example using the array
print X[0] + X[1] + X[2] >=0

# 2. More efficient solution
X = IntVector('x', 3)
print X[0] + X[1] + X[2] >= 0
print Sum(X) >= 0
```

Select and Store

Let us first check a basic property of arrays. Suppose `A` is an array of integers, then the constraints `A[x] == x, Store(A, x, y) == A` are satisfiable for an array that contains an index `x` that maps to `x`, and when `x == y`. We can solve these constraints.

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

The interpretation/solution for array variables is very similar to the one used for functions.

The problem becomes unsatisfiable/infeasible if we add the constraint `x != y`.

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

Constant arrays

The array that maps all indices to some fixed value can be specified in Z3Py using the `K(s, v)` construct where `s` is a sort/type and `v` is an expression. `K(s, v)` returns a array that maps any value of `s` into `v`. The following example defines a constant array containing only ones.

```
AllOne = K(IntSort(), 1)
a, i = Ints('a i')
solve(a == AllOne[i])
# The following constraints do not have a solution
```

```
solve(a == AllOne[i], a != 1)
```

Datatypes

Algebraic datatypes, known from programming languages such as ML, offer a convenient way for specifying common data structures. Records and tuples are special cases of algebraic datatypes, and so are scalars (enumeration types). But algebraic datatypes are more general. They can be used to specify finite lists, trees and other recursive structures.

The following example demonstrates how to declare a List in Z3Py. It is more verbose than using the SMT 2.0 front-end, but much simpler than using the Z3 C API. It consists of two phases. First, we have to declare the new datatype, its constructors and accessors. The function `Datatype('List')` declares a "placeholder" that will contain the constructors and accessors declarations. The method `declare(cname, (aname, sort)+)` declares a constructor named `cname` with the given accessors. Each accessor has an associated `sort` or a reference to the datatypes being declared. For example, `declare('cons', ('car', IntSort()), ('cdr', List))` declares the constructor named `cons` that builds a new `List` using an integer and a `List`. It also declares the accessors `car` and `cdr`. The accessor `car` extracts the integer of a `cons` cell, and `cdr` the list of a `cons` cell. After all constructors were declared, we use the method `create()` to create the actual datatype in Z3. Z3Py makes the new Z3 declarations and constants available as slots of the new object.

```
# Declare a List of integers
List = Datatype('List')
# Constructor cons: (Int, List) -> List
List.declare('cons', ('car', IntSort()), ('cdr', List))
# Constructor nil: List
List.declare('nil')
# Create the datatype
List = List.create()
print is_sort(List)
cons = List.cons
car  = List.car
cdr  = List.cdr
nil  = List.nil
# cons, car and cdr are function declarations, and nil a constant
print is_func_decl(cons)
print is_expr(nil)

l1 = cons(10, cons(20, nil))
print l1
print simplify(cdr(l1))
print simplify(car(l1))
print simplify(l1 == nil)
```

The following example demonstrates how to define a Python function that given a sort creates a list of the given sort.

```
def DeclareList(sort):
    List = Datatype('List_of_%s' % sort.name())
    List.declare('cons', ('car', sort), ('cdr', List))
    List.declare('nil')
    return List.create()

IntList     = DeclareList(IntSort())
RealList    = DeclareList(RealSort())
IntListList = DeclareList(IntList)

l1 = IntList.cons(10, IntList.nil)
print l1
print IntListList.cons(l1, IntListList.cons(l1, IntListList.nil))
```

```
print RealList.cons("1/3", RealList.nil)

print l1.sort()
```

The example above demonstrates that Z3 supports operator overloading. There are several functions named `cons`, but they are different since they receive and/or return values of different sorts. Note that it is not necessary to use a different sort name for each instance of the sort list. That is, the expression `'List_of_%s' % sort.name()` is not necessary, we use it just to provide more meaningful names.

As described above enumeration types are a special case of algebraic datatypes. The following example declares an enumeration type consisting of three values: `red`, `green` and `blue`.

```
Color = Datatype('Color')
Color.declare('red')
Color.declare('green')
Color.declare('blue')
Color = Color.create()

print is_expr(Color.green)
print Color.green == Color.blue
print simplify(Color.green == Color.blue)

# Let c be a constant of sort Color
c = Const('c', Color)
# Then, c must be red, green or blue
prove(Or(c == Color.green,
         c == Color.blue,
         c == Color.red))
```

Z3Py also provides the following shorthand for declaring enumeration sorts.

