Relational + Logic Programming

Helena Dworak
3/26/2021

0. Prehistory: Predicate Logic as Grammar

1. History: Prolog and logic programming

2. Relational Programming in miniKanren

Prehistory: Predicate Logic as Grammar

1970's : push for natural language processing as part of AI

1971: Colmerauer & Roussel develop Prolog as a means to process natural French Language

1974: Kowalski - Predicate Logic as Grammar

1976: Van Emden & Kowalski - Semantics of Predicate Logic as Grammar

Two big ideas at this time:

1. Natural language processing

2. Programming language without features that are only meaningful to the machine

↓

Programming Language shaped to the problem

Predicate Logic

Horn clause:

$$B_1 \vee \ldots \vee B_m \leftarrow A_1 \wedge \ldots \wedge A_n$$

where $m \leq 1$, i.e. there is at most 1 "B"

(also said "at most one positive")

$$a_1 \wedge \ldots \wedge a_n \to b \equiv \neg a_1 \vee \neg a_2 \vee \ldots \vee \neg a_n \vee b$$

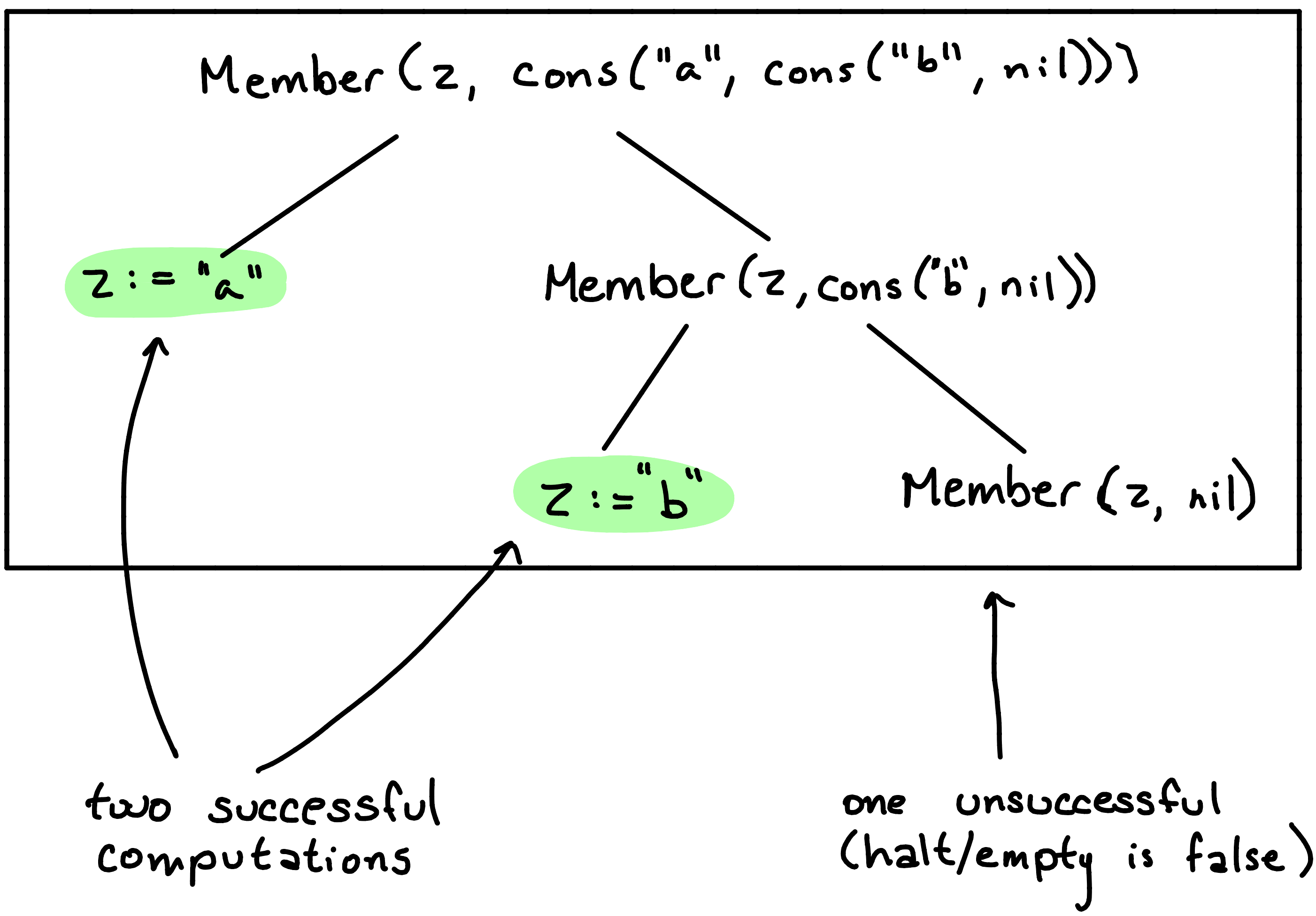
4 types of horn clauses:

1. $n \neq 0, m \neq 0$, i.e. $B \leftarrow A_1 \wedge \ldots \wedge A_n$ — PROCEDURE
2. $n = 0, m \neq 0$, $B \leftarrow$ ____ — ASSERTION
3. $n \neq 0, m = 0$, __ $\leftarrow A_1 \wedge \ldots \wedge A_n$ — GOAL
4. $n = 0, m = 0$, — HALT (false)

Member

Member (x, cons (x , r)) ←

Member (x, cons (y, r)) ← Member (x, r)



Consider append:

$append(nil, z, z) \leftarrow$

$append(cons(x, y), z, cons(x, u)) \leftarrow append(y, z, u)$

append(cons("a", cons("b", nil)), cons("c", cons("d", nil)), ans)

Predicate Logic

- Introduces "input-output" relation (doesn't distinguish between input/output)

```
append( cons("a", cons("b", nil)),
        Z,
        cons("a", cons("b", cons("c", nil))))
```

free variable in what is traditionally the 2nd input arg to append

Predicate logic has no notion of input v. output, just determines relations between them

- known non-determinism
  - More than one procedure can have a name which matches a selected procedure call

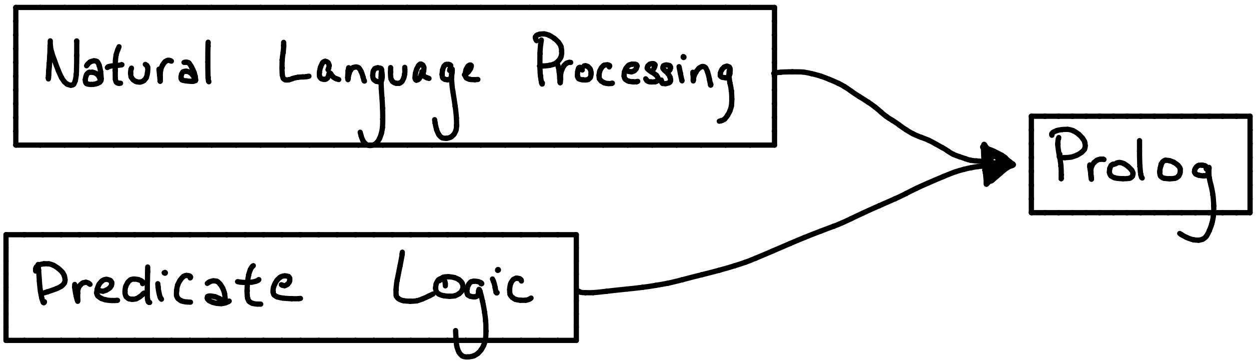
Again, consider `member(z, cons("a", cons("b", nil)))`

member(x, cons(x, r)) ←

member(x, cons(y, r)) ← member(x, r)

- "viability of predicate logic [... depends on development] of auxiliary control language"

Prolog



Developed in 1971 [Colmerauer & Rousell]

Expressed and answered questions in french
& atoms are (nearly) anything

English	Prolog
Helena is a student.	student (Helena).
Is Helena a student? ↳ Yes.	?- student (Helena). ↳ Yes
Which X is a student? ↳ when X is Helena	?- student (X). ↳ X = Helena.

Member from predicate logic

$Member(x, cons(x, r)) \leftarrow$

$Member(x, cons(y, r)) \leftarrow Member(x, r)$

Prolog member function

```
member(X, [X|_]).

member(X, [Y|R]) :- member(X, R).
```

↑ looks like "←" implies

↑ looks like a sentence

Unification

Prolog is a "pattern-matcher"

(pattern matching uncommon at this point)

But how does it match patterns?

