5.

Reasoning with Horn Clauses

Horn clauses

Clauses are used two ways:

- as disjunctions: \((\text{rain} \lor \text{sleet})\)
- as implications: \((\neg \text{child} \lor \neg \text{male} \lor \text{boy})\)

Here focus on 2nd use

Horn clause = at most one +ve literal in clause

- positive / definite clause = exactly one +ve literal
  e.g. \([\neg p_1, \neg p_2, ..., \neg p_n, q]\)

- negative clause = no +ve literals
  e.g. \([\neg p_1, \neg p_2, ..., \neg p_n]\) and also \([\text{ ]}\]

Note: \([\neg p_1, \neg p_2, ..., \neg p_n, q]\) is a representation for
\((\neg p_1 \lor \neg p_2 \lor ... \lor \neg p_n \lor q)\) or \([(p_1 \land p_2 \land ... \land p_n) \supset q]\)

so can read as: If \(p_1\) and \(p_2\) and \(...\) and \(p_n\), then \(q\)

and write as: \(p_1 \land p_2 \land ... \land p_n \Rightarrow q\) or \(\neg q \Leftarrow p_1 \land p_2 \land ... \land p_n\)
Resolution with Horn clauses

Only two possibilities:

It is possible to rearrange derivations of negative clauses so that all new derived clauses are negative

Further restricting resolution

Can also change derivations such that each derived clause is a resolvent of the previous derived one (negative) and some positive clause in the original set of clauses

- Since each derived clause is negative, one parent must be positive (and so from original set) and one parent must be negative.
- Chain backwards from the final negative clause until both parents are from the original set of clauses
- Eliminate all other clauses not on this direct path

This is a recurring pattern in derivations

- See previously:
  - example 1, example 3, arithmetic example
- But not:
  - example 2, the 3 block example
SLD Resolution

An SLD-derivation of a clause \( c \) from a set of clauses \( S \) is a sequence of clause \( c_1, c_2, ... c_n \) such that \( c_n = c \), and

1. \( c_1 \in S \)
2. \( c_{i+1} \) is a resolvent of \( c_i \) and a clause in \( S \)

Write: \( S \xrightarrow{\text{SLD}} c \)

Note: SLD derivation is just a special form of derivation and where we leave out the elements of \( S \) (except \( c_1 \))

In general, cannot restrict ourselves to just using SLD-Resolution

Proof: \( S = \{(p, q), (p, \neg q), (\neg p, q), (\neg p, \neg q)\} \). Then \( S \rightarrow [] \).

Need to resolve some \([ \rho ]\) and \([ \overline{\rho} ]\) to get \([]\).
But \( S \) does not contain any unit clauses.
So will need to derive both \([ \rho ]\) and \([ \overline{\rho} ]\) and then resolve them together.

Completeness of SLD

However, for Horn clauses, we can restrict ourselves to SLD-Resolution

Theorem: SLD-Resolution is refutation complete for Horn clauses: \( H \rightarrow [] \) iff \( H \xrightarrow{\text{SLD}} [] \)

So: \( H \) is unsatisfiable iff \( H \xrightarrow{\text{SLD}} [] \)

This will considerably simplify the search for derivations

Note: in Horn version of SLD-Resolution, each clause in the \( c_1, c_2, ..., c_n \), will be negative
So clauses \( H \) must contain at least one negative clause, \( c_i \)
and this will be the only negative clause of \( H \) used.

Typical case:
- KB is a collection of positive Horn clauses
- Negation of query is the negative clause
Example 1 (again)

**KB**

- FirstGrade
- FirstGrade ⊃ Child
- Child ∧ Male ⊃ Boy
- Kindergarten ⊃ Child
- Child ∧ Female ⊃ Girl
- Female

**SLD derivation**

```
[¬Girl]
  |   
[¬Child, ¬Female]
   |      
[¬Child]
    | 
[¬FirstGrade]
     | 
[]
```

**alternate representation**

```
Goal
  |   
Girl
    |   
Child
      |   
Female
        |   
FirstGrade
          |   
[]
```

