Introduction to Logic Programming

Ambrose Bonnaire-Sergeant
@ambrosebs
abonnairesergeant@gmail.com

Introduction to Logic Programming

- Fundamental Logic Programming concepts
  - Related to FP
- General implementation characteristics of LP languages
- Gain an understanding of the execution model of core.logic

Pure Functions

- Pure **functions** (in Functional Programming)
  - Functions always have one value
    - Deterministic
  - Works for only one pattern of input and output arguments
- Sometimes functions are inappropriate
  - eg. 4 has two square roots, +2 and -2
    - 2 results
  - eg. Dividing a number by zero yields no result
    - 0 results

Relations

- We generalize functions to get **relations**
  - Any number of results (zero or more)
    - Non-deterministic
  - Pattern of inputs and output arguments can be different for each call

Relations

- In mathematics, the expression 'X r Y' is true if X and Y satisfy the relation 'r'
  - eg. 'X < Y', 4 ways the '<' relation can be considered
    - A **generator** of the (infinite) set of all (X,Y) pairs for which $X<Y$
    - A **predicate** that can be applied to (X,Y) pairs
    - A **generator**, that given X, will yield all Y values greater than X
    - A **generator**, that given Y, will yield all X values less than Y

Modified from *LIBRA: A Lazy Interpreter of Binary Relational Algebra (1995), Dwyer*

Converting a Function to a Relation

- Relations return true if the **relation is true**, and false if the **relation is false**
- To convert a function to a relation
  1) Convert the return value to an argument

```
(cons 1 [2])
;=> [1 2]

(cons° 1 [2] [1 2])
;=> true
```

cons^{o}

- We can use cons^{0} as a **predicate** if all arguments are **ground values** (not variables)
- For **(cons^{0} head tail result)**, conso returns true if *head* consed onto *tail* equals *result*

```
(consᵒ 1 [2] [1 2])
;=> true

(consᵒ 1 [] [1 2])
;=> false
```

cons^o

- We can use cons^0 as a **generator** if one argument is a variable
- **solve** introduces a logic variable **x** and returns a list of all values of **x** that satisfy the relation
  - Caps number of results with integer argument

```
(solve 1 [x]
  (cons° 1 [2] x))
;=> ([1 2])
```

```
(solve 1 [x]
  (cons° 1 x [1 2]))
;=> ([2])
```

$sqrt^o$

- A relation that can generate **multiple results**

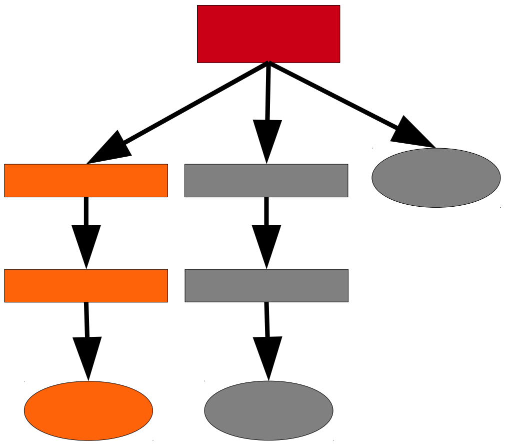
```
(solve 2 [x]
  (sqrt° 4 x))
;=> (2 -2)
```

Logic Language Implementation

- Logic Languages usually calculate zero or more results
  - Non-deterministic
- Execution strategy must be flexible
  - Implemented as a **search**