→ Unification [Robinson 1965]

cons("a", cons(X, Y)) cons(Z, cons("b", nil))

Unification Cont'd

Variables bound once, creates a permanant relation

(define) equal(x, x).

(try) equal(1, 2).

first, bind x to 1
then, try bind x to 2
find, $x = 1 \neq 2$,
thus should not unify

> Variables can be bound to atoms, other variables, or complex data structures

Binding variables to complex data structures in PROLOG

⇩

TRICKY

Prolog lacks the occurs check

Occurs Check

consider

child (X) = X.

in unification, (bind X child(X)) is valid

X = child(child(child(child(child(child(
child(child(child(child(child(child(
child(child(child(child(. . .

The occurs check will check if x "occurs" in child(x) before binding it

(occurs? x child(x)) → Yes

↑

then bind fails

Prolog omits the occurs check for efficiency and consequentially, this leads to unsoundedness

Not the only issue!

Prolog search tree – DFS

- Not memory intensive:
  doesn't maintain branches to traverse like BFS
- Can get stuck in one branch of the search tree!

In practice, prolog programmers use "extralogical" features such as CUT (!) to stop search

Negation

OPTION 1

```
absent(X, nil).

absent(X, [Y|R]) :- dif(X,Y), absent(X,R).
```

ensures not the same.

dif/2: interesting history

- [1971] • introduced in the original Prolog
- [1973] • removed in next iteration, Prolog I
- [1986] • re-added in Prolog II

Negation-as-failure

OPTION 2

```
absent(X, L) :- not(member(:  '_)).
```

↑ necessary without dif/2.

Historical interlude : timeline

- 1971 : Predicate Logic as Programming Language
- 1971 - 1972 : Birth of Prolog
- 1982 : Fifth Generation Computer System [FGCS]
- 1992 : FGCS 1992 conference (to mark end of FGCS)

Fifth Generation Computer System

1982 – MITI (Japan) started massive initiative for FGCS

- Promised large AI achievements
- named Prolog language of choice
- [too ambitious]

1984 - U.S. Congress passed NCRA [National Cooperative Research Act]

A lot of interest in Prolog + its potential

1992 - FGCS had 'failed', LISP machines were replaced with PCs

Logic Programming + Prolog

- promising use of predicate logic as programming language

- flaws with search tree, unification, and "extralogical features" break true relational behavior

- associated with a project that made many promises (and never delivered)

Relational Programming (miniKanren) 2005

[TRS Friedman et. al. 2018]

- attempts to remedy/remove extralogical features that exist in Prolog

Similarities:

- still routed in predicate logic
- shares concept of unification
- also has necessary search tree

but relational programming in miniKanren solves negation/unification/search issues

First, compare syntax:

Prolog:

```
member(X, [X|R]).
member(X, [Y|R]) :- member(X,R).
```

miniKanren: (on Racket)

```
(define (membero x l)
  (fresh (f r)
    (== l (cons f r))
    (conde
      ((== x f))
      ((membero x r)))))
```

This is the Racket-specific implementation

MiniKanren is built on many languages

Unification

- Contains the occurs check that many implementations of prolog lack
  - mathematically-sound unification

