Cuts and Negation

CS 152 -- Programming Paradigms

San José State University

Michael McThrow

November 17, 2021

Agenda

- Backtracking and Search Trees
- Cuts
- Negation
- Project #5 Details

Backtracking and Search Trees

Search Trees

- According to Sterling and Shapiro, "A search tree of a goal G with respect to a program P is defined as follows":
  - G is the tree's root.
  - Nodes are goals (can be conjunctive).
  - "There is an edge leading from a node N for each clause in the program whose head unifies with the selected goal."
  - "Each branch in the tree from the root is a computation of G by P."
  - Leaves are either *success nodes* or *failure nodes*. Each success node is a solution of the query.

Search Tree Example

```
father(abraham,isaac).          male(isaac).
father(haran,lot).              male(lot).
father(haran,milcah).           female(milcah).
father(haran,yiscah).           female(yiscah).

son(X,Y) ← father(Y,X), male(X).
daughter(X,Y) ← father(Y,X), female(X).
```

Program 1.2 Biblical family relationships

From Sterling and Shapiro, p. 23

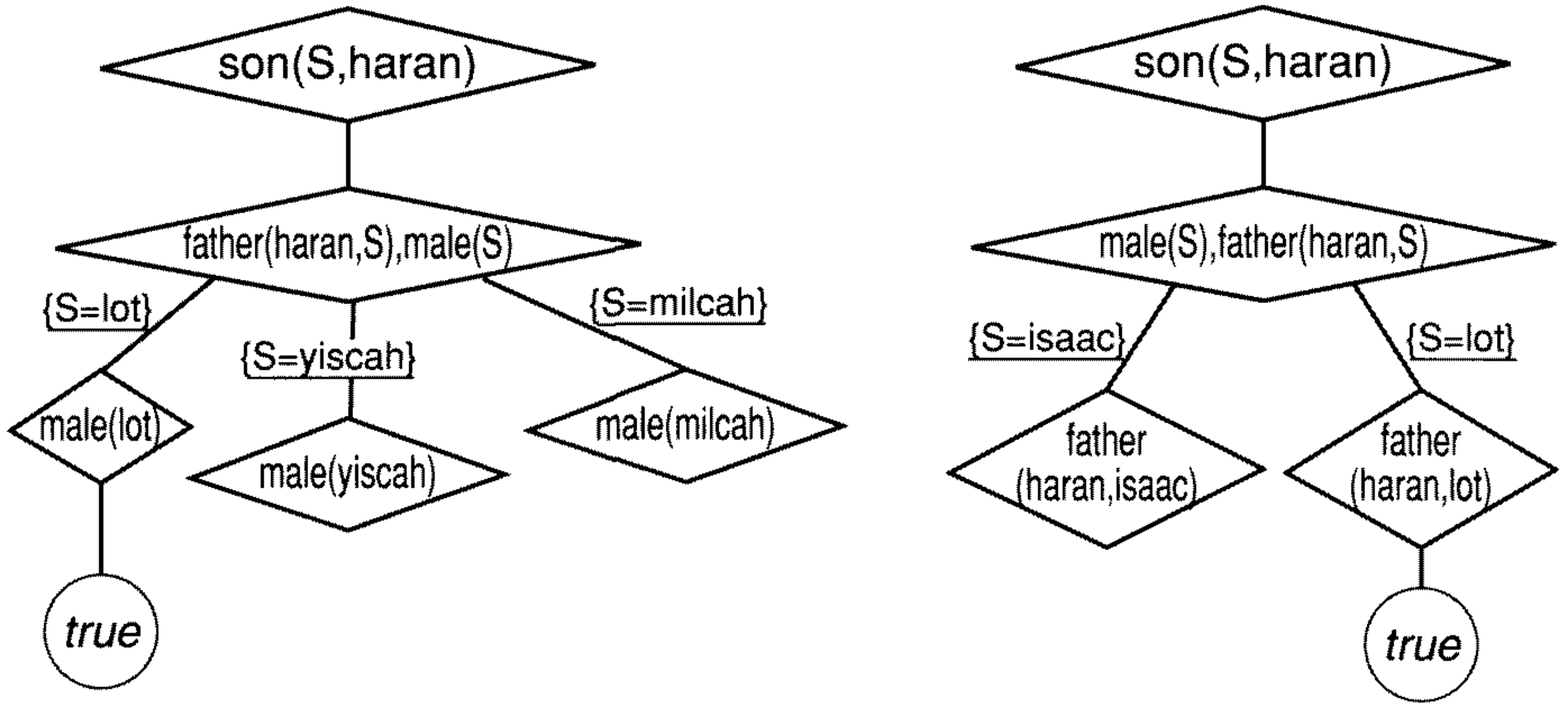

Figure 5.2 Two search trees

From Sterling and Shapiro, p. 111

Search Tree Example

append(*Xs*,*Ys*,*XsYs*) ←
    XsYs is the result of concatenating
    the lists *Xs* and *Ys*.

```
append([ ],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) ← append(Xs,Ys,Zs).
```

Program 3.15 Appending two lists

From Sterling and Shapiro, p. 60

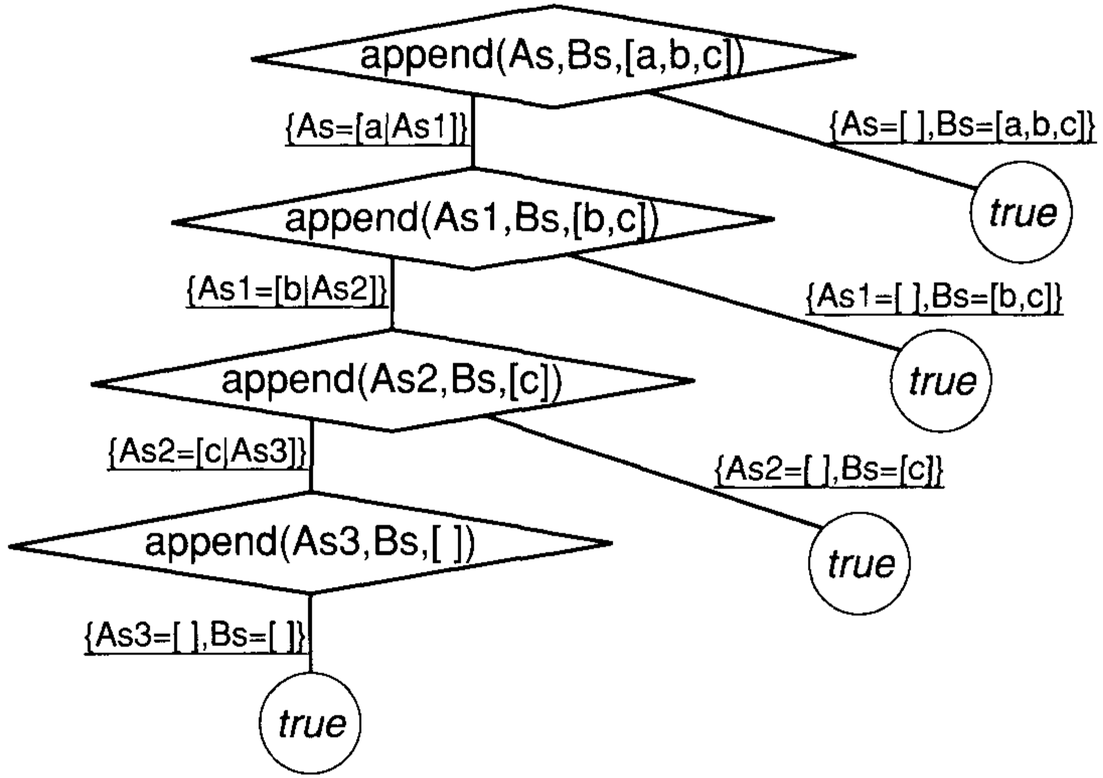

Figure 5.3 Search tree with multiple success nodes

From Sterling and Shapiro, p. 112

Search Trees

- Sometimes there can be multiple possible search trees for a search query.
- Each possible search tree depends on decisions made regarding how new elements are added to the resolvent when resolving the query and how the new clause A' from P is found.