```
Color, (red, green, blue) = EnumSort('Color', ('red', 'green', 'blue'))

print green == blue
print simplify(green == blue)

c = Const('c', Color)
solve(c != green, c != blue)
```

Mutually recursive datatypes can also be declared. The only difference is that we use the function `CreateDatatypes` instead of the method `create()` to create the mutually recursive datatypes.

```
TreeList = Datatype('TreeList')
Tree     = Datatype('Tree')
Tree.declare('leaf', ('val', IntSort()))
Tree.declare('node', ('left', TreeList), ('right', TreeList))
TreeList.declare('nil')
TreeList.declare('cons', ('car', Tree), ('cdr', TreeList))

Tree, TreeList = CreateDatatypes(Tree, TreeList)

t1  = Tree.leaf(10)
tl1 = TreeList.cons(t1, TreeList.nil)
t2  = Tree.node(tl1, TreeList.nil)
print t2
print simplify(Tree.val(t1))

t1, t2, t3 = Consts('t1 t2 t3', TreeList)

solve(Distinct(t1, t2, t3))
```

Uninterpreted Sorts

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 arithmetic operators such as `+` and `-` that have a fixed standard interpretation. 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 an (uninterpreted) sort `A`, and the constants `x`, `y` ranging over `A`. Finally let `f` be an uninterpreted function that takes one argument of sort `A` and results in a value of sort `A`. 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`.

```
A    = DeclareSort('A')
x, y = Consts('x y', A)
f    = Function('f', A, A)

s    = Solver()
s.add(f(f(x)) == x, f(x) == y, x != y)

print s.check()
m = s.model()
print m
print "interpretation assigned to A:"
print m[A]
```

The resulting model introduces abstract values for the elements in `A`, because the sort `A` is uninterpreted. The interpretation for `f` in the model toggles between the two values for `x` and `y`, which are different. The expression `m[A]` returns the interpretation (universe) for the uninterpreted sort `A` in the model `m`.

Quantifiers

Z3 is can solve quantifier-free problems containing arithmetic, bit-vector, Booleans, arrays, functions and datatypes. Z3 also accepts and can work with formulas that use quantifiers. It is no longer a decision procedure for such formulas in general (and for good reasons, as there can be no decision procedure for first-order logic).

```
f = Function('f', IntSort(), IntSort(), IntSort())
x, y = Ints('x y')
print ForAll([x, y], f(x, y) == 0)
print Exists(x, f(x, x) >= 0)

a, b = Ints('a b')
solve(ForAll(x, f(x, x) == 0), f(a, b) == 1)
```

Nevertheless, Z3 is often able to handle formulas involving quantifiers. It uses several approaches to handle quantifiers. The most prolific approach is using *pattern-based* quantifier instantiation. This approach allows instantiating quantified formulas with ground terms that appear in the current search context based on *pattern annotations* on quantifiers. Z3 also contains a model-based quantifier instantiation component that uses a model construction to find good terms to instantiate quantifiers with; and Z3 also handles many decidable fragments.

Note that in the previous example the constants `x` and `y` were used to create quantified formulas. This is a "trick" for simplifying the construction of quantified formulas in Z3Py. Internally, these constants are replaced by bounded variables. The next example demonstrates that. The method `body()` retrives the quantified expression. In the resultant formula the bounded variables are free. The function

`Var(index, sort)` creates a bounded/free variable with the given index and sort.

```
f = Function('f', IntSort(), IntSort(), IntSort())
x, y = Ints('x y')
f = ForAll([x, y], f(x, y) == 0)
print f.body()
v1 = f.body().arg(0).arg(0)
print v1
print eq(v1, Var(1, IntSort()))
```

Modeling with Quantifiers

Suppose we want to model an object oriented type system with single inheritance. We would need a predicate for sub-typing. Sub-typing should be a partial order, and respect single inheritance. For some built-in type constructors, such as for `array_of`, sub-typing should be monotone.

