Strategies

High-performance solvers, such as Z3, contain many tightly integrated, handcrafted heuristic combinations of algorithmic proof methods. While these heuristic combinations tend to be highly tuned for known classes of problems, they may easily perform very badly on new classes of problems. This issue is becoming increasingly pressing as solvers begin to gain the attention of practitioners in diverse areas of science and engineering. In many cases, changes to the solver heuristics can make a tremendous difference.

In this tutorial we show how to create custom strategies using the basic building blocks available in Z3. Z3Py and Z3 4.0 implement the ideas proposed in this article.

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

Introduction

Z3 implements a methodology for orchestrating reasoning engines where "big" symbolic reasoning steps are represented as functions known as **tactics**, and tactics are composed using combinators known as **tacticals**. Tactics process sets of formulas called **Goals**.

When a tactic is applied to some goal `G`, four different outcomes are possible. The tactic succeeds in showing `G` to be satisfiable (i.e., feasible); succeeds in showing `G` to be unsatisfiable (i.e., infeasible); produces a sequence of subgoals; or fails. When reducing a goal `G` to a sequence of subgoals `G1`, ..., `Gn`, we face the problem of model conversion. A **model converter** construct a model for `G` using a model for some subgoal `Gi`.

In the following example, we create a goal `g` consisting of three formulas, and a tactic `t` composed of two built-in tactics: `simplify` and `solve-eqs`. The tactic `simplify` apply transformations equivalent to the ones found in the command `simplify`. The tactic `solver-eqs` eliminate variables using Gaussian elimination. Actually, `solve-eqs` is not restricted only to linear arithmetic. It can also eliminate arbitrary variables. Then, combinator `Then` applies `simplify` to the input goal and `solve-eqs` to each subgoal produced by `simplify`. In this example, only one subgoal is produced.

```
x, y = Reals('x y')
g  = Goal()
g.add(x > 0, y > 0, x == y + 2)
print g

t1 = Tactic('simplify')
t2 = Tactic('solve-eqs')
t  = Then(t1, t2)
print t(g)
```

In the example above, variable `x` is eliminated, and is not present the resultant goal.

In Z3, we say a **clause** is any constraint of the form `Or(f_1, ..., f_n)`. The tactic `split-clause` will select a clause `Or(f_1, ..., f_n)` in the input goal, and split it `n` subgoals. One for each subformula `f_i`.

```
x, y = Reals('x y')
g  = Goal()
g.add(Or(x < 0, x > 0), x == y + 1, y < 0)

t = Tactic('split-clause')
```

```
r = t(g)
for g in r:
    print g
```

Tactics

Z3 comes equipped with many built-in tactics. The command `describe_tactics()` provides a short description of all built-in tactics.

```
describe_tactics()
```

Z3Py comes equipped with the following tactic combinators (aka tacticals):

- `Then(t, s)` applies `t` to the input goal and `s` to every subgoal produced by `t`.
- `OrElse(t, s)` first applies `t` to the given goal, if it fails then returns the result of `s` applied to the given goal.
- `Repeat(t)` Keep applying the given tactic until no subgoal is modified by it.
- `Repeat(t, n)` Keep applying the given tactic until no subgoal is modified by it, or the number of iterations is greater than `n`.
- `TryFor(t, ms)` Apply tactic `t` to the input goal, if it does not return in `ms` millisenconds, it fails.
- `With(t, params)` Apply the given tactic using the given parameters.

The following example demonstrate how to use these combinators.

```
x, y, z = Reals('x y z')
g = Goal()
g.add(Or(x == 0, x == 1),
      Or(y == 0, y == 1),
      Or(z == 0, z == 1),
      x + y + z > 2)

# Split all clauses"
split_all = Repeat(OrElse(Tactic('split-clause'),
                          Tactic('skip')))
print split_all(g)

split_at_most_2 = Repeat(OrElse(Tactic('split-clause'),
                          Tactic('skip')),
                         1)
print split_at_most_2(g)

# Split all clauses and solve equations
split_solve = Then(Repeat(OrElse(Tactic('split-clause'),
                                 Tactic('skip'))),
                   Tactic('solve-eqs'))

print split_solve(g)
```

In the tactic `split_solver`, the tactic `solve-eqs` discharges all but one goal. Note that, this tactic generates one goal: the empty goal which is trivially satisfiable (i.e., feasible)

The list of subgoals can be easily traversed using the Python `for` statement.

```
x, y, z = Reals('x y z')
g = Goal()
g.add(Or(x == 0, x == 1),
      Or(y == 0, y == 1),
      Or(z == 0, z == 1),
      x + y + z > 2)
```

```
# Split all clauses"
split_all = Repeat(OrElse(Tactic('split-clause'),
                          Tactic('skip')))
for s in split_all(g):
    print s
```

