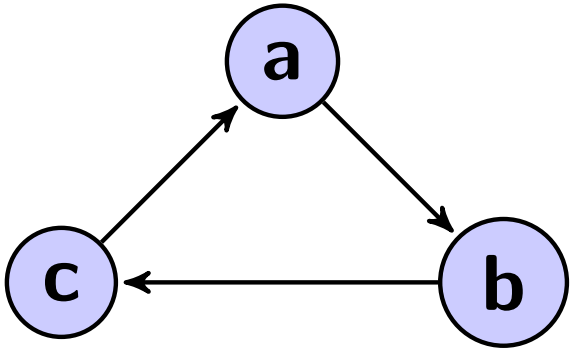
Lightweight Functional Logic Meta-Programming
in Scala

Nada Amin, Tiark Rompf

LAMP, EPFL

March 4, 2014

Logic Programming (Prolog)




```
edge(a,b).
edge(b,c).
edge(c,a).

path(X, Y) :- edge(X, Y).
path(X, Z) :- edge(X, Y), path(Y, Z).
```

```
?- edge(a, X).
X = b.

?- edge(X, a).
X = c.

?- path(a, X).
X = b ;
X = c ;
X = a ;
X = b .
```

Logic Programming (Scala)

```
def edge(x: Exp[String], y: Exp[String]): Rel =
  (x === "a") && (y === "b") ||
  (x === "b") && (y === "c") ||
  (x === "c") && (y === "a")

def path[T](x: Exp[T], y: Exp[T]): Rel =
  edge(x,y) ||
  exists[T] { z => edge(x,z) && path(y,b) }

// Queries
run[(String,String)] { case Pair(x,y) => edge(x,y) }
//=> pair(a,b), pair(b,c), pair(c,a)

runN[String](10) { q => path("a",q) }
//=> b, c, a, b, c, a, b, c, a, b
```

Deep Linguistic Reuse: OO Encapsulation

```
trait Graph[T] {
  def edge(x: Exp[T], y: Exp[T]): Rel
  def path(x: Exp[T], y: Exp[T]): Rel =
    edge(x,y) ||
    exists[T] { z => edge(x,z) && path(y,b) }
}
val g = new Graph[String] {
  def edge(x:Exp[String],y:Exp[String]) =
    (x === "a") && (y === "b") ||
    (x === "b") && (y === "c") ||
    (x === "c") && (y === "a")
}
```

Deep Linguistic Reuse: Type Classes

```
trait Ord[T] { def lt(x:Exp[T],y:Exp[T]): Rel }

implicit class OrdOps[T:Ord](x:Exp[T]) {
  def <(y:Exp[T]): Rel =
    implicitly[Ord[T]].lt(x,y)
}

implicit val ordNat = new Ord[Int] {
  def lt(x:Exp[Int],y:Exp[Int]): Rel =
    // ... elided
}


run[Int] { q => q < 4 } //=> 0,1,2,3
```

Type Classes: Lexicographic Ordering on Polymorphic Lists

```
implicit def ordList[T:Ord] = new Ord[List[T]] {
  def lt(as:Exp[List[T]],bs:Exp[List[T]]): Rel =
    exists[T,List[T]] { (b,bs1) =>
      (bs === cons(b,bs1)) && {
      (as === nil) || exists[T,List[T]] { (a,as1) =>
      (as === cons(a,as1)) && {
      (a < b) || (a === b) && (as1 < bs1) }}}}

run[List[Int]] { q => q < List(0,1,2) }
//=> nil
//   cons(z,nil)
//   cons(z,cons(z,x0))
//   cons(z,cons(s(z),nil)),
//   cons(z,cons(s(z),cons(z,x0)))
//   cons(z,cons(s(z),cons(s(z),x0)))
```

Type Classes: Run Queries Forwards and Backwards

```
run[String] { q =>
  val t = tree(List(1,1,1) -> "a",
               List(1,2,2) -> "b",
               List(2,1,1) -> "c")
  lookup(t,List(1,2,2),q)
} //=> b
run[Int] { q =>
  val t = tree(List(1,1,1) -> "a",
               cons(q,List(2,2)) -> "b",
               List(2,2,2) -> "c")
  lookup(t,List(1,2,2),"b")
} //=> 1
```

Vanilla Meta-Interpreter (Prolog)

```
/*
  A meta-interpreter for pure Prolog (Art of Prolog, 17.5)

  solve(Goal) :-
     Goal is true given the pure Prolog program defined by
     clause/2.
*/

     solve(true).
     solve((A,B)) :- solve(A), solve(B).
     solve(A) :- clause(A,B), solve(B).
```

Tracing Meta-Interpreter (Prolog)

```
/*
  A tracer for Prolog (Art of Prolog 17.7)
*/

solve_trace(Goal) :-
solve_trace(Goal,0).
     solve_trace(true,Depth) :- !.
     solve_trace((A,B),Depth) :-  !,
        solve_trace(A,Depth), solve_trace(B,Depth).
     solve_trace(A,Depth) :-
builtin(A), !, A, display(A,Depth), nl.
     solve_trace(A,Depth) :-
        clause(A,B),
display(A,Depth), nl,
Depth1 is Depth + 1,
solve_trace(B,Depth1).
```

