Chapter 15

Query-answering in Deductive Databases

One of the great virtues of logic programming is that programs have a declarative semantics which can be understood independently of any particular operational semantics. SLD-resolution is one example of a class of interpreters that can be used to compute the logical consequences of a definite program. But also other strategies can be used. In fact, SLD-resolution has several weaknesses also in the case of an ideal interpreter. For instance, consider the following definition of the Fibonacci numbers (for convenience $X + n$ and n abbreviate the terms $s^n(X)$ and $s^n(0)$):

$$\begin{array}{l} \mathit{fib}(0, 1). \\ \mathit{fib}(1, 1). \\ \mathit{fib}(X + 2, Y) \leftarrow \mathit{fib}(X + 1, Z), \mathit{fib}(X, W), \mathit{add}(Z, W, Y). \end{array}$$

Now assume that the following goal clause is given:

$$\leftarrow \mathit{fib}(10, X).$$

The goal reduces to the following:

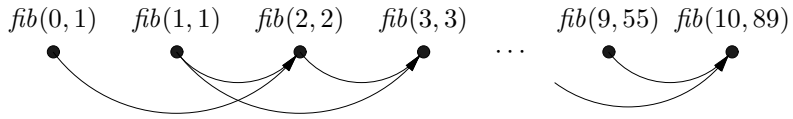
$$\leftarrow \mathit{fib}(9, Z_0), \mathit{fib}(8, W_0), \mathit{add}(Z_0, W_0, X).$$

And selection of the leftmost subgoal yields:

$$\leftarrow \mathit{fib}(8, Z_1), \mathit{fib}(7, W_1), \mathit{add}(Z_1, W_1, Z_0), \mathit{fib}(8, W_0), \mathit{add}(Z_0, W_0, X).$$

This goal contains two subgoals which are identical up to variable renaming: $\mathit{fib}(8, Z_1)$ and $\mathit{fib}(8, W_0)$. In order to resolve the whole goal, both subgoals have to be resolved leading to duplicate work. As a matter of fact, the number of recursive calls to $\mathit{fib}/2$ grows exponentially with the size of the input.

In this particular case it is better to compute answers to $\leftarrow fib(10, X)$ starting from the base cases: Since $fib(0, 1)$ and $fib(1, 1)$ it must hold that $fib(2, 2)$ and so forth:



Another problem with SLD-resolution has to do with termination. Even when no function symbols are involved and there are only a finite number of answers to a goal, SLD-resolution may loop. Consider the goal $\leftarrow married(X, Y)$ and the program:

$$married(X, Y) \leftarrow married(Y, X).$$
$$married(adam, anne).$$

There are only two answers to the goal. However, the SLD-tree is infinite and Prolog would not even find the answers unless the clauses were swapped.

In Chapter 6 logic programming was advocated as a representation language for relational databases. But as illustrated above, SLD-resolution is not always the best mechanism for a query-answering system. In a database system it is particularly important to guarantee termination of the query-answering process whenever that is possible. This chapter considers an alternative inference mechanism for logic programs which has a much improved termination behaviour than SLD-resolution. It may also avoid unnecessary recomputations such as those above. Some assumptions which are often made in the deductive database literature are consciously avoided in our exposition. (In particular, the division of the program into an extensional and intensional part.)

15.1 Naive Evaluation

SLD-resolution is goal-directed — the starting point of the computation is the goal and the aim of the reasoning process is to derive a contradiction by rewritings of the goal. This has several advantages:

- The tree-like structure of the search space lends itself to efficient implementations, both space- and time-wise;
- By focusing attention on the goal it is possible to avoid some inferences which are of no importance for the answers to the goal.

A completely different approach is to start from what is known to be true — the facts and the rules of the program — and to (blindly) generate consequences of the program until the goal can be refuted. (For the sake of simplicity, it will be assumed that all goals are of the form $\leftarrow A$.) For instance, let P be the program:

$$married(X, Y) \leftarrow married(Y, X).$$
$$married(adam, anne).$$

Clearly, $P \models married(adam, anne)$. Moreover, since $married(adam, anne)$ is true in any model of the program it must hold that $P \models married(anne, adam)$. This idea resembles the immediate consequence operator originally introduced in Chapter 3:

```
fun naive(P)
begin
  x := facts(P);
  repeat
    y := x;
    x := S_P(y);
  until x = y;
  return x;
end
```

Figure 15.1: Naive evaluation

$$T_P(I) = \{A_0 \mid A_0 \leftarrow A_1, \ldots, A_n \in \mathit{ground}(P) \text{ and } A_1, \ldots, A_n \in I\}$$