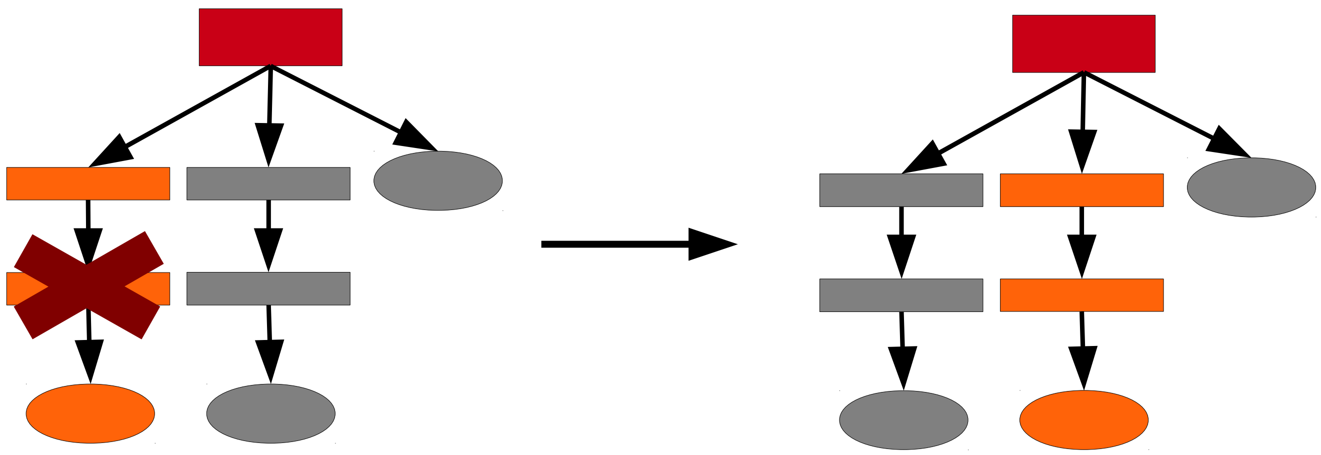
Execution Strategy - Branches

- A **choice point** groups together a set of **alternative statements**
  - If visualized as a tree, they are the branching nodes
- Executing a choice point picks an alternative statement and follows it
- If an alternative is found to be wrong later on, then another one is picked



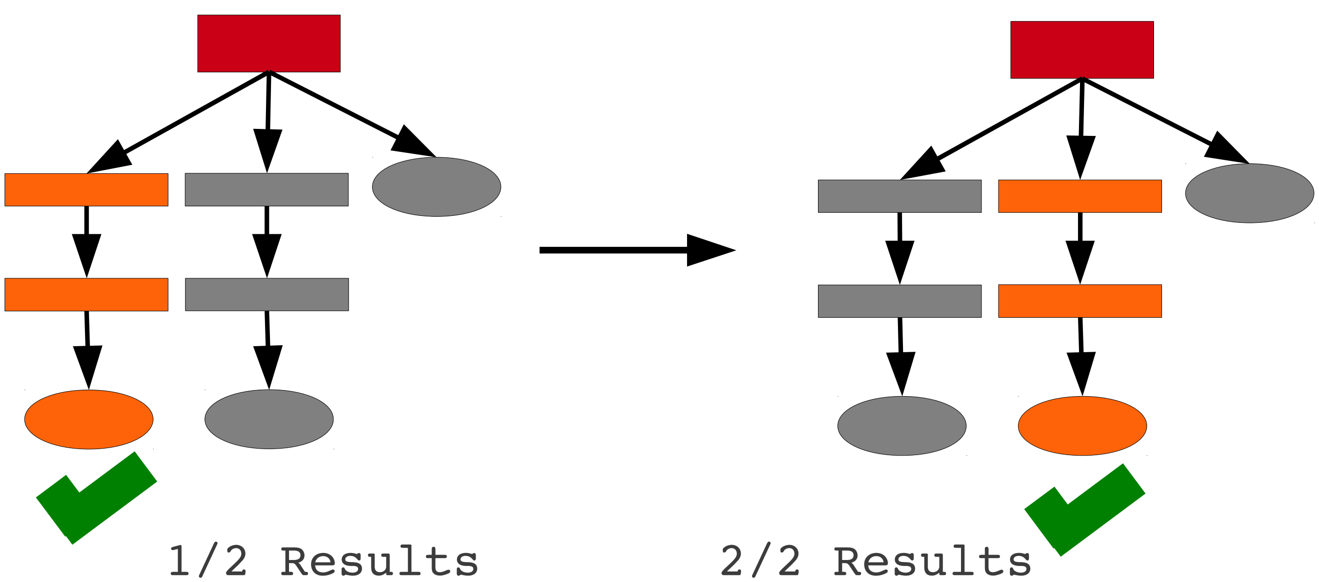
Execution Strategy - Failure

- A node fails if it consists of a **fail** statement that indicates the current alternative is wrong
  - This indicates we backtrack to a **choice point** and try another alternative



