Fixedpoints

This tutorial illustrates uses of Z3's fixedpoint engine. The following papers µZ - An Efficient Engine for Fixed-Points with Constraints. (CAV 2011) and Generalized Property Directed Reachability (SAT 2012) describe some of the main features of the engine.

Please send feedback, comments and/or corrections to nbjorner@microsoft.com.

Introduction

This tutorial covers some of the fixedpoint utilities available with Z3. The main features are a basic Datalog engine, an engine with relational algebra and an engine based on a generalization of the Property Directed Reachability algorithm.

Basic Datalog

The default fixed-point engine is a bottom-up Datalog engine. It works with finite relations and uses finite table representations as hash tables as the default way to represent finite relations.

Relations, rules and queries

The first example illustrates how to declare relations, rules and how to pose queries.

```
fp = Fixedpoint()

a, b, c = Bools('a b c')

fp.register_relation(a.decl(), b.decl(), c.decl())
fp.rule(a,b)
fp.rule(b,c)
fp.set(engine='datalog')

print "current set of rules\n", fp
print fp.query(a)

fp.fact(c)
print "updated set of rules\n", fp
print fp.query(a)
print fp.get_answer()
```

The example illustrates some of the basic constructs.

```
  fp = Fixedpoint()
```

creates a context for fixed-point computation.

```
 fp.register_relation(a.decl(), b.decl(), c.decl())
```

Register the relations `a, b, c` as recursively defined.

```
 fp.rule(a,b)
```

Create the rule that `a` follows from `b`. In general you can create a rule with multiple premises and a name using the format

```
 fp.rule(head,[body1,...,bodyN],name)
```

The *name* is optional. It is used for tracking the rule in derivation proofs. Continuing with the example, `a` is false unless `b` is established.

```
fp.query(a)
```

Asks if `a` can be derived. The rules so far say that `a` follows if `b` is established and that `b` follows if `c` is established. But nothing establishes `c` and `b` is also not established, so `a` cannot be derived.

```
fp.fact(c)
```

Add a fact (shorthand for `fp.rule(c,True)`). Now it is the case that `a` can be derived.

Explanations

It is also possible to get an explanation for a derived query. For the finite Datalog engine, an explanation is a trace that provides information of how a fact was derived. The explanation is an expression whose function symbols are Horn rules and facts used in the derivation.

```
fp = Fixedpoint()

a, b, c = Bools('a b c')

fp.register_relation(a.decl(), b.decl(), c.decl())
fp.rule(a,b)
fp.rule(b,c)
fp.fact(c)
fp.set(generate_explanations=True, engine='datalog')
print fp.query(a)
print fp.get_answer()
```

Relations with arguments

Relations can take arguments. We illustrate relations with arguments using edges and paths in a graph.

```
fp = Fixedpoint()
fp.set(engine='datalog')

s = BitVecSort(3)
edge = Function('edge', s, s, BoolSort())
path = Function('path', s, s, BoolSort())
a = Const('a',s)
b = Const('b',s)
c = Const('c',s)

fp.register_relation(path,edge)
fp.declare_var(a,b,c)
fp.rule(path(a,b), edge(a,b))
fp.rule(path(a,c), [edge(a,b),path(b,c)])

v1 = BitVecVal(1,s)
v2 = BitVecVal(2,s)
v3 = BitVecVal(3,s)
v4 = BitVecVal(4,s)

fp.fact(edge(v1,v2))
fp.fact(edge(v1,v3))
fp.fact(edge(v2,v4))

print "current set of rules", fp


print fp.query(path(v1,v4)), "yes we can reach v4 from v1"

print fp.query(path(v3,v4)), "no we cannot reach v4 from v3"
```

The example uses the declaration

```
fp.declare_var(a,b,c)
```

to instrument the fixed-point engine that `a, b, c` should be treated as variables when they appear in rules. Think of the convention as they way bound variables are passed to quantifiers in Z3Py.

