Skip to main content

Fixedpoints

This tutorial illustrates uses of Z3's fixedpoint engine. The following papers

μZ - An Efficient Engine for Fixed-Points with Constraints.(CAV 2011) andGeneralized Property Directed Reachability (SAT 2012) describe some of the main features of the engine.

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.

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

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.

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():
```

```
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

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,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()
```

Edit this page