Execution Strategy – Leaf Nodes

- A leaf node represents one valid result
  - Contributes to our non-deterministic result
- If another result is requested, we backtrack to a **choice point** and execute another alternative statement



Encapsulated Search

- Relational programs can potentially execute in many different ways. We want to **control** which choices are made, and when they are made
  - Search strategy: depth-first search, breadth-first search, some other strategy
  - Specify the number of results
- One approach is to execute the relational program with **encapsulated search** inside a kind of **environment** which controls which choices are made and when they are made
  - Also **protects** the rest of the environment from (side) effects of the choices

Functional Approach

- **Protects** from the effects of choices by representing state by **substitutions**
  - Like a list of identity-value pairs for logic variables
- **Goals** are the “next state” functions
  - Functions of (Substitution → LazyList Substitution)
  - Relations implemented as goals
- **Controls** which choices are made by different monadic strategies, best visualized by search trees
  - Depth-first search, interleaving search
- **Controls** number of results by directive from programmer

Introducing core.logic

core.logic

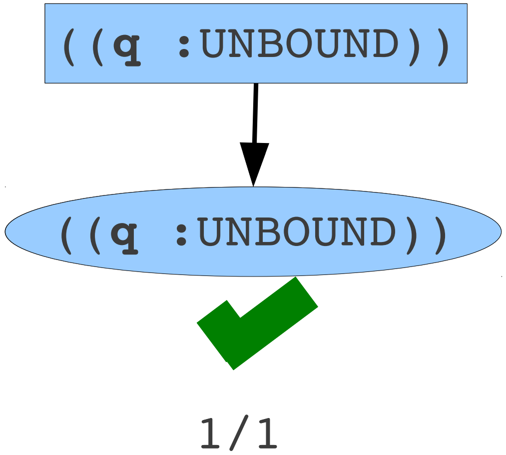
- Non-deterministic
- Substitutions
- Goals
- Queries via **run**
- Unbound logic variable represented by _.0, _.1 … _.n

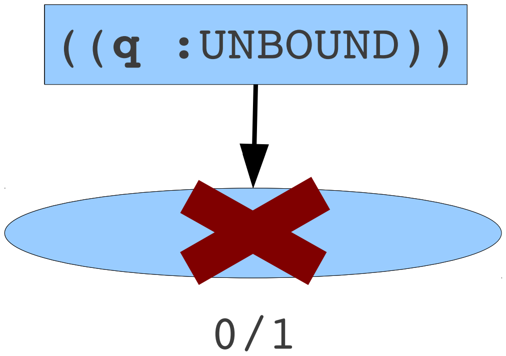
Fundamental Goals

- **succeed** is a no-op
- **fail** indicates that the current branch is wrong

```
(run 1 [q]
  succeed)
;=> (_.0)
```

```
(run 1 [q]
  fail)
;=> ()
```

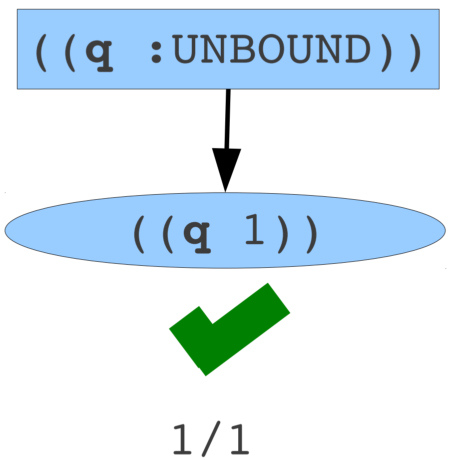




Unification

- Unification answers the question "what must the world look like for the left and right arguments to be equal?"
- eg. What must the world look like for 1 and **q** to be equal?

