flow of control, negation, cut, 2nd order programming, tail recursion

Yves Lespérance

Adapted from Peter Roosen-Runge

simplicity hides complexity

- simple *and/or* composition of goals hides complex control patterns
- not easily represented by traditional flowcharts
- may not be a bad thing
- want important aspects of logic and algorithm to be clearly represented and irrelevant details to be left out

procedural and declarative semantics

- Prolog programs have both a declarative/logical semantics and a procedural semantics
- declarative semantics: query holds if it is a logical consequence of the program
- procedural semantics: query succeeds if a matching fact or rule succeeds, etc.
  - defines order in which goals are attempted, what happens when they fail, etc.

and & or

- Prolog's *and* (,) & *or* (; and alternative facts and rules that match a goal) are not purely logical operations
- often important to consider the order in which goals are attempted
  - left to right for "," and ";"
  - top to bottom for alternative facts/rules

and is not always commutative, e.g.

-
  ```
  sublistV1(S, L):- append(_, L1, L),
                    append(S, _, L1).
  ```
  i.e. S is a sublist of L if L1 is any suffix of L and S is a prefix of L1

-
  ```
  sublistV2(S, L):- append(S, _, L1),
                    append(_, L1 ,L).
  ```
  i.e. S is a sublist of L if S is a prefix of some list L1 and L1 is any suffix of L

---

and is not always commutative, e.g.

-
  ```
  ?- sublistV1([c,b], [a, b, c, d]).
   false.
  ```

-
  ```
  sublistV2([c,b], [a, b, c, d]).
   ERROR: Out of global stack
  ```
  why?

---

uses of or (;)

- or “;” can be used to regroup several rules with the same head
- e.g.

  `parent(X,Y):- mother(X,Y); father(X,Y).`
- can improve efficiency by avoiding redoing unification
- “;” has lower precedence than “,”

Prolog negation

- Prolog uses “\+”, “not provable” or negation as failure
- different from logical negation
- `?- \+ goal.` succeeds if `?- goal.` fails
- interpreting \+ as negation amounts to making the closed-world assumption

example

- Given program:

  `human(ulysses). human(penelope).`

  `mortal(X):- human(X).`
- `?- \+ human(jason).`

  `Yes`
- In logic, the axioms corresponding to the program don't entail ¬Human(Jason).

---

semantics of free variables in \+ is "funny"

- normally, variables in a query are existentially quantified from outside

  e.g. `?- p(X), q(X).` represents "there exists *x* such that P(*x*) & Q(*x*)"
- but `?- \+ (p(X), q(X)).` represents "it is not the case that there exists *x* such that P(*x*) & Q(*x*)"

To avoid this problem

- `\+` works correctly if its argument is instantiated
- so for example in

  ```
  intersect([X|L], Y, I):-
      \+ member(X,Y), intersect(L,Y,I).
  ```

  `X` and `Y` should be instantiated

---

example

- Given program:

  `animal(cat). vegetable(turnip).`
- `?- \+ animal(X), vegetable(X).`

  `No` why?
- `?- vegetable(X),\+ animal(X).`

  `X = turnip` why?

guarding the “else”

- can’t rely on implicit negation in predicates that can be redone
- in predicates with alternative rules, each rule should be logically valid (if backtracking can occur)
- safest thing is repeating the condition with negation

---

e.g. intersect

-
```
intersect([], _, []).
intersect([X|L], Y, [X|I]):-
    member(X,Y), intersect(L, Y, I).
intersect([X|L], Y, I):-
    \+ member(X,Y), intersect(L, Y, I).
```
  is OK.

---

e.g. intersect

-
```
intersect([], _, []).
intersect([X|L], Y, [X|I]):-
    member(X,Y), intersect(L, Y, I).
intersect([_|L], Y, I):-intersect(L, Y, I).
```
  is buggy.

  `?- intersect([a], [b, a], []).` succeeds.

  why?

---

inhibiting backtracking

- the **cut** operator “!” is used to control backtracking
- If the goal G unifies with H in program

  H :- ….

