Logical Inference 2

Rule-based reasoning

Chapter 9

Some material adopted from notes by Andreas Geyer-Schulz,, Chuck Dyer, and Mary Getoor

Automated inference for FOL

- Automated inference for FOL is harder than PL
  - Variables can potentially take on an *infinite* number of possible values from their domains
  - Hence there are potentially an *infinite* number of ways to apply the Universal Elimination rule
- *Godel's Completeness Theorem* says that FOL entailment is only *semi-decidable*
  - If a sentence is **true** given a set of axioms, there is a procedure that will determine this
  - If the sentence is **false**, there's no guarantee a proce-dure will ever determine this — it **may never halt**

Generalized Modus Ponens

- Modus Ponens
  - P, P=>Q |= Q
- Generalized Modus Ponens (GMP) extends this to rules in FOL
- Combines And-Introduction, Universal-Elimination, and Modus Ponens, e.g.
  - *from P(c) and Q(c) and* $\forall x\, P(x) \wedge Q(x) \rightarrow R(x)$ *derive R(c)*
- Must deal with
  - More than one condition on left side of rule
  - variables

Generalized Modus Ponens

- General case: **Given**
  - **atomic sentences** $P_1, P_2, ..., P_N$
  - **implication sentence** $(Q_1 \wedge Q_2 \wedge ... \wedge Q_N) \rightarrow R$
    - $Q_1, ..., Q_N$ and R are atomic sentences
  - **substitution** $subst(\theta, P_i) = subst(\theta, Q_i)$ for i=1,...,N
  - **Derive new sentence: $subst(\theta, R)$**
- Substitutions
  - $subst(\theta, \alpha)$ denotes the result of applying a set of substitutions defined by θ to the sentence α
  - A substitution list $\theta = \{v_1/t_1, v_2/t_2, ..., v_n/t_n\}$ means to replace all occurrences of variable symbol v_i by term t_i
  - Substitutions made in left-to-right order in the list
  - subst({x/Cheese, y/Mickey}, eats(y,x)) = eats(Mickey, Cheese)

Our rules are Horn clauses

- A Horn clause is a sentence of the form:

  $P_1(x) \wedge P_2(x) \wedge \ldots \wedge P_n(x) \rightarrow Q(x)$

where

  - ≥ 0 P_is and 0 or 1 Q
  - P_is and Q are positive (i.e., non-negated) literals

- Equivalently: $P_1(x) \vee P_2(x) \ldots \vee P_n(x)$ where the P_i are all atomic and *at most one* is positive
- Prolog is based on Horn clauses
- Horn clauses represent a *subset* of the set of sentences representable in FOL

Horn clauses II

- Special cases
  - *Typical rule:* $P_1 \wedge P_2 \wedge \ldots P_n \rightarrow Q$
  - *Constraint:* $P_1 \wedge P_2 \wedge \ldots P_n \rightarrow$ false
  - *A fact:* true $\rightarrow$ Q
- These are not Horn clauses:
  - dead(x) ∨ alive(x)
  - married(x, y) → loves(x, y) ∨ hates(x, y)
  - ¬likes(john, mary)
  - ¬likes(x, y) → hates(x, y)
- Can't assert or conclude disjunctions, no negation
- No wonder reasoning over Horn clauses is easier

Horn clauses III

- Where are the quantifiers?
  - Variables in conclusion are universally quantified
  - Variables only in premises are existentially quantified
- Examples:
  - parent(P,X) → isParent(P)
    ∀P ∃X parent(P,X) → isParent(P)
  - parent(P1, X) ∧ parent(X, P2) → grandParent(P1, P2)
    ∀P1,P2 ∃X parent(P1,X) ∧ parent(X, P2) → grandParent(P1, P2)
  - Prolog: grandParent(P1,P2) :- parent(P1,X), parent(X,P2)

Forward & Backward Reasoning

- We usually talk about two reasoning strategies: forward and backward ‘chaining’
- Both are equally powerful
- You can also have a mixed strategy

Forward chaining

- Proofs start with the given axioms/premises in KB, deriving new sentences using GMP until the goal/query sentence is derived
- This defines a **forward-chaining** inference procedure because it moves "forward" from the KB to the goal [eventually]
- Inference using GMP is **sound** and **complete** for KBs containing **only Horn clauses**

Forward chaining algorithm

procedure FORWARD-CHAIN(KB, p)

if there is a sentence in KB that is a renaming of p **then return**
Add p to KB
for each $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$ **in** KB such that for some i, UNIFY$(p_i, p) = \theta$ succeeds **do**
    FIND-AND-INFER(KB, $[p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n], q, \theta$)
end

---

procedure FIND-AND-INFER(KB, *premises*, *conclusion*, θ)

if *premises* $= [\,]$ **then**
    FORWARD-CHAIN(KB, SUBST(θ, *conclusion*))
else for each p' **in** KB such that UNIFY(p', SUBST(θ, FIRST(*premises*))) $= \theta_2$ **do**
    FIND-AND-INFER(KB, REST(*premises*), *conclusion*, COMPOSE(θ, θ_2))
end

