ProofKit

F. Cabot, M. Ferrari, D. Morard

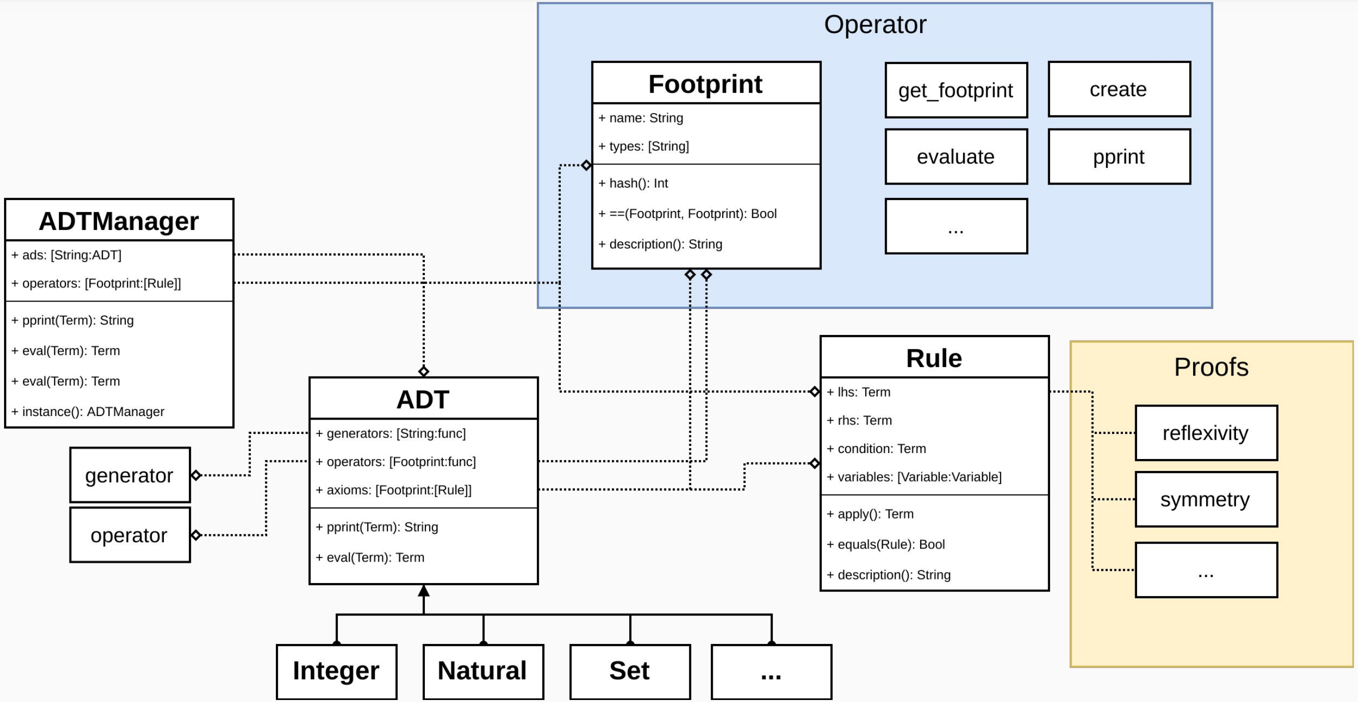
Summary

1. Introduction
2. ADTs
3. Proofs
4. Petri Nets
5. Conclusions

Introduction - Goals

1. Create ADTs
2. Create proofs evaluator
3. Petri net
4. User friendly Interface

Introduction - Structure



ADTs Table

ADT	Generators	Constructor	Operators
Boolean	`True()` `False()`	`n(Bool)`	`not(x)` `and(x,y)` `or(x,y)`
Nat	`zero()` `succ(x)`	`n(Int)`	`add(x,y)` `mul(x,y)` `pre(x)` `sub(x,y)` `div(x,y)` `mod(x)` `lt(x,y)` `gt(x,y)` `eq(x,y)` `gcd(x,y)`
Integer	`int(x,y)`	`n(Int)`	`add(x,y)` `mul(x,y)` `sub(x,y)` `div(x,y)` `abs(x),` `normalize(x)` `lt(x,y)` `gt(x,y)` `eq(x,y)` `sign(x)`
Multiset	`empty()` `cons(first, rest)`	`n([Term])`	`first(x)` `rest(x)` `contains(x,y)` `size(x)` `concat(x,y)` `removeOne(x,y)` `removeAll(x,y)` `eq(x,y)`
Set	`empty()` `cons(first, rest)`	`n([Term])`	`union(x,y)` `subSet(x,y)` `intersection(x,y)` `difference(x,y)` `contains(x,y)` `size(x)` `rest(x)` `first(x)` `removeOne(x,y)` `removeAll(x,y)` `eq(x,y)` `norm(x)` `insert(x,y)`
Sequence	`empty(),` `cons(value,index,rest)`	`n([Term])`	`push(value,rest),` `getAt(sequence, index),` `setAt(sequence, index, value)` `size(sequence)`

