Chapter 9. Inference in First-Order Logic

Function \text{UNIFY}(x, y, 9) returns a substitution to make \( a \) and \( y \) identical inputs: \( a \), a variable, constant, list, or compound expression \( y \), a variable, constant, list, or compound expression \( 9 \), the substitution built up so far (optional, defaults to empty)

\[
\text{if } 0 = \text{failure} \text{ then return failure} \\
\text{else if } a = y \text{ then return } 0 \\
\text{else if VARIABLE}(x) \text{ then return UNIFY-VAR}(x, y, 9) \\
\text{else if COMPOUND}(x) \text{ and COMPOUND}(y) \text{ then} \\
\quad \text{return UNIFY}(x.\text{ARGS}, y.\text{ARGS}, \text{UNIFY}(x.\text{OP}, y.\text{OP}, 9)) \\
\text{else if LIST}(x) \text{ and LIST}(y) \text{ then} \\
\quad \text{return UNIFY}(x.\text{REST}, y.\text{REST}, \text{UNIFY}(x.\text{FIRST}, y.\text{FIRST}, 9)) \\
\text{else return failure}
\]

Function \text{UNIFY-VAR}(\text{var}, 2, 0) returns a substitution

\[
\text{if } \{\text{var}/\text{val}\} F \in \theta \text{ then return } \text{UNIFY}(\text{var}, \text{val}, 0) \\
\text{else if OCCUR-CHECK}(\text{var}, \theta) \text{ then return failure} \\
\text{else return add \text{var} to } \theta
\]

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution \( \theta \) that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as \( P(A, B) \), the \text{OP} field picks out the function symbol \( F \) and the ARCS field picks out the argument list \( (A, B) \).

The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works, and it’s all you need to understand the rest of the chapter. The remainder of this section outlines ways to make retrieval more efficient; it can be skipped on first reading.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have some chance of unifying. For example, there is no point in trying to unify \text{Knows}(\text{John}, x) with \text{Brother}(\text{Richard}, \text{John}). We can avoid such unifications by indexing the facts in the knowledge base. A simple scheme called predicate indexing puts all the \text{Knows} facts in one bucket and all the \text{Brother} facts in another. The buckets can be stored in a hash table for efficient access.

Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol. Sometimes, however, a predicate has many clauses. For example, suppose that the tax authorities want to keep track of who employs whom, using a predicate \text{Employed}(x, y). This would be a very large bucket with perhaps millions of employers...
and tens of millions of employees. Answering a query such as \( \text{Employs}(x, \text{Richard}) \) with predicate indexing would require scanning the entire bucket. 

For this particular query, it would help if facts were indexed both by predicate and by second argument, perhaps using a combined hash table key. Then we could simply construct the key from the query and retrieve exactly those facts that unify with the query. For other queries, such as \( \text{Employs}(\text{IBM}, y) \), we would need to have indexed the facts by combining the predicate with the first argument. Therefore, facts can be stored under multiple index keys, rendering them instantly accessible to various queries that they might unify with.

Given a sentence to be stored, it is possible to construct indices for all possible queries that unify with it. For the fact \( \text{Employs}(\text{IBM}, \text{Richard}) \), the queries are:

- \( \text{Employs}(\text{IBM}, \text{Richard}) \)  Does IBM employ Richard?
- \( \text{Employs}(a, \text{Richard}) \)  Who employs Richard?
- \( \text{Employs}(\text{IBM}, y) \)  Whom does IBM employ?
- \( \text{Employs}(x, y) \)  Who employs whom?

These queries form a **subsumption lattice**, as shown in Figure 9.2(a). The lattice has some interesting properties. For example, the child of any node in the lattice is obtained from its parent by a single substitution; and the "highest" common descendant of any two nodes is the result of applying their most general unifier. The portion of the lattice above any ground fact can be constructed systematically (Exercise 9.5). A sentence with repeated constants has a slightly different lattice, as shown in Figure 9.2(b). Function symbols and variables in the sentences to be stored introduce still more interesting lattice structures.

The scheme we have described works very well whenever the lattice contains a small number of nodes. For a predicate with two arguments, however, the lattice contains \( O(2^n) \) nodes. If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices. At some point, the benefits of indexing are outweighed by the costs of storing and maintaining all the indices. We can respond by adopting a fixed policy, such as maintaining indices only on keys composed of a predicate plus each argument, or by using an adaptive policy that creates indices to meet the demands of the kinds of queries being asked. For most AI systems, the number of facts to be stored is small enough that efficient indexing is considered a solved problem. For commercial databases, where facts number in the billions, the problem has been the subject of intensive study and technology development.