  H :- $G_1,\ldots,G_i$, !, $G_j,\ldots, G_k$.

  H :- … .

  and gets past the !, and $G_j,\ldots, G_k$ fails, then the parent goal G immediately fails. $G_1,\ldots, G_i$ won’t be retried and the subsequent matching rules won’t be attempted.

Using ! e.g. intersect

- cut can be used to improve efficiency, e.g.

  intersect([], _, []).
  intersect([X|L], Y, [X|I]):-
      member(X,Y), intersect(L, Y, I).
  intersect(([X|L], Y, I):-
      \+ member(X,Y), intersect(L, Y, I).

  retests member(X,Y) twice

e.g. intersect

- using cut, we can avoid this

  intersect([], _, []).
  intersect([X|L], Y, [X|I]):-
      member(X,Y), !, intersect(L, Y, I).
  intersect([_|L], Y, I):-intersect(L, Y, I).
- means that the last 2 rules are a conditional branch

cut can be used to define useful features

- If goal G should be false when C_1,…, C_n holds, can write

  G :- C_1,…, C_n, !, fail.
- not provable can be defined using cut

  \+ G :- G, !, fail.
  \+ G.

control predicates

- true (really success), e.g.

  G :- Cond1; Cond2; true.
- fail (opposite of true)
- repeat (always succeeds, infinite number of choice points)

  loopUntilNoMore:- repeat, doStuff,
      checkNoMore.

  but tail recursion is cleaner, e.g.

  loop :- doStuff, (checkNoMore; loop).

forcing all solutions

```
test :- member(X, [1, 2, 3]),
   nl, print(X),
   fail.
% no alternative sols for print(X) and nl
% but member has alternative sols
?- test.
1
2
3
No
```

2nd order features: bagof & setof

- `?- bagof(T,G,L).` instantiates L to the list of all instances of T such for which G succeeds, e.g.

```
?- bagof(X,(member(X,[2,5,7,3,5]),X >= 3),L).
X = _G172
L = [5, 7, 3, 5]
Yes
```

2nd order features: bagof & setof

- setof is similar to bagof except that it removes duplicates from the list, e.g.

```
?- setof(X,(member(X,[2,5,7,3,5]),X >= 3),L).
X = _G172
L = [3, 5, 7]
Yes
```

- can collect values of several variables, e.g.

```
?- bagof(pair(X,Y),(member(X,[a,b]),member(Y,[c,d])),
          L).
X = _G157
Y = _G158
L = [pair(a, c), pair(a, d), pair(b, c), pair(b, d)]
Yes
```

2nd order features

- setof and bagof are called 2nd order features because they are queries about the value of a set or relation
- in logic, this is quantification over a set or relation
- not allowed in first order logic, but can be done in 2nd order logic

entering and leaving

- Trace steps are labelled:
  - Call: enter the procedure
  - Exit: exit successfully with bindings for variable
  - Fail: exit unsuccessfully
  - Redo: look for an alternative solution
- 4 ports model

Tail recursion optimization in Prolog

- suppose have goal A and rule A’ :- B_1, B_2, …, B_{n-1}, B_n. and A unifies with A’ and B_2, …, B_{n-1} succeed
- if there are no alternatives left for A and for B_2, …, B_{n-1} then can simply replace A by B_n on execution stack
- in such cases the predicate A is tail recursive
- nothing left to do in A when B_n succeeds or fails/backtracks, so we can replace call stack frame for A by B_n’s; recursion can be as space efficient as iteration

---

e.g. factorial

- simple implementation:

```
fact(0,1).
fact(N,F):- N > 0, N1 is N – 1,
        fact(N1,F1), F is N * F1.
```

- close to mathematical definition
- but not tail-recursive
- requires O(N) in stack space

---

e.g. factorial

- better implementation:

```
fact(N,F):- fact1(N,1,F).
fact1(0,F,F).
fact1(N,T,F):- N > 0, T1 is T * N,
        N1 is N – 1, fact1(N1,T1,F).
```

- uses accumulator
- is tail-recursive and each call can replace the previous call
- can prove correctness

e.g. append

-
  ```
  append([],L,L).
  append([X|R],L,[X|RL]):-
      append(R,L,RL).
  ```
- `append` is tail recursive if first argument is fully instantiated
- Prolog must detect the fact that there are no alternatives left; may depend on clause indexing mechanism used
- use of unification means more relations are tail recursive in Prolog than in other languages

---

split

```
split([],[],[]).
split([X],[X],[]).
split([X1,X2|R],[X1|R1],[X2|R2]):-
  split(R,R1,R2).
```

Tail recursive!

