6.

Procedural Control of Reasoning

Declarative / procedural

Theorem proving (like resolution) is a general domain-independent method of reasoning

Does not require the user to know how knowledge will be used

will try all logically permissible uses

Sometimes we have ideas about how to use knowledge, how to search for derivations

do not want to use arbitrary or stupid order

Want to communicate to theorem-proving procedure some guidance based on properties of the domain

- perhaps specific method to use
- perhaps merely method to avoid

Example: directional connectives

In general: control of reasoning

DB + rules

Can often separate (Horn) clauses into two components:

Example:

MotherOf(jane,billy) FatherOf(john,billy) FatherOf(sam, john) ...	a database of facts • basic facts of the domain • usually ground atomic wffs
ParentOf(x,y) ⇐ MotherOf(x,y) ParentOf(x,y) ⇐ FatherOf(x,y) ChildOf(x,y) ⇐ ParentOf(y,x) AncestorOf(x,y) ⇐	collection of rules • extends the predicate vocabulary • usually universally quantified conditionals

Both retrieved by unification matching

Control issue: how to use the rules

Rule formulation

Consider AncestorOf in terms of ParentOf

Three logically equivalent versions:

1. AncestorOf(x,y) ⇐ ParentOf(x,y)
   AncestorOf(x,y) ⇐ ParentOf(x,z) ∧ AncestorOf(z,y)
2. AncestorOf(x,y) ⇐ ParentOf(x,y)
   AncestorOf(x,y) ⇐ ParentOf(z,y) ∧ AncestorOf(x,z)
3. AncestorOf(x,y) ⇐ ParentOf(x,y)
   AncestorOf(x,y) ⇐ AncestorOf(x,z) ∧ AncestorOf(z,y)

Back-chaining goal of AncestorOf(sam,sue) will ultimately reduce to set of ParentOf(–,–) goals

1. get ParentOf(sam,z): find child of Sam searching *downwards*
2. get ParentOf(z,sue): find parent of Sue searching *upwards*
3. get ParentOf(–,–): find parent relations searching *in both directions*

Search strategies are not equivalent

if more than 2 children per parent, (2) is best

Algorithm design

Example: Fibonacci numbers

1, 1, 2, 3, 5, 8, 13, 21, ...

Version 1:

Fibo(0, 1)

Fibo(1, 1)

Fibo(s(s(n)), x) $\Leftarrow$ Fibo(n, y) $\wedge$ Fibo(s(n), z) $\wedge$ Plus(y, z, x)

Requires *exponential* number of Plus subgoals

Version 2:

Fibo(n, x) $\Leftarrow$ F(n, 1, 0, x)

F(0, c, p, c)

F(s(n), c, p, x) $\Leftarrow$ Plus(p, c, s) $\wedge$ F(n, s, c, x)

Requires only *linear* number of Plus subgoals

Ordering goals

Example:

AmericanCousinOf(x,y) $\Leftarrow$ American(x) $\wedge$ CousinOf(x,y)

In back-chaining, can try to solve either subgoal first

Not much difference for AmericanCousinOf(fred, sally), but big difference for AmericanCousinOf(x, sally)

1. find an American and then check to see if she is a cousin of Sally
2. find a cousin of Sally and then check to see if she is an American

So want to be able to order goals

better to <u>generate</u> cousins and <u>test</u> for American

In Prolog: order clauses, and literals in them

Notation: G :- $G_1, G_2, ..., G_n$ stands for

$G \Leftarrow G_1 \wedge G_2 \wedge ... \wedge G_n$

but goals are attempted in presented order

Commit

Need to allow for backtracking in goals

AmericanCousinOf(x,y) :- CousinOf(x,y), American(x)

for goal AmericanCousinOf(x,sally), may need to try to solve the goal American(x) for many values of x

But sometimes, given clause of the form

G :- *T*, *S*

goal *T* is needed only as a test for the applicability of subgoal *S*

- if *T* succeeds, commit to *S* as the *only* way of achieving goal *G*.
- if *S* fails, then *G* is considered to have failed
  - do not look for other ways of solving *T*
  - do not look for other clauses with *G* as head

In Prolog: use of cut symbol

Notation: G :- T_1, T_2, ..., T_m, !, G_1, G_2, ..., G_n

attempt goals in order, but if all T_i succeed, then commit to G_i

If-then-else

Sometimes inconvenient to separate clauses in terms of unification:

G(zero, –) :- *method 1*
G(succ(n), –) :- *method 2*

For example, may split based on computed property:

Expt(a, n, x) :- Even(n), ... (*what to do when* n *is even*)
Expt(a, n, x) :- Even(s(n)), ... (*what to do when* n *is odd*)

want: check for even numbers only once

Solution: use ! to do if-then-else

G :- *P*, !, *Q*.
G :- *R*.