Show KB ∪ {¬Girl} unsatisfiable

A goal tree whose nodes are atoms, whose root is the atom to prove, and whose leaves are in the KB

Prolog

Horn clauses form the basis of Prolog

```
Append(nil,y,y)
Append(x,y,z) ⇒ Append(cons(w,x),y,cons(w,z))
```

What is the result of appending [c] to the list [a,b]?

```
Append(cons(a,cons(b,nil)), cons(c,nil), u) goal
  |  
  | u / cons(a,u')
  |  
Append(cons(b,nil), cons(c,nil), u')
    |  
    | u' / cons(b,u'')
    |  
    | Append(nil, cons(c,nil), u'') solved: u'' / cons(c,nil)
```

So goal succeeds with u = cons(a,cons(b,cons(c,nil))))

That is: Append([a b],[c],[a b c])
Back-chaining procedure

\[ \text{Solve}[q_1, q_2, \ldots, q_n] = \]
\[ \text{If } n=0 \text{ then return YES; empty clause detected} \]
\[ \text{For each } d \in \text{KB do} \]
\[ \text{If } d = [q_1, \neg p_1, \neg p_2, \ldots, \neg p_m] \text{ match first } q \]
\[ \text{and replace } q \text{ by -ve lits} \]
\[ \text{Solve}[p_1, p_2, \ldots, p_m, q_2, \ldots, q_n] \text{ recursively} \]
\[ \text{then return YES} \]
\[ \text{end for; can't find a clause to eliminate } q \]
\[ \text{Return NO} \]

Depth-first, left-right, back-chaining

- depth-first because attempt \( p_i \) before trying \( q_i \)
- left-right because try \( q_i \) in order, 1, 2, 3, ...
- back-chaining because search from goal \( q \) to facts in KB \( p \)

This is the execution strategy of Prolog
First-order case requires unification etc.

Problems with back-chaining

Can go into infinite loop
tautologous clause: \([p, \neg p]\) (corresponds to Prolog program with \( p :- \neg p \)).

Previous back-chaining algorithm is inefficient

Example: Consider \( 2n \) atoms, \( p_0, \ldots, p_{n-1}, q_0, \ldots, q_{n-1} \) and \( 4n-4 \) clauses

\[ (p_{i-1} \Rightarrow p_i), (q_{i-1} \Rightarrow p_i), (p_{i-1} \Rightarrow q_i), (q_{i-1} \Rightarrow q_i). \]

With goal \( p_k \) the execution tree is like this

\[ \text{Solve}[p_i] \text{ eventually fails after } 2^k \text{ steps!} \]

Is this problem inherent in Horn clauses?
Forward-chaining

Simple procedure to determine if Horn KB $\models q$.

main idea: mark atoms as solved

1. If $q$ is marked as solved, then return YES
2. Is there a $\{p_1, \neg p_2, \ldots, \neg p_n\} \subseteq$ KB such that
   $p_2, \ldots, p_n$ are marked as solved, but the
   positive lit $p_1$ is not marked as solved?
      no: return NO
      yes: mark $p_1$ as solved, and go to 1.

FirstGrade example:

Marks: FirstGrade, Child, Female, Girl then done!

Observe:

- only letters in KB can be marked, so at most a linear number of iterations
- not goal-directed, so not always desirable
- a similar procedure with better data structures will run in linear time overall

First-order undecidability

Even with just Horn clauses, in the first-order case we still have
the possibility of generating an infinite branch of resolvents.

KB:

$$\text{LessThan}(\text{succ}(x), y) \Rightarrow \text{LessThan}(x, y)$$

Query:

$$\text{LessThan}(\text{zero}, \text{zero})$$

As with full Resolution, there is no way to detect when this will happen

There is no procedure that will test for the satisfiability of first-order Horn clauses
the question is undecidable

As with non-Horn clauses, the best that we can do is to give control of the deduction to the user

to some extent this is what is done in Prolog, but we will see more in “Procedural Control”
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)
  
  ParentOf(x, y) ← MotherOf(x, y)
  ParentOf(x, y) ← FatherOf(x, y)
  ChildOf(x, y) ← ParentOf(y, x)
  AncestorOf(x, y) ← ...