Tracing (Scala)

```
val globalTrace = DVar(nil: Exp[List[List[String]]])
def path(x: Exp[T], y: Exp[T]): Rel = {
  globalTrace := cons(term("path",List(x,y)), globalTrace())
  edge(x,y) || exists[T] { z => edge(x,z) && path(y,b) }
}
```

Modular Tracing (Scala)

```
trait TracingGraph[T] extends Graph[T] {
  override def path(x:Exp[T],y:Exp[T]) =
    rule("path")(super.path)(x,y)
}
// tracing logic
val globalTrace = DVar(nil: Exp[List[List[String]]])
def rule[T,U](s: String)(f: (Exp[T],Exp[U]) => Rel):
  (Exp[T],Exp[U]) => Rel =
  { (a,b) =>
    globalTrace := cons(term(s,List(a,b)), globalTrace())
    f(a,b)
  }

runN[(String,List[String])](5) { case Pair(q1,q2) =>
      g.path("a",q1) && globalTrace() === q2 }
//=> pair(b,cons(path(a,b),nil)),
//   pair(c,cons(path(b,c),cons(path(a,c),nil))), ...
```

Probabilistic Logic Computation

```
val theprob = DVar(1.0)
def flip(p: Double)(a: => Rel)(b: => Rel): Rel =
  { theprob := theprob() * p; a } ||
  { theprob := theprob() * (1.0 - p); b }

runN[(Boolean,Double)](3) { case Pair(c,p) =>
  flip(0.2,c) && { p === theprob() } }
//=> pair(true,0.2), pair(false,0.8)
```

Clause Reification as Controlled Side Effect (Ex.)

```
var allclauses = Map[String,Clause]()
def reifyClause(goal: => Rel)(
    head: Exp[Goal], body: Exp[List[Goal]]): Rel =
  reifyGoals(goal)(cons(head,nil)) &&
  allclauses(extractKey(head))(head,body)
run[List[Any]] { q =>
  exists[Goal,List[Goal]] { (head,body) =>
    q === cons("to prove", cons(head,
          cons("prove", cons(body, nil)))) &&
    reifyClause(path(g)(fresh,fresh))(head,body)
  }
}
// cons(to prove,cons(path(a,b),cons(prove,cons(nil,nil)))),
// ... 2 more
// cons(to prove,cons(path(a,x0),
//   cons(prove,cons(cons(path(b,x0),nil),nil)))),
// ... 2 more
```

Clause Reification as Controlled Side Effect (Impl.)

```
val moregoals = DVar(fresh[List[Goal]])
def reifyGoals(goal: => Rel)(goals: Exp[List[Goal]]): Rel = {
  moregoals := goals
  goal && moregoals() === nil
}
def reflectGoal(goal: Exp[Goal]): Rel = {
  val hole = moregoals()
  moregoals := fresh
  hole === cons(goal,moregoals())
}
// reifyGoals(reflectGoal("path(a,b)")
//=>  "cons(path(a,b),nil)"

def rule[A,B](s: String)(f:(Exp[A], Exp[B]) => Rel) = {
  // ...
}
```

Tabling: Semantics of Memoization

Excerpt from Memoing for Logic Programs by David S. Warren:

Intuitively, we think of a machine that is carrying out a nondeterministic procedure as duplicating itself at a point of choice, and as disappearing when it encounters failure. Thus at any time, we have a set of deterministic machines computing away. The set gets larger when any one has to make a nondeterministic choice, and it gets smaller when any one fails. To add memoing, we imagine a single global table containing every procedure call that has been made by any machine, and for each such call, the answers that have been returned for it. Since the situation is nondeterministic, there may be none, one, or many answers for any single call. Now each machine, before it makes a procedure call, looks in the global table to see if the call has already been made. If not, it adds the call to the table and continues computing. During its computation, whenever a machine returns from a procedure, it finds the associated call in the global table, adds the answer it has just computed, and continues computing. (If the answer is already in the table, then this answer is a duplicate, and the machine fails.) When a procedure is about to be called, if the call is found to be already in the table, then for each associated answer in the table, the machine must fork off a new copy of itself to continue the computation with that answer. It is possible that not all the answers are in the table at this time; some may still be in the process of being computed by other machines and will show up later. Thus when a machine encounters a call already in the table, it forks off copies of itself to continue with the answers that are there, and it remains suspended on that table entry. Then whenever a new answer gets added to the table, the suspended machine makes a duplicate of itself to continue computing with that new answer. When a machine finishes its computation successfully, it disappears. The entire computation is complete when (and if) no machines are computing.

Conclusion

- like in Lightweight Modular Staging
  - exploit deep linguistic reuse
  - use virtualization instead of reflection for meta-programming
- further reading
  - dynamic variables local to search thread

    `http://okmij.org/ftp/Computation/dynamic-binding.html`
  - tutorial on purely embedding logic programming in functional host accessible to undergraduate students

    `http://webyrd.net/scheme-2013/papers/HemannMuKanren2013.pdf`

The Future: Embedding Logical Frameworks in Scala

- Possible improvements over Twelf
  - custom search strategies
  - meta-level abstraction: e.g. parametric lists
  - do better than copy'n'paste modularity
  - notion of functions: once you proved a relation is a function, use it as a function
  - z3 for totality / coverage checking
  - z3 as logic/constraint solving engine