To achieve *G*: if *P* then use *Q* else use *R*

Example:

Expt(a, n, x) :- n = 0, !, x = 1.
Expt(a, n, x) :- Even(n), !, (*for even* n)
Expt(a, n, x) :- (*for odd* n)

Note: it would be correct to write

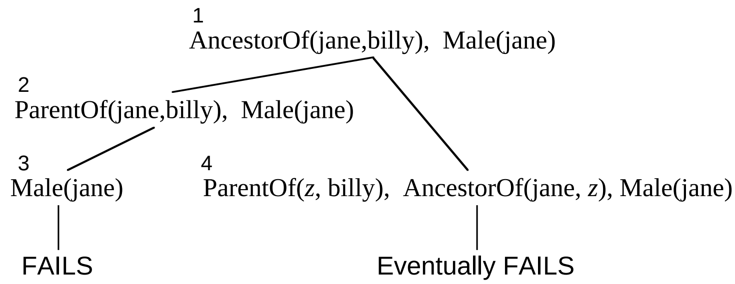
Expt(a, 0, x) :- !, x = 1.

but not

Expt(a, 0, 1) :- !.

Controlling backtracking

Consider solving a goal like



So goal should really be: AncestorOf(jane,billy), !, Male(jane)

Similarly:

Member(x,l) $\Leftarrow$ FirstElement(x,l)
Member(x,l) $\Leftarrow$ Rest(l,l') $\wedge$ Member(x,l')

If only to be used for testing, want

Member(x,l) :- FirstElement(x,l), !, .

On failure, do not try to find another x later in the rest of the list

Negation as failure

Procedurally: we can distinguish between the following:

can solve goal $\neg G$ *vs.* cannot solve goal G

Use **not**(G) to mean the goal that succeeds if G fails, and fails if G succeeds

Roughly: **not**(G) :- G, !, fail. /* fail if G succeeds */
not(G). /* otherwise succeed */

Only terminates when failure is *finite* (no more resolvents)

Useful when DB + rules is complete

NoChildren(x) :- **not**(ParentOf(x,y))

or when method already exists for complement

Composite(n) :- $n > 1$, **not**(PrimeNum(n))

Declaratively: same reading as ¬, but not when *new* variables in G

[**not**(ParentOf(x,y)) $\supset$ NoChildren(x)] ✓

vs. [¬ParentOf(x,y) $\supset$ NoChildren(x)] ✗

Dynamic DB

Sometimes useful to think of DB as a snapshot of the world that can be changed dynamically

assertions and deletions to the DB

then useful to consider 3 procedural interpretations for rules like

$$\text{ParentOf}(x,y) \Leftarrow \text{MotherOf}(x,y)$$

1. If-needed: Whenever have a goal matching ParentOf(x,y), can solve it by solving MotherOf(x,y)

   ordinary back-chaining, as in Prolog

2. If-added: Whenever something matching MotherOf(x,y) is added to the DB, also add ParentOf(x,y)

   forward-chaining

3. If-removed: Whenever something matching ParentOf(x,y) is removed from the DB, also remove MotherOf(x,y), if this was the reason

   keeping track of dependencies in DB

Interpretations (2) and (3) suggest demons

procedures that monitor DB and fire when certain conditions are met

The Planner language

Main ideas:

1. DB of facts

   (Mother susan john) (Person john)

2. If-needed, if-added, if-removed procedures consisting of
   - body: program to execute
   - pattern for invocation **(**Mother *x* *y***)**

3. Each program statement can succeed or fail
   - **(goal** *p***)**, **(assert** *p***)**, **(erase** *p***)**,
   - **(and** *s* ... *s***)**, statements with backtracking
   - **(not** *s***)**, negation as failure
   - **(for** *p* *s***)**, do *s* for every way *p* succeeds
   - **(finalize** *s***)**, like cut
   - a lot more, including all of Lisp

Shift from proving conditions to making conditions hold!

examples:

```
(proc if-needed (cleartable)
   (for (on x table)
      (and (erase (on x table)) (goal (putaway x)))))
(proc if-removed (on x y) (print x " is no longer on " y))
```