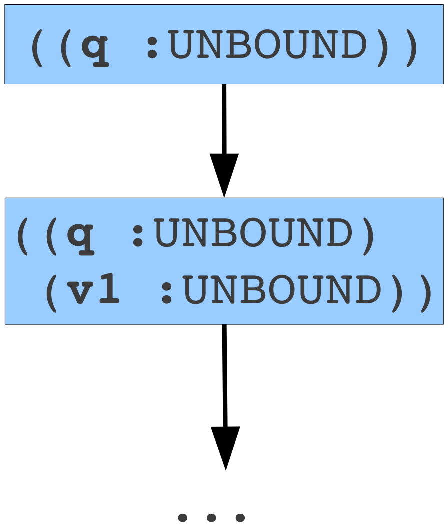
```
(run 1 [q]
  (== 1 q))
;=> (1)
```



Initialising Logic Variables

- **fresh** is similar to let, but initialises unbound (fresh) logic variables

```
(run 1 [q]
  (fresh [v1]
    (== v1 1)
    (== q v1)))
;=> (1)
```



Choice points

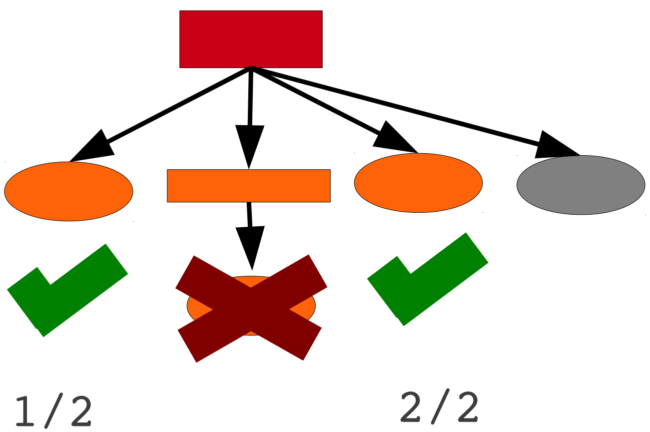
- **conde** is how we define a choice point between multiple alternatives
- Syntax like Scheme's **cond**, but can have 0+ answers

```
(conde
  (<question 1> <answer 1> <answer ..>)
  (<question 2> <answer 1>)
  (<question n>))
```

conde

- **conde** is used as branch point for multiple results

```
(run 2 [q]
  (conde
    ((== q 1))        1/2
    (succeed
     fail)
    (succeed)       2/2
    ((== q 2))))
;=> (1 _.0)
```



Relational Arithmetic

```
(defn succ [p n]
  "p, n are natural numbers such that n
  is the successor of p"
  (conso p [] n))

(def zero 0)
(def one  '(0))

(run 1 [q]
  (succ zero q))
;=> ((0))

(run 1 [q]
  (succ q one))
;=> (0)
```

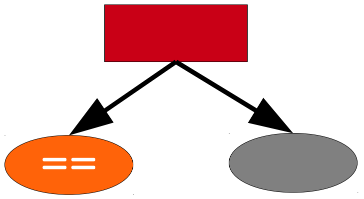
Numbers

```
(defn natural-number [x]
  "x is a natural number"
  (conde
    ((== x zero))
    ((fresh [previous]
       (succ previous x)
       (natural-number previous)))))

(run 1 [q]
  (natural-number one))
;=> (_.0)

(run 6 [q]
  (natural-number q))
;=> (0 (0) ((0)) (((0)))
;    ((((0)))) (((((0))))))
```

Tracing Execution

```
(fresh [q]
  (conde
    ((== q zero))
    ((fresh [prev]
       (succ prev q)
       (natural-number prev)))))
```

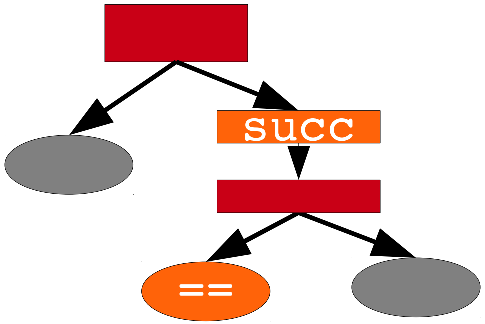
```
(run 6 [q]
  (natural-number q))
;=> (0 (0) ((0)) (((0)))
;    ((((0)))) (((((0))))))
```