ADTs - Base types

1. Boolean
2. Natural
3. Integer

```
let a = Integer.n(-2)
let b = Integer.n(2)
let c = Integer.add(a,b)
res = ADTm.eval(c)
print("\(ADTm.pprint(c)) => \(ADTm.pprint(res))")
```

(-2 + +2) => 0

ADTs - Data structures

1. MultiSet

```
var k = Multiset.n([Nat.n(2),Nat.n(5),Nat.n(3), Nat.n(1),Nat.n(4)])
var exists = ADTm["multiset"]["contains"](k,Nat.n(1))
res = ADTm.eval(exists)//resolve(exists, contains)
print("\n \(ADTs.pprint(exists)) = \(ADTs.pprint(res))")
```

([2, 5, 3, 1, 4] contains 1) = true

2. Set

```
var s0 = Set.n([Nat.n(2),Nat.n(4),Nat.n(2),Nat.n(1)])
var s1 = Set.n([Nat.n(4),Nat.n(5),Nat.n(3),Nat.n(0),Nat.n(1)])
var s3 = Set.union(s0,s1)
res = ADTs.eval(s3)
print("\n \(ADTs.pprint(s3)) => \(ADTs.pprint(res))")
```

({1, 2, 4} union {1, 0, 3, 5, 4})
=> {1, 0, 3, 5, 4, 2}

ADTs - Data structures

3. Sequence

```
s0 = Sequence.n([Nat.n(2), Nat.n(3), Nat.n(4)])
s1 = Sequence.getAt(s0, Nat.n(1))
s3 = Sequence.setAt(s0, Nat.n(1), Nat.n(5))
print("\n s0: \(ADTm.pprint(s0))")
res = ADTm.eval(s1)
print(" \(ADTm.pprint(s1)) => \(ADTm.pprint(res))")
res = ADTm.eval(s3)
print(" \(ADTm.pprint(s3)) => \(ADTm.pprint(res))")
```

```
s0: [2: 4, 1: 3, 0: 2]

([2: 4, 1: 3, 0: 2] get 1) => 3

set([2: 4, 1: 3, 0: 2], 1, 5) => [2: 4, 1: 5, 0: 2]
```

Proofs

- reflexivity(Term) ⇒ Rule
- symmetry(Rule) ⇒ Rule
- transitivity(Rule,Rule) ⇒ Rule
- substitutivity((Term…)⇒Term, [Rule]) ⇒ Rule
- substitution(Rule, Variable, Term) ⇒ Rule
- inductive(Rule, Variable, ADT, [String:(Rule)⇒Rule]) ⇒ Rule

Proofs: induction

```
// proof for generator zero
func zero_proof(t: Rule...)->Rule{
  let ax0 = adtm["nat"].a("+")[0]
  // s(0)+0 = s(0)
  return Proof.substitution(ax0, Variable(named: "x"), Nat.succ(x: Nat.zero()))
}
// proof for generator succ(x)
func succ_proof(t: Rule...)->Rule{
  let ax1 = adtm["nat"].a("+")[1]
  // s(0) + s(y) = s(s(0) + y)
  let t2 = Proof.substitution(ax1, Variable(named: "x"), Nat.succ(x: Nat.zero()))
  // s(s(0) + x) = s(s(x))
  let t3 = Proof.substitutivity (Nat.succ, [t[0]])
  // s(0) + s(y) = s(s(y))
  return Proof.transitivity(t2, t3)
}
```

Proofs: induction

```
let conj = Rule(
  Nat.add(Nat.succ(x: Nat.zero()), Variable(named:"x")),
  Nat.succ(x: Variable(named: "x"))
)
do {
  let teo = try Proof.inductive(conj, Variable(named: "x"), ADTm["nat"], [
    "zero": zero_proof,
    "succ": succ_proof
  ])
  print("Inductive result: \(teo)")
}
catch ProofError.InductionFail {
  print("Induction failed!")
}
```

```
>> Inductive result: (1 + x) = succ(x)
```

LogicKit Integration

```
let x = Variable(named: "x")
let y = Variable(named: "y")

// x,y in Nat such that (x+y) < 9
let goal = x ∈ Nat.self &&  y ∈ Nat.self => (x + y) <-> Nat.n(6) && (x < y) <-> Boolean.True()
// < instead of ∈ can be used
```

```
>> x = 0, y = 0
>> x = 0, y = 1
>> x = 1, y = 0
>> x = 0, y = 2
>> x = 0, y = 3
...
```