Forward chaining example

- KB:
  - allergies(X) → sneeze(X)
  - cat(Y) ∧ allergicToCats(X) → allergies(X)
  - cat(felix)
  - allergicToCats(mary)
- Goal:
  - sneeze(mary)

Backward chaining

- **Backward-chaining** deduction using GMP is **complete** for KBs containing **only Horn clauses**
- Start with goal query, find rules with that conclusion, then prove each rule antecedent
- Keep going until you reach premises
- Avoid loops: check if new subgoal is already on goal stack
- Avoid repeated work: check if new subgoal
  - Has already been proved true
  - Has already failed

Backward chaining algorithm

function BACK-CHAIN(KB, q) **returns** a set of substitutions

  BACK-CHAIN-LIST(KB, $[q]$, $\{\}$)

---

function BACK-CHAIN-LIST(KB, $qlist$, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    $qlist$, a list of conjuncts forming a query (θ already applied)
    θ, the current substitution
  static: $answers$, a set of substitutions, initially empty

  if $qlist$ is empty **then return** $\{\theta\}$
  $q \leftarrow$ FIRST($qlist$)
    for each q'_i **in** KB such that $\theta_i \leftarrow$ UNIFY(q, q'_i) succeeds **do**
      Add COMPOSE(θ, θ_i) to $answers$
    end
    for each sentence ($p_1 \wedge \ldots \wedge p_n \Rightarrow q'_i$) **in** KB such that $\theta_i \leftarrow$ UNIFY(q, q'_i) succeeds **do**
      $answers \leftarrow$ BACK-CHAIN-LIST(KB, SUBST(θ_i, $[p_1 \ldots p_n]$), COMPOSE(θ, θ_i)) $\cup$ $answers$
    end
  return the union of BACK-CHAIN-LIST(KB, REST($qlist$), θ) for each $\theta \in answers$

Backward chaining example

- KB:
  - allergies(X) $\rightarrow$ sneeze(X)
  - cat(Y) $\wedge$ allergicToCats(X) $\rightarrow$ allergies(X)
  - cat(felix)
  - allergicToCats(mary)
- Goal:
  - sneeze(mary)

Forward vs. backward chaining

- Forward chaining is *data-driven*
  - Automatic, unconscious processing, e.g., object recognition, routine decisions
  - May do lots of work that is irrelevant to the goal
  - Efficient when you want to compute all conclusions
- Backward chaining is goal-driven, better for problem-solving and query answering
  - Where are my keys? How do I get to my next class?
  - Complexity of BC can be much less than linear in the size of the KB
  - Efficient when you want one or a few decisions
  - Good where the underlying facts are changing

Mixed strategy

- Many practical reasoning systems do both forward and backward chaining
- The way you encode a rule determines how it is used, as in

```
% this is a forward chaining rule
spouse(X,Y) => spouse(Y,X).
% this is a backward chaining rule
wife(X,Y) <= spouse(X,Y), female(X).
```

- Given a set of rules and the kind of reasoning needed, it’s possible to decide which to encode as FC and which as BC rules.

Completeness of GMP

- GMP (using forward or backward chaining) is complete for KBs that contain only Horn clauses
- ***not* complete** for simple KBs with **non-Horn clauses**
- What is entailed by the following sentences:
  1. $(\forall x)\ P(x) \rightarrow Q(x)$
  2. $(\forall x)\ \neg P(x) \rightarrow R(x)$
  3. $(\forall x)\ Q(x) \rightarrow S(x)$
  4. $(\forall x)\ R(x) \rightarrow S(x)$

Completeness of GMP

- The following entail that S(A) is true:
  1. $(\forall x)\ P(x) \rightarrow Q(x)$
  2. $(\forall x)\ \neg P(x) \rightarrow R(x)$
  3. $(\forall x)\ Q(x) \rightarrow S(x)$
  4. $(\forall x)\ R(x) \rightarrow S(x)$
- If we want to conclude S(A), with GMP we cannot, since the second one is not a Horn clause
- It is equivalent to $P(x) \vee R(x)$

How about in Prolog?

Try encoding this in Prolog

```
1. q(X) :- p(X).
2. r(X) :- neg(p(X)).
3. s(X) :- q(X).
4. s(X) :- r(X).
```

1. $(\forall x)\ P(x) \rightarrow Q(x)$
2. $(\forall x)\ \neg P(x) \rightarrow R(x)$
3. $(\forall x)\ Q(x) \rightarrow S(x)$
4. $(\forall x)\ R(x) \rightarrow S(x)$

- We should not use **\+** or **not** (in SWI) for negation since it means *"negation as failure"*
- Prolog explores possible proofs independently
- It can't take a larger view and realize that one branch must be true since **p(x)** v **~p(x)** is always true