Rush Hour

A more entertaining example of using the basic fixed point engine is to solve the Rush Hour puzzle. The puzzle is about moving a red car out of a gridlock. We have encoded a configuration and compiled a set of rules that encode the legal moves of the cars given the configuration. Other configurations can be tested by changing the parameters passed to the constructor for Car. We have encoded the configuration from an online puzzle you can solve manually, or cheat on by asking Z3.

```
class Car():
    def __init__(self, is_vertical, base_pos, length, start, color):
        self.is_vertical = is_vertical
        self.base = base_pos
        self.length = length
        self.start = start
        self.color = color

    def __eq__(self, other):
        return self.color == other.color

    def __ne__(self, other):
        return self.color != other.color

dimension = 6

red_car = Car(False, 2, 2, 3, "red")
cars = [
    Car(True, 0, 3, 0, 'yellow'),
    Car(False, 3, 3, 0, 'blue'),
    Car(False, 5, 2, 0, "brown"),
    Car(False, 0, 2, 1, "lgreen"),
    Car(True,  1, 2, 1, "light blue"),
    Car(True,  2, 2, 1, "pink"),
    Car(True,  2, 2, 4, "dark green"),
    red_car,
    Car(True,  3, 2, 3, "purple"),
    Car(False, 5, 2, 3, "light yellow"),
    Car(True,  4, 2, 0, "orange"),
    Car(False, 4, 2, 4, "black"),
    Car(True,  5, 3, 1, "light purple")
    ]

num_cars = len(cars)
B = BoolSort()
bv3 = BitVecSort(3)


state = Function('state', [ bv3 for c in cars] + [B])

def num(i):
    return BitVecVal(i,bv3)

def bound(i):
    return Const(cars[i].color, bv3)

fp = Fixedpoint()
fp.set(generate_explanations=True)
fp.declare_var([bound(i) for i in range(num_cars)])
fp.register_relation(state)

def mk_state(car, value):
    return state([ (num(value) if (cars[i] == car) else bound(i))
                   for i in range(num_cars)])
```

```
def mk_transition(row, col, i0, j, car0):
    body = [mk_state(car0, i0)]
    for index in range(num_cars):
        car = cars[index]
        if car0 != car:
            if car.is_vertical and car.base == col:
                for i in range(dimension):
                    if i <= row and row < i + car.length and i + car.length <= dimension:
                            body += [bound(index) != num(i)]
            if car.base == row and not car.is_vertical:
                for i in range(dimension):
                    if i <= col and col < i + car.length and i + car.length <= dimension:
                            body += [bound(index) != num(i)]

    s = "%s %d->%d" % (car0.color, i0, j)

    fp.rule(mk_state(car0, j), body, s)


def move_down(i, car):
    free_row = i + car.length
    if free_row < dimension:
        mk_transition(free_row, car.base, i, i + 1, car)


def move_up(i, car):
    free_row = i  - 1
    if 0 <= free_row and i + car.length <= dimension:
        mk_transition(free_row, car.base, i, i - 1, car)

def move_left(i, car):
    free_col = i - 1;
    if 0 <= free_col and i + car.length <= dimension:
        mk_transition(car.base, free_col, i, i - 1, car)


def move_right(i, car):
    free_col = car.length + i
    if free_col < dimension:
        mk_transition(car.base, free_col, i, i + 1, car)


# Initial state:
fp.fact(state([num(cars[i].start) for i in range(num_cars)]))

# Transitions:
for car in cars:
    for i in range(dimension):
        if car.is_vertical:
            move_down(i, car)
            move_up(i, car)
        else:
            move_left(i, car)
            move_right(i, car)


def get_instructions(ans):
    lastAnd = ans.arg(0).children()[-1]
    trace = lastAnd.children()[1]
    while trace.num_args() > 0:
        print trace.decl()
        trace = trace.children()[-1]

print fp

goal = state([ (num(4) if cars[i] == red_car else bound(i))
               for i in range(num_cars)])
```

```
fp.query(goal)

get_instructions(fp.get_answer())
```

Abstract Domains

The underlying engine uses table operations that are based on relational algebra. Representations are opaque to the underlying engine. Relational algebra operations are well defined for arbitrary relations. They don't depend on whether the relations denote a finite or an infinite set of values. Z3 contains two built-in tables for infinite domains. The first is for intervals of integers and reals. The second is for bound constraints between two integers or reals. A bound constraint is of the form `x or x . When used in conjunction, they form an abstract domain that is called the` *Pentagon* `abstract domain. Z3 implements` *reduced products* `of abstract domains that enables sharing constraints between the interval and bounds domains.`

