8b.
Procedural Control of Reasoning

Outline

- Need for procedural control
  - How to represent knowledge to allow control of reasoning?
- Examples of procedural control
  - We will consider the simple case of backward chaining
    - Conjunct ordering
    - The cut operator
    - Negation as failure
      - Closed world reasoning
- The planner language

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 Battleship(x) → Gray (x)

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)
...
$\text{ParentOf}(x,y) \Leftarrow \text{MotherOf}(x,y)$
$\text{ParentOf}(x,y) \Leftarrow \text{FatherOf}(x,y)$
$\text{ChildOf}(x,y) \Leftarrow \text{ParentOf}(y,x)$
$\text{AncestorOf}(x,y) \Leftarrow$ ...
...

a database of facts
- basic facts of the domain
- usually ground atomic wffs

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. $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,y)$
   $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,z) \wedge \text{AncestorOf}(z,y)$
2. $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,y)$
   $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(z,y) \wedge \text{AncestorOf}(x,z)$
3. $\text{AncestorOf}(x,y) \Leftarrow \text{ParentOf}(x,y)$
   $\text{AncestorOf}(x,y) \Leftarrow \text{AncestorOf}(x,z) \wedge \text{AncestorOf}(z,y)$

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:

$\text{Fibo}(0, 1)$

$\text{Fibo}(1, 1)$

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

Requires *exponential* number of Plus subgoals

Version 2:

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

$\text{F}(0, c, p, c)$

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

Requires only *linear* number of Plus subgoals

This example illustrates that reasoning control is not so different than looking for efficient algorithms for computational tasks

Ordering goals

Example:

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

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 generate cousins and test 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

Ordering Goals is not Enough

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

Cut as an If-then-else control structure

Consider defining exponentiation a^n

- We could do n-1 multiplications
- Or, $\log_2(n)$ multiplications if we work with a^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) :- !.

Specifying Control in a Query

- Sometimes we know which steps in a proof might be helpful. Consider:

  Cousin(x,y) <= (x ≠ y) /\ ¬Sibling(x,y) /\ GParent(z,x) /\ GParent(z,y)

  Suppose, we want to show that Jane is an American cousin of Billy, ie, we want to prove

  cousin (jane, billy), american (jane)

  -> Henry is a grandparent of both, but Jane is not American
  -> Elizabeth is a second grandparent of both, and backtracking will test american(jane) again

  We can control this behavior by saying

  cousin (jane, billy), ! , american (jane)

Control could be either in KB or the Query

- Consider the definition

  Member(x,l) <= FirstElement(x,l)

  Member(x,l) <= RemainingElements(l,l') /\ Member(x, l')

- Suppose, we want to prove that an object is an element of a list, and has a property Q, ie,

  Member (a, c) /\ Q(a)

- If Member(a.c) were to succeed but Q(a) fails, it does not make sense to see if a occurs later in the list. We can control this by saying

  Member (a, c), !, Q(a)

- If we knew, that we only care for membership check, the same control could be specified in the KB:

  Member(x,l) <= FirstElement(x,l), !.

  Member(x,l) <= RemainingElements(l,l') /\ Member(x, l')

- Allows us to ask queries without control information

  member(george, Friends), rich(george)

Negation As Failure

- While proving a goal G, we can encounter two situations
  - Derive an explicit proof for ¬G
    - Given male (John), male(x)→ ¬ Female(x), we can prove that ¬female(John)
  - Fail to prove G
    - ie, we run out of options in trying to show that G is true
      - Given only male(John), we cannot show ¬ female(John)
    - we assume that G must be false: negation as failure

```
not(G) :- G, !, fail.        (fail if G succeeds)
not(G)                       (otherwise succeed)
```

Negation As Failure (NAF)

- Works only when failure is finite
  - If the proof results in an infinite search space, or does not terminate, we cannot use this approach
  - Some systems assume failure after a certain timeout period
- Especially useful in situations
  - when the KB can be assumed to be complete for a specific predicate

    noChildren (x) :- not(parent (x,y)).

    Some systems will let a user mark certain predicates *complete.*
  - -- We have a complete method of computing something

    composite(N) :- N>1, not (primeNumber (N)).

Open-World Assumption

Classical Negation gives open world semantics

Example:

OWL 2 has *open-world* semantics, so negation in OWL 2 is the same as in classical (first-order) logic. To understand open-world semantics, consider the ontology consisting of the following assertion.

```
ClassAssertion( a:Dog a:Brian )
```

Brian is a dog.

One might expect *a:Brian* to be classified as an instance of the following class expression:

```
ObjectComplementOf( a:Bird )
```

Intuitively, the ontology does not explicitly state that *a:Brian* is an instance of *a:Bird*, so this statement seems to be false. In OWL 2, however, this is not the case: it is true that the ontology does not state that *a:Brian* is an instance of *a:Bird*; however, the ontology does not state the opposite either. In other words, this ontology simply does not contain enough information to answer the question whether *a:Brian* is an instance of *a:Bird* or not: it is perfectly possible that the information to that effect is actually true but it has not been included in the ontology.

Closed-World Assumption

Use of NAF gives us closed world semantics

There are usually more negative facts than positive facts. A flight database would contain facts such as:

- DirectConnect(cleveland, toronto)
- DirectConnect(toronto, northBay)
- DirectConnect(cleveland, phoenix)
- But, not facts such as:

  ¬DirectConnect(northBay, phoenix)

But note: KB $\not\models$ -ve facts (would have to answer: "I don't know")

Proposal: a new version of entailment: KB $\models_c \alpha$ iff KB $\cup$ *Negs* $\models \alpha$

where *Negs* = $\{\neg p \mid p$ atomic and KB $\not\models p\}$

Note: relation to negation as failure

a common pattern:
KB´ = KB $\cup$ Δ

Gives: KB $\models_c$ +ve facts and -ve facts

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 $\text{ParentOf}(x,y)$, can solve it by solving $\text{MotherOf}(x,y)$

   ordinary back-chaining, as in Prolog

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

   forward-chaining

3. If-removed: Whenever something matching $\text{ParentOf}(x,y)$ is removed from the DB, also remove $\text{MotherOf}(x,y)$, 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))
```

Recommended Reading

- Chapter 6 of Brachman & Levesque