Unification and Lifting

Craig Eminger

Propositional approach = not efficient.

Ex.) \( \forall x \ King(x) \land Greedy(x) \Rightarrow Evil(x) \)

\[ \begin{align*}
  & King(John) \\
  & Greedy(John) \\
  & Brother(Richard, John)
\end{align*} \]

The query is \( Evil(x) \).

Apply **Universal Instantiation** and we get:

\[ \begin{align*}
  & King(John) \land Greedy(John) \Rightarrow Evil(John) \\
  & King(Richard) \land Greedy(Richard) \Rightarrow Evil(Richard)
\end{align*} \]

By applying a complete propositional algorithm (chapter 7), we can determine \( Evil(John) \).
What happens when we add $\textit{Siblings}(\textit{Peter, Sharon})$ to our knowledge base?

\begin{align*}
\forall x \; \textit{King}(x) \land \textit{Greedy}(x) & \Rightarrow \textit{Evil}(x) \\
\textit{King}(\textit{John}) \\
\textit{Greedy}(\textit{John}) \\
\textit{ Brother}(\textit{Richard, John}) \\
\textit{ Siblings}(\textit{Peter, Sharon})
\end{align*}

2 new values in our vocabulary = “Peter” and “Sharon”.

Apply **Universal Instantiation** and we get:

\begin{align*}
\textit{King}(\textit{John}) \land \textit{Greedy}(\textit{John}) & \Rightarrow \textit{Evil}(\textit{John}) \\
\textit{King}(\textit{Richard}) \land \textit{Greedy}(\textit{Richard}) & \Rightarrow \textit{Evil}(\textit{Richard}) \\
\textit{King}(\textit{Peter}) \land \textit{Greedy}(\textit{Peter}) & \Rightarrow \textit{Evil}(\textit{Peter}) \\
\textit{King}(\textit{Sharon}) \land \textit{Greedy}(\textit{Sharon}) & \Rightarrow \textit{Evil}(\textit{Sharon})
\end{align*}

These sentences are not all necessary in our KB. We can teach the computer to make better inferences.
We want an X where X is a King and X is Greedy (then X is evil).

Ideally, we want $\theta = \{\text{substitution set}\}$.

i.e. $\theta = \{x/\text{John}\}$

$\forall y \text{Greedy}(y)$ = “for all values y, y is greedy”

Or basically, “everyone is greedy.”

Now our $\theta = \{x/\text{John}, y/\text{John}\}$, so we can infer Evil($x$)

The inference rule that encapsulates this process called **Generalized Modus Ponens.**

**Generalized Modus Ponens:**

For atomic sentences $p_i, p'_i$, and $q$, where there is a substitution $\theta$ such that $\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p'_i)$, for all $i$,

$p'_1, p'_2, \ldots, p'_n, (p_1 \land p_2 \land \ldots \land p_n \Rightarrow q)$

$\text{SUBST}(\theta, q)$

N + 1 premises = N atomic sentences + one implication.
Applying $\text{SUBST}(\theta, q)$ yields the conclusion we seek.


\[ p_1' = King(John) \quad p_2' = Greedy(y) \]
\[ p_1 = King(x) \quad p_2 = Greedy(x) \]
\[ \theta = \{ x / John, y / John \} \quad q = Evil(x) \]

\[ \text{SUBST}(\theta, q) \text{ is } Evil(John) \]

Generalized Modus Ponens = lifted Modus Ponens

**Lifted** - transformed from:

Propositional Logic → First-order Logic

(Note: Backwards chaining, forwards chaining, and resolution algorithms also have lifted forms you will see later)

---

How do we determine substitution \( \theta \)? \( \rightarrow \) **unification**.

**Unification** = process we use to find substitutions that make different logical expressions look identical

Algorithm: UNIFY(p, q) = \( \theta \) where \( \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q) \)

\( \theta \) is our **unifier** value (if one exists).

Ex.) “Who does John know?”

UNIFY(\( \text{Knows}(John, x), \text{Knows}(John, Jane) \)) = \( \{ x / Jane \} \).

UNIFY(\( \text{Knows}(John, x), \text{Knows}(y, Bill) \)) = \( \{ x/Bill, y/John \} \).

UNIFY(\( \text{Knows}(John, x), \text{Knows}(y, \text{Mother}(y)) \)) = \( \{ x/Bill, y/John \} \).
Can anyone see the problem with the next example?

UNIFY(Knows(John, x), Knows(x, Elizabeth)) = FAIL.

What can we do to fix it?

Both use the same variable, X. X can’t equal both John and Elizabeth.

The solution: change the variable X to Y (or any other value) in Knows(X, Elizabeth)

Knows(X, Elizabeth) → Knows(Y, Elizabeth)

Still means the same.

This is called standardizing apart.
Another problem = sometimes possible for > 1 unifier returned:

\[\text{UNIFY}(\text{Knows}(John, x), \text{Knows}(y, z)) = ???\]

This can return two possible unifications:

\(\{y/John, x/z\}\) which means \(\text{Knows}(John, z)\)

OR

\(\{y/John, x/John, z/John\}\) which means \(\text{Knows}(John, John)\).

For each unifiable pair of expressions there is a single **most general unifier** (MGU). (algorithm on page 278)

**Occur check:** makes sure same variable isn’t used twice
increases time complexity

---

**Storage and Retrieval:**

<table>
<thead>
<tr>
<th>What we do</th>
<th>Function</th>
<th>Primative</th>
</tr>
</thead>
<tbody>
<tr>
<td>Store info in KB</td>
<td>TELL</td>
<td>STORE</td>
</tr>
<tr>
<td>Get info from KB</td>
<td>ASK</td>
<td>FETCH</td>
</tr>
</tbody>
</table>

**Easy way to implement:**

Store all sentences in a long list, browse list one sentence at a time with UNIFY on an ASK query.

Easy way = inefficient.

Any ideas how to improve this?
**Problem:** On a FETCH, you would compare your query sentence with sentences that have no chance of unification.

i.e.) $\text{Knows}(\text{John}, x)$ vs. $\text{Brother}(\text{Richard}, \text{John})$

Not compatible.

**Solution:** categorize sentences w/ **indexing**.

**Predicate indexing** - one method of such

- store facts in “buckets”
- buckets stored in hash table
- accessed by index keys
- best w/ lots of predicate symbols & few clauses for each

More clauses for symbols $\rightarrow$ Multiple index keys

Ex.) $\text{Employs}($nameOfCompany, nameOfEmployee$)$

Given $\text{Employs(AIMA.org, Richard)}$, can answer:

- $\text{Employs(AIMA.org, Richard)}$ Does AIMA.org employ Richard?
- $\text{Employs(x, Richard)}$ Who employs Richard?
- $\text{Employs(AIMA.org, y)}$ Whom does AIMA.org employ?
- $\text{Employs(x, y)}$ Who employs whom?

We can arrange this into a **subsumption lattice**.

(the lattice for this example is shown on page 280)
A subsumption lattice has the following properties:

• child of any node obtained from its parents by one substitution

• the “highest” common descendant of any two nodes is the result of applying their most general unifier

• predicate with \( n \) arguments contains \( O(2^n) \) nodes (in our example, we have two arguments, so our lattice has four nodes)

• repeated constants = slightly different lattice [see Lattice b]

Adding function symbols:

• creates new lattice structures

• lattice gains in complexity (size increases exponentially)

• indexing benefit lost by a > cost for storing/maintaining indices