Below we give a simple example that illustrates a loop at location `l0`. The loop is incremented as long as the loop counter does not exceed an upper bound. Using the combination of bound and interval domains we can collect derived invariants from the loop and we can also establish that the state after the loop does not exceed the bound.

```
I  = IntSort()
B  = BoolSort()
l0 = Function('l0',I,I,B)
l1 = Function('l1',I,I,B)

s = Fixedpoint()
s.set(engine='datalog',compile_with_widening=True,
      unbound_compressor=False)

s.register_relation(l0,l1)
s.set_predicate_representation(l0, 'interval_relation', 'bound_relation')
s.set_predicate_representation(l1, 'interval_relation', 'bound_relation')

m, x, y = Ints('m x y')

s.declare_var(m, x, y)
s.rule(l0(0,m),   0 < m)
s.rule(l0(x+1,m), [l0(x,m), x < m])
s.rule(l1(x,m),   [l0(x,m), m <= x])

print "At l0 we learn that x, y are non-negative:"
print s.query(l0(x,y))
print s.get_answer()

print "At l1 we learn that x <= y and both x and y are bigger than 0:"
print s.query(l1(x,y))
print s.get_answer()


print "The state where x < y is not reachable"
print s.query(And(l1(x,y), x < y))
```

The example uses the option

```
   set_option(dl_compile_with_widening=True)
```

to instrument Z3 to apply abstract interpretation *widening* on loop boundaries.

Engine for Property Directed Reachability

A different underlying engine for fixed-points is based on an algorithm for Property Directed Reachability

(PDR). The PDR engine is enabled using the instruction

```
set_option(dl_engine=1)
```

The version in Z3 applies to Horn clauses with arithmetic and Boolean domains. When using arithmetic you should enable the main abstraction engine used in Z3 for arithmetic in PDR.

```
set_option(dl_pdr_use_farkas=True)
```

The engine also works with domains using algebraic data-types and bit-vectors, although it is currently not overly tuned for either. The PDR engine is targeted at applications from symbolic model checking of software. The systems may be infinite state. The following examples also serve a purpose of showing how software model checking problems (of safety properties) can be embedded into Horn clauses and solved using PDR.

Procedure Calls

McCarthy's 91 function illustrates a procedure that calls itself recursively twice. The Horn clauses below encode the recursive function:

```
mc(x) = if x > 100 then x - 10 else mc(mc(x+11))
```

The general scheme for encoding recursive procedures is by creating a predicate for each procedure and adding an additional output variable to the predicate. Nested calls to procedures within a body can be encoded as a conjunction of relations.

```
mc = Function('mc', IntSort(), IntSort(), BoolSort())
n, m, p = Ints('n m p')

fp = Fixedpoint()

fp.declare_var(n,m)
fp.register_relation(mc)

fp.rule(mc(m, m-10), m > 100)
fp.rule(mc(m, n), [m <= 100, mc(m+11,p),mc(p,n)])

print fp.query(And(mc(m,n),n < 90))
print fp.get_answer()

print fp.query(And(mc(m,n),n < 91))
print fp.get_answer()

print fp.query(And(mc(m,n),n < 92))
print fp.get_answer()
```

The first two queries are unsatisfiable. The PDR engine produces the same proof of unsatisfiability. The proof is an inductive invariant for each recursive predicate. The PDR engine introduces a special query predicate for the query.

