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

\[
(\text{cons } 1 \ [2]) \\
\Rightarrow \ [1 \ 2]
\]

\[
(\text{cons}^\circ \ 1 \ [2] \ [1 \ 2]) \\
\Rightarrow \ true
\]
We can use \texttt{cons} as a \texttt{predicate} if all arguments are \texttt{ground values} (not variables)

For \texttt{(cons head tail result)}, \texttt{cons} returns true if \texttt{head} consed onto \texttt{tail} equals \texttt{result}

\[(\text{cons} 1 [2] [1 2])\]
\[=> \text{true}\]

\[(\text{cons} 1 [] [1 2])\]
\[=> \text{false}\]
• We can use $\text{cons}^\circ$ 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^\circ 1 [2] x))
;=> ([1 2])

(solve 1 [x]
  (cons^\circ 1 x [1 2]))
;=> ([2])
```
\( \sqrt{\circ} \)

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

\[
\text{(solve } 2 \ [x] \\
\quad \text{(sqrt}^\circ \ 4 \ x))
\]

\[\implies (2 \ -2)\]
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

\[
\text{run 1 [q] succeed)}
\]
\[=> (_0)
\]
\[
\text{run 1 [q] fail)}
\]
\[=> ()
\]
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?

\[
\text{(run } 1 \ [q] \ \\
(== 1 \ q)) \ \\
;=> \ (1)
\]

```
((q :UNBOUND))
```

```
((q 1))
```

1/1
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

(\texttt{conde} \\
\texttt{(question 1\textgreater{} answer 1\textgreater{} answer ..\textgreater{})} \\
\texttt{(question 2\textgreater{} answer 1\textgreater{)}} \\
\texttt{(question n\textgreater{)})}
conde

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

```
(run 2 [q]
  (conde
    (((== q 1)) ✓ 1/2
    (succeed)
    (fail) ✗ 1/2
    (succeed)
    (((== 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)
(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

\[
\text{(fresh } [q])
\text{cond}
\begin{align*}
& (== \ q \ \text{zero}) \\
& (\text{fresh } [\text{prev}] \\
& \quad (\text{succ } \text{prev } q) \\
& \quad (\text{natural-number } \text{prev}))
\end{align*}
\]

\[
\text{(run 6 } [q])
\text{cond}
\begin{align*}
& (\text{natural-number } q) \\
& ; \Rightarrow (0 \ (0) ((0)) (((0)))) \\
& ; \ (((0))) ((((0))))))
\end{align*}
\]
(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))))
(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)))))))
(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)
```
(run 1 [q]
  (typedo [['f :- [Integer -> Integer]]
            ['g :- Integer]]
    [:apply 'f 'g]
    q))
=> (Integer)
(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