Tracing Execution

```
(fresh [q]
  (conde
    ((== q zero))
    ((fresh [prev]
       (succ prev q)
       (conde
         ((== prev zero))
         ((fresh [prev2]
            (succ prev2 prev)
            (natural-number prev2)))))))
```

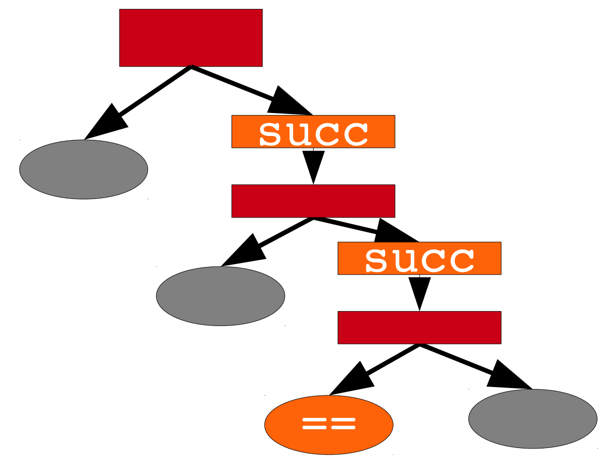
```
(run 6 [q]
  (natural-number q))
;=> (0 (0) ((0)) (((0)))
;    ((((0)))) (((((0))))))
```



Tracing Execution

```
(fresh [q]
  (conde
    ((== q zero))
    ((fresh [prev]
       (succ prev q)
       (conde
         ((== prev zero))
         ((fresh [prev2]
            (succ prev2 prev)
            (conde
              ((== prev2 zero))
              ((fresh [prev3]
                 (succ prev3 prev2)
                 (natural-number prev3))))))))))))
```

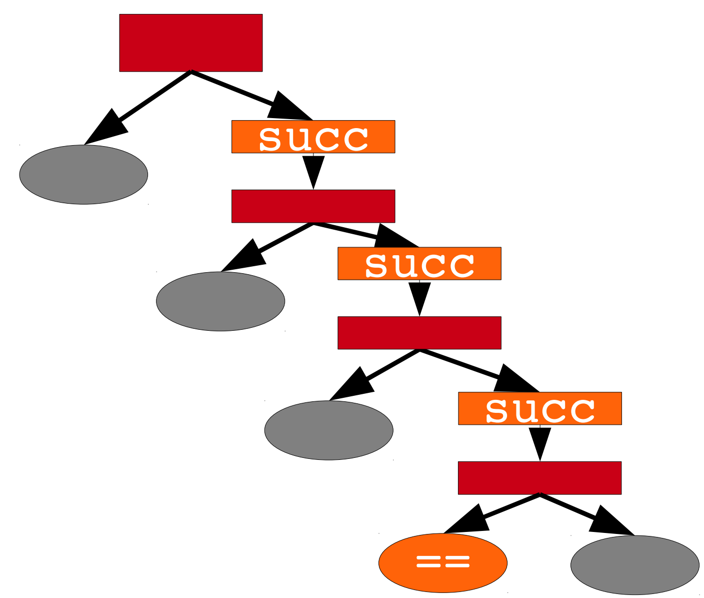
```
(run 6 [q]
  (natural-number q))
;=> (0 (0) ((0)) (((0)))
;    ((((0)))) (((((0))))))
```



Tracing Execution

```
(fresh [q]
  (conde
    ((== q zero))
    ((fresh [prev]
       (succ prev q)
       (conde
         ((== prev zero))
         ((fresh [prev2]
            (succ prev2 prev)
            (conde
              ((== prev2 zero))
              ((fresh [prev3]
                 (succ prev3 prev2)
                 (conde
                   ((== prev3 zero))
                   ((fresh [prev4]
                      (succ prev4 prev3)
                      (natural-number prev4))))))))))))))
```

```
(run 6 [q]
  (natural-number q))
;=> (0 (0) ((0)) (((0)))
;    ((((0)))) (((((0))))))
```