---

merge

```
merge([],L,L).
merge(L,[],L).
merge([X1|R1],[X2|R2],[X1|R]):-
  order(X1,X2), merge(R1,[X2|R2],R).
merge([X1|R1],[X2|R2],[X2|R]):-
  not order(X1,X2), merge([X1|R1],R2,R).
```

Tail recursive, but lack of alternatives may be hard to detect (can use cut to simplify).

---

merge sort

```
mergesort([],[]).
mergesort([X],[X]).
mergesort(L,S):- split(L,L1,L2),
                 mergesort(L1,S1),
                 mergesort(L2,S2),
                 merge(S1,S2,S).
```

for more on tail recursion

- see Sterling & Shapiro The Art of Prolog Sec. 11.2

Example: Finite State Automata

finite state automata

- a finite state automaton (Σ, S, s_0, δ, F) is a representation of a machine as a
  - finite set of states S
  - a state transition relation/table δ
    - mapping current state & input symbol from alphabet Σ to the next state
  - an initial state s_0
  - a set of final states F

accepting an input

- a fsa *accepts* an input sequence from an alphabet Σ if, starting in the designated starting state, scanning the input sequence leaves the automaton in a final state
- sometimes called *recognition*
- e.g. automaton that accepts strings of x's and y's with an even number of x's and an odd number of y's

example

- automaton that accepts strings of x’s and y’s with an even number of x’s and an odd number of y’s
- idea: keep track of whether we have seen even number of x’s and y’s
- S = {ee, eo, oe, oo}
- s_0 = ee
- δ = {(ee, x, oe), (ee, y, eo),…}
- F = {eo}

---

implementation

- fsa(Input) succeeds if and only if the fsa accepts or recognizes the sequence (list) Input.
- initial state represented by a predicate
  - initial_state(State)
- final states represented by a predicate
  - final_states(List)
- state transition table represented by a predicate
  - next_state(State, InputSymbol, NextState)
- note: next_state need not be a function

---

implementing fsa/1

- fsa(Input) :- initial_state(S), scan(Input, S).

  % scan is a Boolean predicate

- scan([], State) :- final_states(F),

  member(State, F).
- scan([Symbol | Seq], State) :- next_state(State, Symbol, Next), scan(Seq, Next).

---

result propagation

- scan uses pumping/result propagation
- carries around current state and remainder of input sequence
- if FSA is deterministic, when end of input is reached, can make an accept/reject decision immediately; tail recursion optimization can be applied
- if FSA is nondeterministic, may have to backtrack; must keep track of remaining alternatives on execution stack

non-determinism

- a non-deterministic fsa accepts an input sequence if there exists *at least one sequence* which leaves the automaton in one of its final states
- ?- fsa(Input).
- scan searches through all possible choices for Symbol at each state;
- fails only if no sequence leads to a final state

representing tables

- can use binary connector, e. g., A-B-C instead of next_state(A,B,C)
  - reduces typing;
  - can make it easier to check for errors
- ee-x-oe. ee-y-eo.
- oe-x-ee. oe-y-oo.
- etc.

revised version

```
scan([], State) :- final_states(F),
    member(State, F).
scan([Symbol | Seq], State) :-
    State-Symbol-Next,
    scan(Seq, Next).
```

Example: modeling and analyzing concurrent processes

process algebra

- concurrent programs are hard to implement correctly
- many subtle non-local interactions
- deadlock occurs when some processes are blocked forever waiting for each other
- process algebra are used to model and analyze concurrent processes