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, z)
   AncestorOf(x, y) ← ParentOf(z, y) ∧ AncestorOf(x, z)

3. AncestorOf(x, y) ← ParentOf(x, z)
   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) ← Fibo(n, y) ∧ Fibo(s(n), z) ∧ Plus(y, z, x)

Requires exponential number of Plus subgoals

Version 2:
Fibo(n, x) ← F(n, 1, 0, x)
F(0, c, p, c)
F(s(n), c, p, x) ← Plus(p, c, s) ∧ F(n, s, c, x)

Requires only linear number of Plus subgoals

Ordering goals

Example:
AmericanCousinOf(x,y) ← American(x) ∧ 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: \( G \cdot G_1, G_2, ..., G_n \) stands for
\( G \leftarrow G_1 \land G_2 \land ... \land G_n \)
but goals are attempted in presented order
Commit

Need to allow for backtracking in goals

\[ \text{AmericanCousinOf}(x,y) \leftarrow \text{CousinOf}(x,y), \text{American}(x) \]

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

But sometimes, given clause of the form

\[ G \leftarrow T, S \]
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: \[ G \leftarrow T_1, T_2, ..., T_m, !, G_1, G_2, ..., G_n \]

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:

\[ G(zero, – ) \leftarrow \text{method 1} \]
\[ G(succ(n), – ) \leftarrow \text{method 2} \]

For example, may split based on computed property:

\[ \text{Expt}(a, n, x) \leftarrow \text{Even}(n), ..., (\text{what to do when } n \text{ is even}) \]
\[ \text{Expt}(a, n, x) \leftarrow \text{Even}(s(n)), ..., (\text{what to do when } n \text{ is odd}) \]

want: check for even numbers only once

Solution: use \(!\) to do if-then-else

\[ G \leftarrow P, \!, Q. \]
\[ G \leftarrow R. \]

To achieve \( G \): if \( P \) then use \( Q \) else use \( R \)

Example:

\[ \text{Expt}(a, n, x) \leftarrow n = 0, \!, x = 1. \]
\[ \text{Expt}(a, n, x) \leftarrow \text{Even}(n), \!, (\text{for even } n) \]
\[ \text{Expt}(a, n, x) \leftarrow (\text{for odd } n) \]

Note: it would be correct to write

\[ \text{Expt}(a, 0, x) \leftarrow \!, x = 1. \]

but not

\[ \text{Expt}(a, 0, 1) \leftarrow \!. \]
**Controlling backtracking**

Consider solving a goal like

1. AncestorOf(jane,billy), Male(jane)
2. ParentOf(jane,billy), Male(jane)
3. Male(jane)
4. ParentOf(z, billy), AncestorOf(jane, z), Male(jane)

FAILS

Eventually FAILS

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

Similarly:

Member(x,l) :- FirstElement(x,l)
Member(x,l) :- Rest(l,l') ∧ 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: \[
\text{not}(G) :: G, !, \text{fail}. \\
/\ * \text{fail if } G \text{ succeeds} * /
\]

\[
\text{not}(G). \\
/\ * \text{otherwise succeed} * /
\]

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 \( \neg \), but not when new variables in \( G \)

\[
[\text{not}(\text{ParentOf}(x,y)) \supset \text{NoChildren}(x)] \checkmark
\]

vs.

\[
[\neg \text{ParentOf}(x,y) \supset \text{NoChildren}(x)] \times
\]
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

\[ \text{ParentOf}(x, y) \iff \text{MotherOf}(x, y) \]

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
   
   \[(\text{Mother susan john}) \ (\text{Person john})\]

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

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

examples:

\[
(\text{proc if-needed} \ \text{(clearatable)}
\begin{align*}
& (\text{for} \ (\text{on x table})
& \quad (\text{and} \ (\text{erase} \ (\text{on x table})) \ (\text{goal} \ \text{(putaway x)})))
& \quad (\text{proc if-removed} \ (\text{on x y}) \ (\text{print} \ x \ \text{" is no longer on "} \ y))
\end{align*}
\]

Shift from proving conditions to making conditions hold!