Petri Nets Library

Petri Nets Library

Abstract Data Type

Tools

Adapted For Students

Petri Net Library : ADT

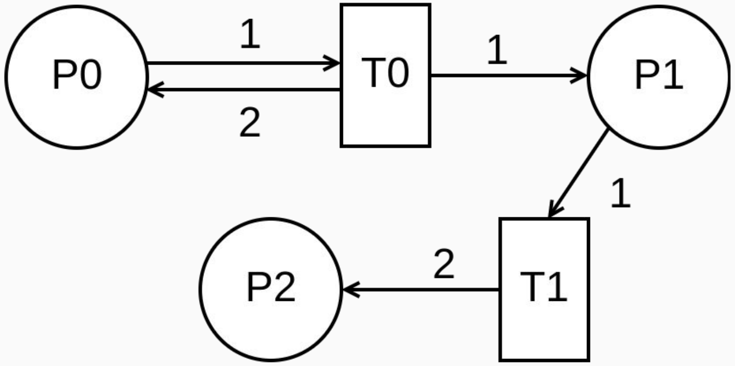
ADT	Generators	Operators
Marking	`null()` `next_place(p, m)`	`has_enough(m, w, p)`
Petrinet	`null()` `add_edge(p,t,w,net)`	`is_triggerable(net, t, m)`

Petri Net Library : ADT

```
let net = NicePetrinet(adtm:ADTm) +
          Place("P0") + Place("P1") + Place("P2") +
          Transition("T0") + Transition("T1") +
          InputEdge(p:"P0", t:"T0", weight:1) +
          InputEdge(p:"P1", t:"T0", weight:1) +
          InputEdge(p:"P1", t:"T1", weight:1) +
          OutputEdge(t:"T0", p:"P0", weight:2) +
          OutputEdge(t:"T1", p:"P2", weight:2)

print(net.to_string())
```

```
Places :
P2, P1, P0,
Transitions :
T0, T1,
Edges :
out(p:P2, t:T1, w:2)
out(p:P0, t:T0, w:2)
int(p:P1, t:T1, w:1)
int(p:P1, t:T0, w:1)
int(p:P0, t:T0, w:1)
```



Petri Net Library : ADT

```
let mark = NiceMarking(onPetrinet:net)
 mark["P0"] = 0
 mark["P1"] = 1
 mark["P2"] = 0

print(mark.to_string())
```

```
0, 1, 0,
```

Petri Net Library : ADT

```
print(mark.has_enough(weight:1, p:"P1"))

print(net.is_triggerable(t:"T1", marking:mark))
```

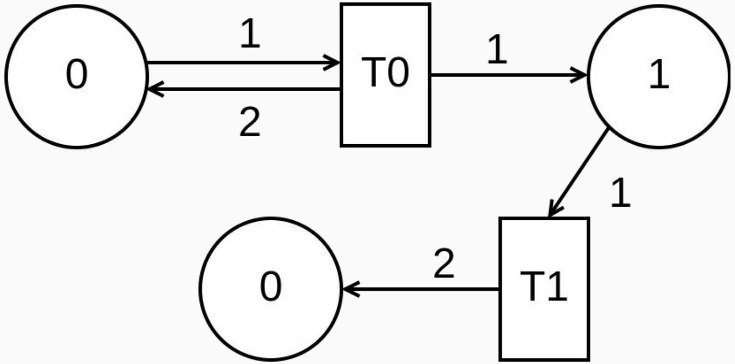
```
true
true
```

Petri Net Library : ADT

```
let v = Variable(named:"t")
let g = v < Nat.self && Petrinet.is_triggerable(net.as_term(), v, mark.as_term()) <-> Boolean.True()
for s in solve(g).prefix(2) {
  print(ADTm.pprint(s.reified()[v]))
}
```

```
1
2
```

Transition 0 is not triggerable !
But transition 2 doesn't exist
-> Efforts to be done on pretty printing



Petri Net Library : Tools

P-invariants and T-invariants checking

Minimal P-invariants and T-invariants computing (Farkas algorithm)

Petri Net Library : Tools

```
let dynMat = DynamicMatrix(
           [
             [-1, 1, 1, -1],
             [1, -1, -1, 1],
             [0, 0, 1, 0],
             [1, -1, 1, -1],
             [-1, 1, -1, 1]
           ])

 print(dynMat.get_p_invariants())
 print(dynMat.get_t_invariants())


 //to obtain the NicePetrinet’s matrix
 net.incidence_matrix()
```

```
[[1, 1, 0, 0, 0], [0, 0, 0, 1, 1], [1, 1, 0, 1, 1]]
[[1, 1, 0, 0]]
```

Conclusions and Demo

- Modular project
- User friendly syntax
- LogicKit integration