Recall that the least fixed point of this operator (which can be obtained as the limit of $T_P \uparrow n$ where $n \geq 0$) characterizes the set of all *ground* atomic consequences of the program. Hence, the T_P-operator *can* be used for query-answering. However, for computational reasons it is often more practical to represent Herbrand interpretations by sets of atoms (ground or non-ground). A "non-ground T_P-operator" may be defined as follows:[1]

$$S_P(I) = \{A_0\theta \mid A_0 \leftarrow A_1, \ldots, A_n \in P \text{ and } \theta \in \mathit{solve}((A_1, \ldots, A_n), I)\}$$

where $\theta \in \mathit{solve}((A_1, \ldots, A_n), I)$ if θ is an mgu of $\{A_1 \doteq B_1, \ldots, A_n \doteq B_n\}$ and $B_1, \ldots, B_n$ are members in I renamed apart form each other and $A_0 \leftarrow A_1, \ldots, A_n$.

It can be shown that S_P is closed under logical consequence. That is, if every atom in I is a logical consequence of P then so is $S_P(I)$. In particular, every atom in $\varnothing$ is clearly a logical consequence of P. Thus, every atom in $S_P(\varnothing)$ is a logical consequence of P. Consequently every atom in $S_P(S_P(\varnothing))$ is a logical consequence of P and so forth. This iteration — which may be denoted:

$$S_P \uparrow 0, \quad S_P \uparrow 1, \quad S_P \uparrow 2, \quad \ldots$$

yields larger and larger sets of atomic consequences. By analogy to the immediate consequence operator it can be shown that there exists a least set I of atomic formulas such that $S_P(I) = I$ and that this set equals the limit of $S_P \uparrow n$. An algorithmic formulation of this iteration can be found in Figure 15.1. (Let $\mathit{facts}(P)$ denote the set of all facts in P.) The algorithm is often referred to as *naive evaluation*.

The result of the naive evaluation can be used to answer queries to the program: If B is an atom in $\mathit{naive}(P)$ renamed apart from A and θ is an mgu of $A \doteq B$. Then θ is an answer to the goal $\leftarrow A$.

Example 15.1 Consider the following transitive closure program:

[1] For the sake of simplicity it is assumed that $S_P(I)$ never contains two atoms which are renamings of each other.

```
fun semi-naive(P)
begin
  Δx := facts(P);
  x := Δx;
  repeat
    Δx := ΔS_P(x, Δx);
    x := x ∪ Δx;
  until Δx = ∅;
  return x;
end
```

Figure 15.2: Semi-naive evaluation

$$path(X,Y) \leftarrow edge(X,Y).$$
$$path(X,Y) \leftarrow path(X,Z), edge(Z,Y).$$

$$edge(a,b).$$
$$edge(b,a).$$

Let x_i denote the value of x after i iterations. Then the iteration of the naive-evaluation algorithm looks as follows:

$$\begin{aligned}
x_0 &= \{edge(a,b), edge(b,a)\} \\
x_1 &= \{edge(a,b), edge(b,a), path(a,b), path(b,a)\} \\
x_2 &= \{edge(a,b), edge(b,a), path(a,b), path(b,a), path(a,a), path(b,b)\} \\
x_3 &= \{edge(a,b), edge(b,a), path(a,b), path(b,a), path(a,a), path(b,b)\}
\end{aligned}$$

The goal $\leftarrow path(a,X)$ has two answers: $\{X/a\}$ and $\{X/b\}$. ∎

15.2 Semi-naive Evaluation

As suggested by the name, naive evaluation can be improved in several respects. In particular, each iteration of the algorithm recomputes everything that was computed in the previous iteration. That is $x_i \subseteq x_{i+1}$. The revised algorithm in Figure 15.2 avoids this by keeping track of the difference $x_i \setminus x_{i-1}$ in the auxiliary variable Δx. Note first that the loop of the naive evaluation may be replaced by:

```
repeat
  Δx := S_P(x) \ x;
  x := x ∪ Δx;
until Δx = ∅;
```

The expensive operations here is $S_P(x) \setminus x$ which renders the modified algorithm even more inefficient than the original one. However, the new auxiliary function call $\Delta S_P(x, \Delta x)$ computes the difference more efficiently:

$$\Delta S_P(I, \Delta I) = \{A_0\theta \notin I \mid \begin{array}{l} A_0 \leftarrow A_1, \ldots, A_n \in P \text{ and} \\ \theta \in \mathit{solve}((A_1, \ldots, A_n), I, \Delta I)\} \end{array}$$

where $\theta \in \mathit{solve}((A_1, \ldots, A_n), I, \Delta I)$ if θ is an mgu of $\{A_1 \doteq B_1, \ldots, A_n \doteq B_n\}$ and $B_1, \ldots, B_n$ are atoms in I renamed apart form each other and $A_0 \leftarrow A_1, \ldots, A_n$ and at least one $B_i \in \Delta I$.