```
Type     = DeclareSort('Type')
subtype  = Function('subtype', Type, Type, BoolSort())
array_of = Function('array_of', Type, Type)
root     = Const('root', Type)

x, y, z  = Consts('x y z', Type)

axioms = [ ForAll(x, subtype(x, x)),
           ForAll([x, y, z], Implies(And(subtype(x, y), subtype(y, z)),
                                     subtype(x, z))),
           ForAll([x, y], Implies(And(subtype(x, y), subtype(y, x)),
                                  x == y)),
           ForAll([x, y, z], Implies(And(subtype(x, y), subtype(x, z)),
                                     Or(subtype(y, z), subtype(z, y)))),
           ForAll([x, y], Implies(subtype(x, y),
                                  subtype(array_of(x), array_of(y)))),

           ForAll(x, subtype(root, x))
           ]
s = Solver()
s.add(axioms)
print s
print s.check()
print "Interpretation for Type:"
print s.model()[Type]
print "Model:"
print s.model()
```

Patterns

The Stanford Pascal verifier and the subsequent Simplify theorem prover pioneered the use of pattern-based quantifier instantiation. The basic idea behind pattern-based quantifier instantiation is in a sense straight-forward: Annotate a quantified formula using a pattern that contains all the bound variables. So a pattern is an expression (that does not contain binding operations, such as quantifiers) that contains variables bound by a quantifier. Then instantiate the quantifier whenever a term that matches the pattern is created during search. This is a conceptually easy starting point, but there are several subtleties that are important.

In the following example, the first two options make sure that Model-based quantifier instantiation engine is disabled. We also annotate the quantified formula with the pattern `f(g(x))`. Since there is no ground instance of this pattern, the quantifier is not instantiated, and Z3 fails to show that the formula is unsatisfiable.

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

```
g = Function('g', IntSort(), IntSort())
a, b, c = Ints('a b c')
x = Int('x')

s = Solver()
s.set(auto_config=False, mbqi=False)

s.add( ForAll(x, f(g(x)) == x, patterns = [f(g(x))]),
       g(a) == c,
       g(b) == c,
       a != b )

# Display solver state using internal format
print s.sexpr()
print s.check()
```

When the more permissive pattern `g(x)` is used. Z3 proves the formula to be unsatisfiable. More restrive patterns minimize the number of instantiations (and potentially improve performance), but they may also make Z3 "less complete".

```
f = Function('f', IntSort(), IntSort())
g = Function('g', IntSort(), IntSort())
a, b, c = Ints('a b c')
x = Int('x')

s = Solver()
s.set(auto_config=False, mbqi=False)

s.add( ForAll(x, f(g(x)) == x, patterns = [g(x)]),
       g(a) == c,
       g(b) == c,
       a != b )

# Display solver state using internal format
print s.sexpr()
print s.check()
```

Some patterns may also create long instantiation chains. Consider the following assertion.

```
ForAll([x, y], Implies(subtype(x, y),
                       subtype(array_of(x), array_of(y))),
       patterns=[subtype(x, y)])
```

The axiom gets instantiated whenever there is some ground term of the form `subtype(s, t)`. The instantiation causes a fresh ground term `subtype(array_of(s), array_of(t))`, which enables a new instantiation. This undesirable situation is called a matching loop. Z3 uses many heuristics to break matching loops.

Before elaborating on the subtleties, we should address an important first question. What defines the terms that are created during search? In the context of most SMT solvers, and of the Simplify theorem prover, terms exist as part of the input formula, they are of course also created by instantiating quantifiers, but terms are also implicitly created when equalities are asserted. The last point means that terms are considered up to congruence and pattern matching takes place modulo ground equalities. We call the matching problem **E-matching**. For example, if we have the following equalities:

```
f = Function('f', IntSort(), IntSort())
g = Function('g', IntSort(), IntSort())
a, b, c = Ints('a b c')
x = Int('x')
```

```
s = Solver()
s.set(auto_config=False, mbqi=False)

s.add( ForAll(x, f(g(x)) == x, patterns = [f(g(x))]),
       a == g(b),
       b == c,
       f(a) != c )

print s.check()
```

The terms `f(a)` and `f(g(b))` are equal modulo the equalities. The pattern `f(g(x))` can be matched and `x` bound to `b` and the equality `f(g(b)) == b` is deduced.

While E-matching is an NP-complete problem, the main sources of overhead in larger verification problems comes from matching thousands of patterns in the context of an evolving set of terms and equalities. Z3 integrates an efficient E-matching engine using term indexing techniques.

Multi-patterns

In some cases, there is no pattern that contains all bound variables and does not contain interpreted symbols. In these cases, we use multi-patterns. In the following example, the quantified formula states that `f` is injective. This quantified formula is annotated with the multi-pattern `MultiPattern(f(x), f(y))`.