Review: Resolution Algorithm

Input: A goal G and a program P

Output: An instance of G that is a logical consequence of P, or *no* otherwise

Algorithm: Initialize the resolvent to G.
while the resolvent is not empty *do*
choose a goal A from the resolvent
choose a (renamed) clause $A' \leftarrow B_1,\ldots,B_n$ from P
such that A and A' unify with mgu θ
(if no such goal and clause exist, exit the *while* loop)
replace A by $B_1,\ldots,B_n$ in the resolvent
apply θ to the resolvent and to G
If the resolvent is empty, *then* output G, *else* output *no*.

Figure 4.2 An abstract interpreter for logic programs

From Sterling and Shapiro, p. 93

Review: Resolution Algorithm (with Prolog implementation details)

Input: A goal G and a program P

Output: An instance of G that is a logical consequence of P, or *no* otherwise

Algorithm: Initialize the resolvent to G.

while the resolvent is not empty *do*
  choose a goal A from the resolvent
  choose a (renamed) clause $A' \leftarrow B_1,\ldots,B_n$ from P
    such that A and A' unify with mgu θ
  (if no such goal and clause exist, exit the *while* loop)
  replace A by $B_1,\ldots,B_n$ in the resolvent
  apply θ to the resolvent and to G
If the resolvent is empty, *then* output G, *else* output *no*.

resolvent is a stack

A = resolvent.pop()

choose first A' in program P that unifies with A

B1,...,BN are pushed onto the stack

Figure 4.2 An abstract interpreter for logic programs

From Sterling and Shapiro, p. 93

Backtracking

Input: A goal G and a program P

Output: An instance of G that is a logical consequence of P, or *no* otherwise

Algorithm: Initialize the resolvent to G.
while the resolvent is not empty *do*
choose a goal A from the resolvent
choose a (renamed) clause $A' \leftarrow B_1,\ldots,B_n$ from P
such that A and A' unify with mgu θ
(if no such goal and clause exist, exit the *while* loop)
replace A by $B_1,\ldots,B_n$ in the resolvent
apply θ to the resolvent and to G
If the resolvent is empty, *then* output G, *else* output *no*.

Figure 4.2 An abstract interpreter for logic programs

What happens when there is no A'? We **backtrack** to the last A that successfully unified. This allows us to try a different computation path. Note that backtracking is not shown in this algorithm.

From Sterling and Shapiro, p. 93

Cuts

Problems That Arise When Using Prolog

- Unnecessary backtracking in some queries.
  - This unnecessary backtracking leads to wasted computations.
- It would be nice for the programmer to be able to ignore, or "prune" branches of a search tree that the programmer knows are "unfruitful."
- Prolog provides such functionality by providing cuts.

Cuts

- A cut is expressed as a ! in Prolog.
- Whenever Prolog encounters a ! inside of a rule, this means that Prolog will commit to all of the choices made before ! appeared; the interpreter will not backtrack on any decision made before !.

