such that $\text{SUBST}(\theta_i) = \text{SUBST}(\theta, p_i)$, for all $i$,

$$p_1' \land \ldots \land p_n' \land (p_1') \land \ldots \land (p_n') = q$$

There are $n+1$ premises to this rule: the $n$ atomic sentences $p_i$ and the one implication. The conclusion is the result of applying the substitution $\theta$ to the consequent $q$. For our example:

- $p_1'$ is $\text{King}(\text{John})$
- $p_2'$ is $\text{Greedy}(y)$
- $p_3$ is $\{x/\text{John}, y/\text{John}\}$
- $\theta$ is $\{q/\text{Evil}(\text{John})\}$

It is easy to show that Generalized Modus Ponens is a sound inference rule. First, we observe that, for any sentence $p$ (whose variables are assumed to be universally quantified) and for any substitution $\theta$, $\text{SUBST}(\theta, p)$ holds by Universal Instantiation. It holds in particular for a $\theta$ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1', \ldots, p_n'$ we can infer

$$\text{SUBST}(\theta, p_1) \land \ldots \land \text{SUBST}(\theta, p_n')$$

and from the implication $p_1 \land \ldots \land p_n \land q$ we can infer

$$\text{SUBST}(\theta, p_1) \land \ldots \land \text{SUBST}(\theta, p_n) = \text{SUBST}(\theta, q).$$

Now, $E$ in Generalized Modus Ponens is defined so that $\text{SUBST}(\theta, p_i) = \text{SUBST}(\theta, p_i')$, for all $i$; therefore the first of these two sentences matches the premise of the second exactly, Hence, $\text{SUBST}(\theta, q)$ follows by Modus Ponens.

Generalized Modus Ponens is a lifted version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic. We will see in the rest of this chapter that we can develop lifted versions of the forward chaining, backward chaining, and resolution algorithms introduced in Chapter 7. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions that are required to allow particular inferences to proceed.

### 9.2.2 Unification

Lifed inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them if one exists:

$$\text{UNIFY}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

Let us look at some examples of how UNIFY should behave. Suppose we have a query $\text{AskVars}(\text{Knows}(\text{John}, x))$: whom does John know? Answers to this query can be found
by finding all sentences in the knowledge base that unify with Knows(John, x). Here are the results of unification with four different sentences that might be in the knowledge base:

UNIFY(Knows(John, x)Knows(John, Jane)) = {x \mapsto Jane}

UNIFY(Knows(John, x)Knows(y, Bill)) = {x \mapsto Bill, y \mapsto John}

UNIFY(Knows(John, x)Knows(y, Mother(y))) = {y \mapsto John, x \mapsto Mother(John)}

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

The last unification fails because x cannot take on the values John and Elizabeth at the same time. Now, remember that Knows(x, Elizabeth) means "Everyone knows Elizabeth," so we should be able to infer that John knows Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by standardizing apart one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in Knows(x, Elizabeth) to $x_{17}$ (a new variable name) without changing its meaning. Now the unification will work:

UNIFY(Knows(John, x), Knows($x_{17}$, Elizabeth)) = {Elizabeth \mapsto x_{17}, x \mapsto John}.

Exercise 9.12 delves further into the need for standardizing apart.

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier, For example, UNIFY(Knows(John, x), Knows(y, z)) could return \{y \mapsto John, x \mapsto 2\} or \{y \mapsto John, x \mapsto John\}. The first unifier gives Knows(John, 2) as the result of unification, whereas the second gives Knows(John, John). The second result could be obtained from the first by an additional substitution \{2 \mapsto John\}; we say that the first unifier is more general than the second, because it places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single most general unifier (or MGU) that is unique up to renaming and substitution of variables. (For example, \{x \mapsto John\} and \{y \mapsto John\} are considered equivalent, as are \{x \mapsto John, y \mapsto John\} and \{a \mapsto John, y \mapsto x\}.) In this case it is \{y \mapsto John, x \mapsto 2\}.

An algorithm for computing most general unifiers is shown in Figure 9.1. The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, S(x) can’t unify with S(S(x)). This so-called occur check makes the complexity of the entire algorithm quadratic in the size of the expressions being unified. Some systems, including all logic programming systems, simply omit the occur check and sometimes make unsound inferences as a result; other systems use more complex algorithms with linear-time complexity.

9.2.3 Storage and retrieval

Underlying the TELL and ASK functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some