---

deadlocking system example

```
defproc(deadlockingSystem, user1 |
  user2 $ lock1s0 | lock2s0 |
  iterDoSomething).

defproc(user1, acquireLock1 >
  acquireLock2 > doSomething >
  releaseLock2 > releaseLock1).

defproc(user2, acquireLock2 >
  acquireLock1 > doSomething >
  releaseLock1 > releaseLock2).
```

---

deadlocking system example

```
defproc(lock1s0,
  acquireLock1 > lock1s1 ? 0).

defproc(lock1s1, releaseLock1 > lock1s0).

defproc(lock2s0,
  acquireLock2 > lock2s1 ? 0).

defproc(lock2s1,releaseLock2 > lock2s0).

defproc(iterDoSomething,
  doSomething > iterDoSomething ? 0).
```

---

transition relation

- `P – A – RP` means that `P` can do a *single step* by doing action `A` and leaving program `RP` remaining
- *empty program*: `0 – A – P` is always false.
- *primitive action*: `A – A – 0` holds, i. e., an action that has completed leaves nothing more to be done.
- *sequence*: `(A > P) – A – P`
- *nondeterministic choice*: $(P_1\ ?\ P_2)$ – A – P holds if either P_1 – A – P holds or P_2 – A – P holds.

transition relation

- *interleaved concurrency*: $(P_1 \mid P_2) - A - P$ holds if either $P_1 - A - P_{11}$ holds and $P = (P_{11} \mid P_2)$, or $P_2 - A - P_{21}$ holds and $P = (P_1 \mid P_{21})$
- *synchronized concurrency*: $(P_1 \ \$ \ P_2) - A - P$ holds if both $P_1 - A - P_{11}$ holds and $P_2 - A - P_{21}$ holds and $P = (P_{11} \ \$ \ P_{21})$
- *recursive procedures*: `ProcName - A - P` holds if `ProcName` is the name of a procedure that has body `B` and `B - A - P` holds.

can check properties by searching process graph

- a process has an *infinite execution* if there is a cycle in its configuration graph
- e.g. defproc(aloop, a > aloop)
- has_infinite_run(P):- P - _ - PN, has_infinite_run(PN,[P]).
- has_infinite_run(P,V) holds if process P has an infinite run when it has already visited configurations in the list V

checking properties by searching process graph

- cannot_occur(P,A) holds if no execution of P where action A occurs
- search graph for a transition P1 - A - P2
- useful built-in predicate: forall(+Cond, +Action) holds iff for all bindings of Cond, Action succeeds
- e.g. forall(member(C,[8,3,9]), C >= 3) succeeds

cannot_occur examples

- ?- cannot_occur(a > b | a > c, b). succeeds or fails?
- ?- cannot_occur((a > b | a > c)$(a > c), b). succeeds or fails?

whenever_eventually

- whenever_eventually(P,A1,A2) holds if in all executions of P whenever action A1 occurs, action A2 occurs afterwards
- ?- whenever_eventually(a > b > a , a, b). succeeds or fails?
- ?- whenever_eventually(a > b | a > c, a, b). succeeds or fails?

whenever_eventually examples

- ?- whenever_eventually(loop1 , a, b).
  succeeds or fails, where
  defproc(loop1, a > b > loop1)?
- ?- whenever_eventually(loop1 , b, a).
  succeeds or fails, where
  defproc(loop1, a > b > loop1)?
- ?- whenever_eventually(loop2 , b, a).
  succeeds or fails, where
  defproc(loop2, a > b > (loop2 ? 0)).

deadlock_free

- deadlock_free(P) holds if process P cannot reach a deadlocked configuration, i.e. one where the remaining process is not final, but no transition is possible
- ?- deadlock_free(a $ a). succeeds or fails?
- ?- deadlock_free(a > a $ a). succeeds or fails?

deadlock_free examples

- ?- deadlock_free(loop3 $ a). where
  defproc(loop3, (a > loop3) ? 0))

succeeds or fails?