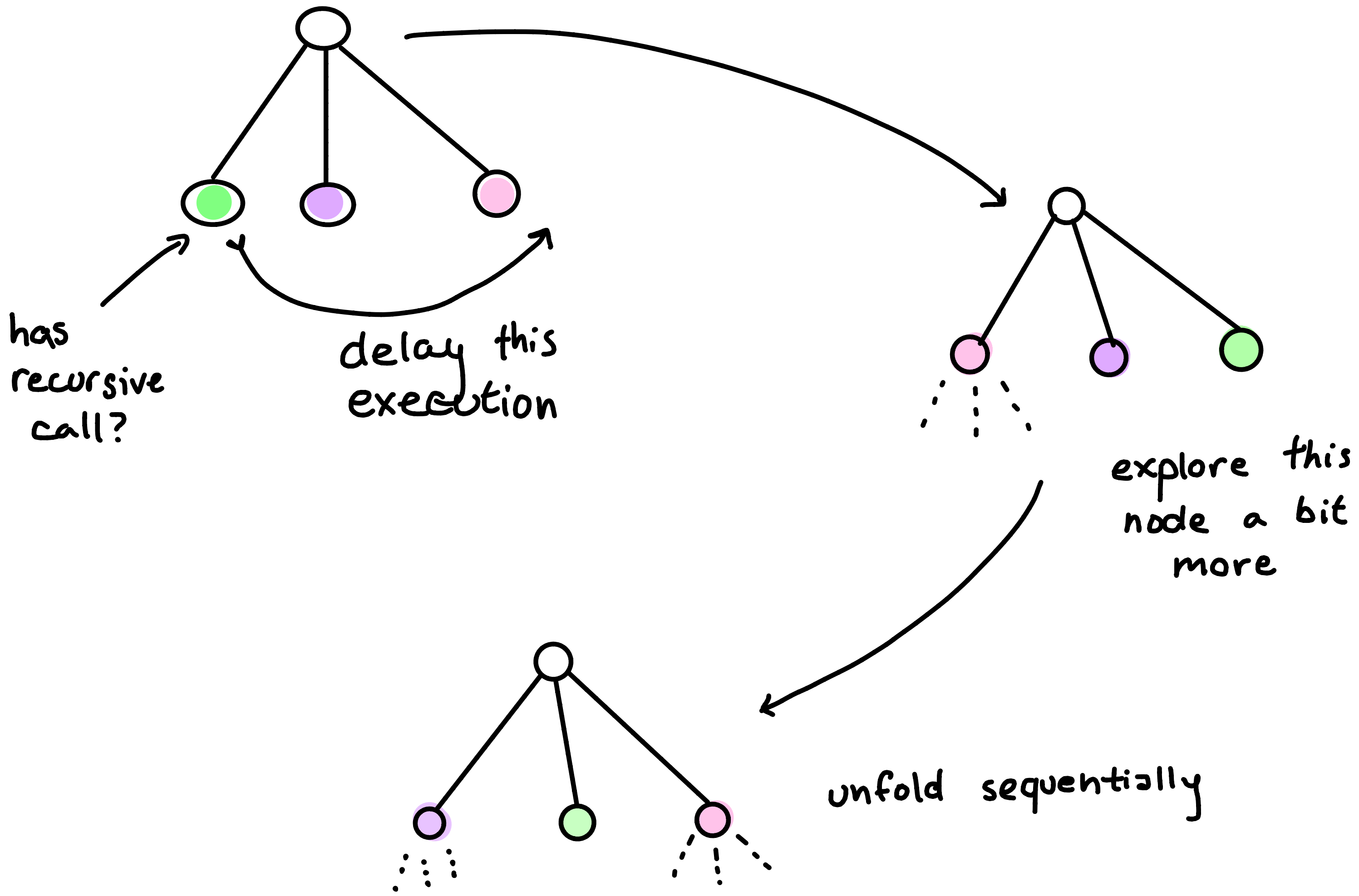
Negation

- no more negation as failure
- can simulate N-A-F with "conda"
  - prunes the search try in a way similar to cut (!)

Search tree

interleaving search

- more memory intensive than DFS
- eliminates or reduces need for cut
- delays when possible before executing recursive call



logic + relational programming is <u>heavily</u> reliant on search tree

- relational programming solves some issues, but not all

- search tree can still diverge
  - may ask for infite answers
  - may ask for a non-existing answer in an infinite search tree

Applications of Relational Programming

- not for every problem
    - we want control over runtime + procedure

- imported as a package to solve problems that are conducive to this type of programming
    - graphs, list manipulation, etc.

- miniKanren lacks a compiler, so it leverages other languages to do the computations

Program Sythesis

evalo:

- semantics for λ-calculus expressed as a relation
- allows us to evaluate how we normally would in Scheme/Racket in our relational world

```
(evalo  p   i   o)
        ↑   ↑   ↑
  program  input  output
```

- we can use evalo to make assertions about our program P and have the interpreter guess the program

[Chirkov, et al. 2020]

Synthesizing Recursion

↳ give interpreter Fib(2) = 1
and Fib(5) = 5

relational programming can show

- the bound n>2
- the base case (return n)
- both recursive calls

Continued potential for development in area of program synthesis

Summary

- Predicate Logic as Grammar
  - logical implication as grammar
    + natural language processing

- Prolog
  - sacrifices made in implimentation for efficiency
    (Unification, Search, Cut, etc.)

- MiniKanren
  - improvements in negation, search, and unification