```
A = DeclareSort('A')
B = DeclareSort('B')
f = Function('f', A, B)
a1, a2 = Consts('a1 a2', A)
b      = Const('b', B)
x,  y  = Consts('x y', A)

s = Solver()
s.add(a1 != a2,
      f(a1) == b,
      f(a2) == b,
      ForAll([x, y], Implies(f(x) == f(y), x == y),
             patterns=[MultiPattern(f(x), f(y))])
      )
print s.check()
```

The quantified formula is instantiated for every pair of occurrences of `f`. A simple trick allows formulating injectivity of `f` in such a way that only a linear number of instantiations is required. The trick is to realize that `f` is injective if and only if it has a partial inverse.

```
A = DeclareSort('A')
B = DeclareSort('B')
f = Function('f', A, B)
finv = Function('finv', B, A)
a1, a2 = Consts('a1 a2', A)
b      = Const('b', B)
x,  y  = Consts('x y', A)

s = Solver()
s.add(a1 != a2,
      f(a1) == b,
      f(a2) == b,
      ForAll(x, finv(f(x)) == x)
      )
print s.check()
```

Other attributes

In Z3Py, the following additional attributes are supported: **qid** (quantifier identifier for debugging), **weight** (hint to the quantifier instantiation module: "more weight equals less instances"), **no_patterns** (expressions that should not be used as patterns, **skid** (identifier prefix used to create skolem constants/functions.

Multiple Solvers

In Z3Py and Z3 4.0 multiple solvers can be simultaneously used. It is also very easy to copy assertions/formulas from one solver to another.

```
x, y = Ints('x y')
s1 = Solver()
s1.add(x > 10, y > 10)
s2 = Solver()
# solver s2 is empty
print s2
# copy assertions from s1 to s2
s2.add(s1.assertions())
print s2
```

Unsat Cores and Soft Constraints

Z3Py also supports *unsat core extraction*. The basic idea is to use *assumptions*, that is, auxiliary propositional variables that we want to track. Assumptions are also available in the Z3 SMT 2.0 frontend, and in other Z3 front-ends. They are used to extract unsatisfiable cores. They may be also used to "retract" constraints. Note that, assumptions are not really *soft constraints*, but they can be used to implement them.

```
p1, p2, p3 = Bools('p1 p2 p3')
x, y = Ints('x y')
# We assert Implies(p, C) to track constraint C using p
s = Solver()
s.add(Implies(p1, x > 10),
      Implies(p1, y > x),
      Implies(p2, y < 5),
      Implies(p3, y > 0))
print s
# Check satisfiability assuming p1, p2, p3 are true
print s.check(p1, p2, p3)
print s.unsat_core()

# Try again retracting p2
print s.check(p1, p3)
print s.model()
```

The example above also shows that a Boolean variable (`p1`) can be used to track more than one constraint. Note that Z3 does not guarantee that the unsat cores are minimal.

Formatter

Z3Py uses a formatter (aka pretty printer) for displaying formulas, expressions, solvers, and other Z3 objects. The formatter supports many configuration options. The command `set_option(html_mode=False)` makes all formulas and expressions to be displayed in Z3Py notation.

```
x = Int('x')
y = Int('y')
print x**2 + y**2 >= 1
set_option(html_mode=False)
print x**2 + y**2 >= 1
```

By default, Z3Py will truncate the output if the object being displayed is too big. Z3Py uses … to denote the output is truncated. The following configuration options can be set to control the behavior of Z3Py's formatter:

- `max_depth` Maximal expression depth. Deep expressions are replaced with ….
- `max_args` Maximal number of arguments to display per node.
- `rational_to_decimal` Display rationals as decimals if True.
- `precision` Maximal number of decimal places for numbers being displayed in decimal notation.
- `max_lines` Maximal number of lines to be displayed.
- `max_width` Maximal line width (this is a suggestion to Z3Py).
- `max_indent` Maximal indentation.

```
x = IntVector('x', 20)
y = IntVector('y', 20)
f = And(Sum(x) >= 0, Sum(y) >= 0)

set_option(max_args=5)
print "\ntest 1:"
print f

print "\ntest 2:"
set_option(max_args=100, max_lines=10)
print f

print "\ntest 3:"
set_option(max_width=300)
print f
```