Merge Example from Textbook (p. 190)

merge(*Xs*,*Ys*,*Zs*) ←
Zs is an ordered list of integers obtained from merging the ordered lists of integers *Xs* and *Ys*.

```
merge([X|Xs],[Y|Ys],[X|Zs]) ← X < Y, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ← X=:=Y, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) ← X > Y, merge([X|Xs],Ys,Zs).
merge(Xs,[ ],Xs).
merge([ ],Ys,Ys).
```

Program 11.1 Merging ordered lists

```
merge([X|Xs],[Y|Ys],[X|Zs]) ← X < Y, !, merge(Xs,[Y |Ys],Zs).
```

Replacement of first rule with an included cut

Merge Example with Cuts (p. 192)

merge(*Xs*,*Ys*,*Zs*) ←

Zs is an ordered list of integers obtained from merging the ordered lists of integers *Xs* and *Ys*.

```
merge([X|Xs],[Y|Ys],[X|Zs]) ←
    X < Y, !, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) ←
    X=:=Y, !, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) ←
    X > Y, !, merge([X|Xs],Ys,Zs).
merge(Xs,[ ],Xs) ← !.
merge([ ],Ys,Ys) ← !.
```

Program 11.2 Merging with cuts

Minimum Example with Cuts (p. 193)

minimum(*X*,*Y*,*Min*) ←

Min is the minimum of the numbers *X* and *Y*.

```
minimum(X,Y,X) ← X≤Y, !.
minimum(X,Y,Y) ← X > Y, !.
```

Program 11.3 `minimum` with cuts

Green and Red Cuts

- Green cuts are used for removing unnecessary backtracking in Prolog programs.
  - All examples shown have been examples of green cuts.
  - Green cuts are less controversial.
- Red cuts are used for changing the set of goals the program can prove.
  - "A standard Prolog programming technique using red cuts is the omission of explicit conditions. Knowledge of the behavior of Prolog, specifically the order in which rules are used in a program, is relied on to omit conditions that could be inferred to be true" [Sterling and Shapiro, p. 203].
  - Highly controversial; **USE WITH EXTREME CAUTION!**

Negation

Why should Prolog programmers be cautious about negation?

Recall that in logic programming, if a query results in a "no" or "false" answer, this does not state anything about the truth of the query; it means that the interpreter failed to prove the query from the program [Sterling and Shapiro, p. 13].

However, it is convenient for programmers to express certain logical statements using negation.

Examples of Negation

- single(X) :- not married(X).
- fake(X) :- not authentic(X).
- import(X) :- not domestic(X).

The concept "negation as failure" allows us to express negation in logic programming.

According to Sterling and Shapiro, "A goal `not G` will be assumed to be a consequence of a program P if G is not a consequence of P" (p. 114).

Cuts can be used to implement negation as failure.

Negation as Failure

- Prolog provides a `fail_if(Goal)` predicate, which is equivalent to the `not` statement.
- Prolog also has a system predicate called `fail` that always fails.
- Semantics of `not G` [Sterling and Shapiro, p. 198]:

> Let us consider the behavior of Program 11.6 in answering the query `not G`? The first rule applies, and `G` is called using the meta-variable facility. If `G` succeeds, the cut is encountered. The computation is then committed to the first rule, and `not G` fails. If the call to `G` fails, then the second rule of Program 11.6 is used, which succeeds. Thus `not G` fails if `G` succeeds and succeeds if `G` fails.

Project #5 Details

Project #5 Details

- You will be implementing a simple Prolog interpreter.
  - No numbers, no type predicates, no meta-logical predicates, no cuts or negation; just the Prolog you learned in the first 5-6 chapters of *The Art of Prolog*.
  - The key is implementing resolution, unification, and backtracking correctly. You will use the resolution and unification algorithms from the textbook.
- Choices of programming languages for implementation:
  - C, C++, Java, Python, R5RS Scheme (`#lang r5rs` in DrRacket), Racket (`#lang racket` in DrRacket).
  - If you want to use a different language, ask me in advance (however, logic programming languages are banned).
- May work with one partner. Only one partner has to submit, but both names need to be in the submission files.
- Feel free to test your interpreter on your Project #4 submissions (though any code taking advantage of features not in the first six chapters of the textbook won't work).
- **Due date:** Sunday, December 12 at 11:59pm Pacific Standard Time. Late submissions will be accepted until Wednesday, December 15 at 11:59pm PST.

The specifications for Projects #5 and #6 will be posted no later than Saturday morning.