Bakery

We can also prove invariants of reactive systems. It is convenient to encode reactive systems as guarded transition systems. It is perhaps for some not as convenient to directly encode guarded transitions as recursive Horn clauses. But it is fairly easy to write a translator from guarded transition systems to recursive Horn clauses. We illustrate a translator and Lamport's two process Bakery algorithm in the next example.

```
set_option(relevancy=0,verbose=1)

def flatten(l):
    return [s for t in l for s in t]
```

```
class TransitionSystem():
    def __init__(self, initial, transitions, vars1):
        self.fp = Fixedpoint()
        self.initial     = initial
        self.transitions = transitions
        self.vars1 = vars1

    def declare_rels(self):
        B = BoolSort()
        var_sorts   = [ v.sort() for v in self.vars1 ]
        state_sorts = var_sorts
        self.state_vals = [ v for v in self.vars1 ]
        self.state_sorts  = state_sorts
        self.var_sorts = var_sorts
        self.state  = Function('state', state_sorts + [ B ])
        self.step   = Function('step',  state_sorts + state_sorts + [ B ])
        self.fp.register_relation(self.state)
        self.fp.register_relation(self.step)

# Set of reachable states are transitive closure of step.

    def state0(self):
        idx = range(len(self.state_sorts))
        return self.state([Var(i,self.state_sorts[i]) for i in idx])

    def state1(self):
        n = len(self.state_sorts)
        return self.state([Var(i+n, self.state_sorts[i]) for i in range(n)])

    def rho(self):
        n = len(self.state_sorts)
        args1 = [ Var(i,self.state_sorts[i]) for i in range(n) ]
        args2 = [ Var(i+n,self.state_sorts[i]) for i in range(n) ]
        args = args1 + args2
        return self.step(args)

    def declare_reachability(self):
        self.fp.rule(self.state1(), [self.state0(), self.rho()])


# Define transition relation

    def abstract(self, e):
        n = len(self.state_sorts)
        sub = [(self.state_vals[i], Var(i,self.state_sorts[i])) for i in range(n)]
        return substitute(e, sub)

    def declare_transition(self, tr):
        len_s  = len(self.state_sorts)
        effect = tr["effect"]
        vars1  = [Var(i,self.state_sorts[i]) for i in range(len_s)] + effect
        rho1   = self.abstract(self.step(vars1))
        guard = self.abstract(tr["guard"])
        self.fp.rule(rho1, guard)

    def declare_transitions(self):
        for t in self.transitions:
            self.declare_transition(t)

    def declare_initial(self):
        self.fp.rule(self.state0(),[self.abstract(self.initial)])

    def query(self, query):
        self.declare_rels()
        self.declare_initial()
        self.declare_reachability()
```

```
        self.declare_transitions()
        query = And(self.state0(), self.abstract(query))
        print self.fp
        print query
        print self.fp.query(query)
        print self.fp.get_answer()
#       print self.fp.statistics()


L = Datatype('L')
L.declare('L0')
L.declare('L1')
L.declare('L2')
L = L.create()
L0 = L.L0
L1 = L.L1
L2 = L.L2


y0 = Int('y0')
y1 = Int('y1')
l  = Const('l', L)
m  = Const('m', L)


t1 = { "guard" : l == L0,
       "effect" : [ L1, y1 + 1, m, y1 ] }
t2 = { "guard" : And(l == L1, Or([y0 <= y1, y1 == 0])),
       "effect" : [ L2, y0,     m, y1 ] }
t3 = { "guard" : l == L2,
       "effect" : [ L0, IntVal(0), m, y1 ]}
s1 = { "guard" : m == L0,
       "effect" : [ l,  y0, L1, y0 + 1 ] }
s2 = { "guard" : And(m == L1, Or([y1 <= y0, y0 == 0])),
       "effect" : [ l,  y0, L2, y1 ] }
s3 = { "guard" : m == L2,
       "effect" : [ l,  y0, L0, IntVal(0) ]}


ptr = TransitionSystem( And(l == L0, y0 == 0, m == L0, y1 == 0),
                        [t1, t2, t3, s1, s2, s3],
                        [l, y0, m, y1])

ptr.query(And([l == L2, m == L2 ]))
```

The rather verbose (and in no way minimal) inductive invariants are produced as answers.

Functional Programs

We can also verify some properties of functional programs using Z3's generalized PDR. Let us here consider an example from *Predicate Abstraction and CEGAR for Higher-Order Model Checking, Kobayashi et.al. PLDI 2011*. We encode functional programs by taking a suitable operational semantics and encoding an evaluator that is specialized to the program being verified (we don't encode a general purpose evaluator, you should partial evaluate it to help verification). We use algebraic data-types to encode the current closure that is being evaluated.

```
# let max max2 x y z = max2 (max2 x y) z
# let f x y = if x > y then x else y
# assert (f (max f x y z) x) = (max f x y z)


Expr = Datatype('Expr')
Expr.declare('Max')
Expr.declare('f')
Expr.declare('I', ('i', IntSort()))
Expr.declare('App', ('fn',Expr),('arg',Expr))
```

```
Expr = Expr.create()
Max  = Expr.Max
I    = Expr.I
App  = Expr.App
f    = Expr.f
Eval = Function('Eval',Expr,Expr,Expr,BoolSort())

x   = Const('x',Expr)
y   = Const('y',Expr)
z   = Const('z',Expr)
r1  = Const('r1',Expr)
r2  = Const('r2',Expr)
max = Const('max',Expr)
xi  = Const('xi',IntSort())
yi  = Const('yi',IntSort())

fp = Fixedpoint()
fp.register_relation(Eval)
fp.declare_var(x,y,z,r1,r2,max,xi,yi)

# Max max x y z = max (max x y) z
fp.rule(Eval(App(App(App(Max,max),x),y), z, r2),
        [Eval(App(max,x),y,r1),
         Eval(App(max,r1),z,r2)])

# f x y = x if x >= y
# f x y = y if x < y
fp.rule(Eval(App(f,I(xi)),I(yi),I(xi)),xi >= yi)
fp.rule(Eval(App(f,I(xi)),I(yi),I(yi)),xi < yi)

print fp.query(And(Eval(App(App(App(Max,f),x),y),z,r1),
                   Eval(App(f,x),r1,r2),
                   r1 != r2))

print fp.get_answer()
```