COMP4418: Knowledge Representation and Reasoning

Procedural Control

Maurice Pagnucco
School of Computer Science and Engineering

COMP4418, Week 3

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 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 ATP procedure guidance based on properties of domain

- perhaps specific method to us
- perhaps merely method to avoid

Example: directional connectives
In general: control of reasoning

DB + rules

Can often separate (Horn) clauses into two components:

- database of facts
  - basic facts of the domain
  - usually ground atomic wffs
- collection of rules
  - extend vocabulary in terms of basic facts
  - usually universally quantified conditionals

Both retrieved by unification matching

Example:

MotherOf(jane,billy)
FatherOf(john,billy)
FatherOf(sam,john)
. . .
ParentOf(x,y) $\leftarrow$ MotherOf(x,y)
ParentOf(x,y) $\leftarrow$ FatherOf(x,y)
ChildOf(x,y) $\leftarrow$ ParentOf(y,x)
. . .

Control Issue: how to use rules

Rule formulation

Consider AncestorOf in terms of ParentOf

Three logically equivalent versions:

1. AncestorOf(x,y) $\Leftarrow$ ParentOf(x,y)
   AncestorOf(x,y) $\Leftarrow$ ParentOf(x,z) $\wedge$ AncestorOf(z,y)
2. AncestorOf(x,y) $\Leftarrow$ ParentOf(x,y)
   AncestorOf(x,y) $\Leftarrow$ ParentOf(z,y) $\wedge$ AncestorOf(x,z)
3. AncestorOf(x,y) $\Leftarrow$ ParentOf(x,y)
   AncestorOf(x,y) $\Leftarrow$ AncestorOf(x,z) $\wedge$ AncestorOf(z,y)

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

1. get	ParentOf(sam,z): searches *downward* from Sam	find child of Sam
2. get	ParentOf(z,sue): searches *upward* from Sue	find parent of Sue
3. get	ParentOf(-,-): searches in both directions	find parent relations

Search strategies are not equivalent

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

B&L (2005)

Algorithm design

Example: Fibonacci numbers

$1, 1, 2, 3, 5, 8, 13, 21, \ldots$

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

B&L (2005)

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)

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 :- $G_1, G_2, \ldots, G_n$ stands for $G \Leftarrow G_1 \wedge G_2 \wedge \ldots \wedge G_n$
- 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 AmericanCounsinOf(x,sally), may need to try American(x) for various 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

In other words: if T succeeds, commit to S as the *only* way of achieving goal G.

- so 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 :- $T_1, T_2, \ldots, T_m$, !, $G_1, G_2, \ldots, G_n$
- attempt goals in order, but if all T_i succeed, then commit to G_i

B&L (2005)

If-then-else

Sometimes inconvenient to separate clauses in terms of unification, as in

G(zero, –) :- *method 1*

G(succ(*n*), –) :- *method 2*

For example, might not have distinct cases:

NumberOfParentsOf(adam, 0)

NumberOfParentsOf(eve, 0)

NumberOfParentsOf(*x*, 2)

want: 2 for everyone except Adam and Eve

Or cases 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*.

Expt(*a*, *n*, *x*) :- Even(*n*), !, *(for even n)*

Expt(*a*, *n*, *x*) :- *(for odd n)*

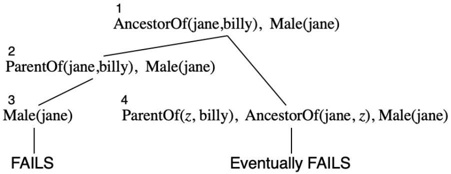
NumberOfParentsOf(adam, 0) :- !

NumberOfParentsOf(eve, 0) :- !

NumberOfParentsOf(*x*, 2)

Controlling backtracking

Consider a goal



So goal should 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 rest of list

Negation as failure

Procedurally: can distinguish between

- can solve goal $\neg G$
- cannot solve G

Use not(G) to mean 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 vs. infinite branch

Useful when DB + rules is complete

NoParents(x) :- not(ParentOf(z,x))

or when method already exists for complement

Composite(n) :- not(PrimeNum(n))

Declaratively: same reading as $\neg$, but complications with *new* variables in G

[not(ParentOf(z,x)) $\rightarrow$ NoParents(x)]

vs. $\neg$ParentOf(z,x) $\rightarrow$ NoParents(x)]