Type Checker for the Simply Typed Lambda Calculus

```
(defn geto [key env value]
  "env is an environment such that the expression key is
  associated with the expression value"
  (matche [env]
          ([[[key :- value] . _]])
          ([[_ . ?rest]] (geto key ?rest value))))

(defn typedo [context exp result-type]
  "`context` is an environment such that expression `exp` executed in
  environment `context` results in type `result-type`"
  (conde
    ((geto exp context result-type))
    ((matche [context exp result-type]
             ([_ [:apply ?fun ?arg] _]
              (fresh [arg-type]
                     (!= ?fun ?arg)
                     (typedo context ?arg arg-type)
                     (typedo context ?fun [arg-type :> result-type])))))))
```

Type Checker..

```
(run 1 [q]
    (typedo [['f :- [Integer :> Integer]]
             ['g :- Integer]]
            [:apply 'f 'g]
            Integer))
;=> (_.0)
```

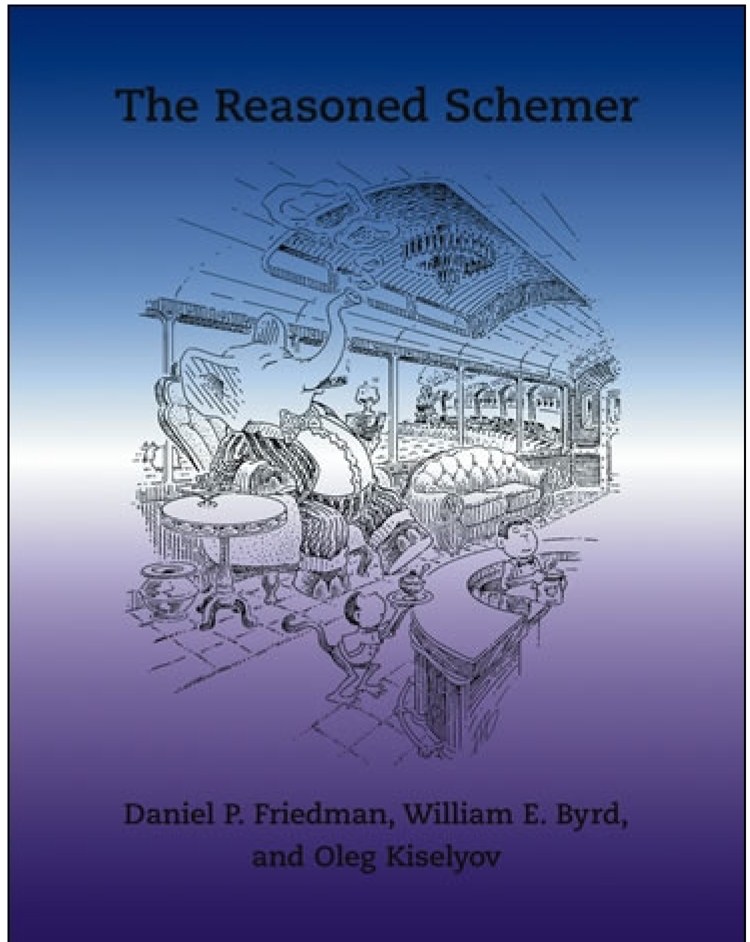
Type Inferencer...

```
(run 1 [q]
    (typedo [['f :- [Integer :> Integer]]
             ['g :- Integer]]
            [:apply 'f 'g]
            q))
;=> (Integer)
```

Code Generator..

```
(run 4 [q]
     (typedo [['f :- [Integer :> Integer]]
              ['g :- Integer]]
             q
             Integer))
;=> (g
;    [:apply f g]
;    [:apply f [:apply f g]]
;    [:apply f [:apply f [:apply f g]]])


(run 2 [q]
     (typedo [['a :- [Integer :> Float]]
              q]
             [:apply 'a 'b]
             Float))
;=> ([[:apply a b] :- java.lang.Float]
;    [b :- java.lang.Integer])
```

Resources


The Reasoned Schemer
Daniel P. Friedman, William E. Byrd,
and Oleg Kiselyov

Resources

- Introduction to Logic Programming with Clojure
- https://github.com/frenchy64/**Logic-Starter**/wiki