A tactic can be converted into a solver object using the method `solver()`. If the tactic produces the empty goal, then the associated solver returns `sat`. If the tactic produces a single goal containing `False`, then the solver returns `unsat`. Otherwise, it returns `unknown`.

```
bv_solver = Then('simplify',
                 'solve-eqs',
                 'bit-blast',
                 'sat').solver()

x, y = BitVecs('x y', 16)
solve_using(bv_solver, x | y == 13, x > y)
```

In the example above, the tactic `bv_solver` implements a basic bit-vector solver using equation solving, bit-blasting, and a propositional SAT solver. Note that, the command `Tactic` is suppressed. All Z3Py combinators automatically invoke `Tactic` command if the argument is a string. Finally, the command `solve_using` is a variant of the `solve` command where the first argument specifies the solver to be used.

In the following example, we use the solver API directly instead of the command `solve_using`. We use the combinator `With` to configure our little solver. We also include the tactic `aig` which tries to compress Boolean formulas using And-Inverted Graphs.

```
bv_solver = Then(With('simplify', mul2concat=True),
                 'solve-eqs',
                 'bit-blast',
                 'aig',
                 'sat').solver()
x, y = BitVecs('x y', 16)
bv_solver.add(x*32 + y == 13, x & y < 10, y > -100)
print bv_solver.check()
m = bv_solver.model()
print m
print x*32 + y, "==", m.evaluate(x*32 + y)
print x & y, "==", m.evaluate(x & y)
```

The tactic `smt` wraps the main solver in Z3 as a tactic.

```
x, y = Ints('x y')
s = Tactic('smt').solver()
s.add(x > y + 1)
print s.check()
print s.model()
```

Now, we show how to implement a solver for integer arithmetic using SAT. The solver is complete only for problems where every variable has a lower and upper bound.

```
s = Then(With('simplify', arith_lhs=True, som=True),
         'normalize-bounds', 'lia2pb', 'pb2bv',
         'bit-blast', 'sat').solver()
x, y, z = Ints('x y z')
solve_using(s,
            x > 0, x < 10,
            y > 0, y < 10,
            z > 0, z < 10,
            3*y + 2*x == z)
```

```
# It fails on the next example (it is unbounded)
s.reset()
solve_using(s, 3*y + 2*x == z)
```

Tactics can be combined with solvers. For example, we can apply a tactic to a goal, produced a set of subgoals, then select one of the subgoals and solve it using a solver. The next example demonstrates how to do that, and how to use model converters to convert a model for a subgoal into a model for the original goal.

```
t = Then('simplify',
         'normalize-bounds',
         'solve-eqs')

x, y, z = Ints('x y z')
g = Goal()
g.add(x > 10, y == x + 3, z > y)

r = t(g)
# r contains only one subgoal
print r

s = Solver()
s.add(r[0])
print s.check()
# Model for the subgoal
print s.model()
# Model for the original goal
print r.convert_model(s.model())
```

Probes

Probes (aka formula measures) are evaluated over goals. Boolean expressions over them can be built using relational operators and Boolean connectives. The tactic `FailIf(cond)` fails if the given goal does not satisfy the condition `cond`. Many numeric and Boolean measures are available in Z3Py. The command `describe_probes()` provides the list of all built-in probes.

```
describe_probes()
```

In the following example, we build a simple tactic using `FailIf`. It also shows that a probe can be applied directly to a goal.

```
x, y, z = Reals('x y z')
g = Goal()
g.add(x + y + z > 0)

p = Probe('num-consts')
print "num-consts:", p(g)

t = FailIf(p > 2)
try:
    t(g)
except Z3Exception:
    print "tactic failed"

print "trying again..."
g = Goal()
g.add(x + y > 0)
print t(g)
```

Z3Py also provides the combinator (tactical) `If(p, t1, t2)` which is a shorthand for:

```
OrElse(Then(FailIf(Not(p)), t1), t2)
```

The combinator `When(p, t)` is a shorthand for:

```
If(p, t, 'skip')
```

The tactic `skip` just returns the input goal. The following example demonstrates how to use the `If` combinator.

```
x, y, z = Reals('x y z')
g = Goal()
g.add(x**2 - y**2 >= 0)

p = Probe('num-consts')
t = If(p > 2, 'simplify', 'factor')

print t(g)

g = Goal()
g.add(x + x + y + z >= 0, x**2 - y**2 >= 0)

print t(g)
```