It can be shown that the algorithm in Figure 15.2 — usually called *semi-naive evaluation* — is equivalent to naive evaluation:

Theorem 15.2 (Correctness of semi-naive evaluation) Let P be a definite program, then $\mathit{naive}(P) = \mathit{semi\text{-}naive}(P)$. ∎

Example 15.3 The following is a trace of the semi-naive evaluation of the programs in Example 15.1:

$$\begin{array}{lcl} \Delta x_0 & = & \{\mathit{edge}(a,b), \mathit{edge}(b,a)\} \\ \Delta x_1 & = & \{\mathit{path}(a,b), \mathit{path}(b,a)\} \\ \Delta x_2 & = & \{\mathit{path}(a,a), \mathit{path}(b,b)\} \\ \Delta x_3 & = & \varnothing \end{array}$$

∎

The main advantage of the naive and semi-naive approach compared to SLD-resolution is that they terminate for some programs where SLD-resolution loops. In particular when no function symbols are involved (i.e. datalog programs). For instance, the goal $\leftarrow \mathit{path}(a, X)$ loops under SLD-resolution. On the other hand, there are also examples where the (semi-) naive approach loops and SLD-resolution terminates. For instance, consider the goal $\leftarrow \mathit{fib}(5, X)$ and the following program (extended with the appropriate definition of $\mathit{add}/3$):

$$\begin{array}{l} \mathit{fib}(0,1). \\ \mathit{fib}(1,1). \\ \mathit{fib}(X+2, Y) \leftarrow \mathit{fib}(X+1, Z), \mathit{fib}(X, W), \mathit{add}(Z, W, Y). \end{array}$$

The SLD-derivation terminates but both the naive and the semi-naive evaluation loop. The reason is that both naive and semi-naive evaluation blindly generate consequences without taking the goal into account. However, the fact $\mathit{fib}(5, 8)$ is obtained early on in the iteration. (In fact, if it was not for the addition it would be computed in the fourth iteration.)

Both naive and semi-naive evaluation also lend themselves to *set-oriented operations* in contrast to SLD-resolution which uses a tuple-at-a-time strategy. The set-oriented approach is often advantageous in database applications where data may reside on secondary storage and the number of disk accesses must be minimized.

15.3 Magic Transformation

This section presents a query-answering approach which combines the advantages of semi-naive evaluation with goal-directedness. The approach amounts to transforming

the program P and a goal $\leftarrow A$ into a new program $magic(P \cup \{\leftarrow A\})$ which may be executed by the naive or semi-naive algorithm.

One of the problems with the semi-naive evaluation is that it blindly generates consequences which are not always needed to answer a specific query. This can be repaired by inserting a "filter" (an extra condition) into the body of each program clause $A_0 \leftarrow A_1, \ldots, A_n$ so that (an instance of) A_0 is a consequence of the program only if it is *needed* in order to compute an answer to a specific atomic goal.

For the purpose of defining such filters, the alphabet of predicate symbols is extended with one new predicate symbol *call_p* for each original predicate symbol p. If A is of the form $p(t_1, \ldots, t_n)$ then $call(A)$ will be used to denote the atom $call\_p(t_1, \ldots, t_n)$. Such an atom is called a *magic template*. The basic transformation scheme may be formulated as follows:

Definition 15.4 (Magic transformation) Let $magic(P)$ be the smallest program such that if $A_0 \leftarrow A_1, \ldots, A_n \in P$ then:

- $A_0 \leftarrow call(A_0), A_1, \ldots, A_n \in magic(P)$;
- $call(A_i) \leftarrow call(A_0), A_1, \ldots, A_{i-1} \in magic(P)$ for each $1 \leq i \leq n$.

∎

Given an initial goal $\leftarrow A$ a transformed clause of the form:

$$A_0 \leftarrow call(A_0), A_1, \ldots, A_n$$

can be interpreted as follows:

A_0 is true if A_0 is needed (to answer $\leftarrow A$) and $A_1, \ldots, A_n$ are true.

The statement "... is needed (to answer $\leftarrow A$)" can also be read as "... is called (in a goal-directed computation of $\leftarrow A$)". Similarly a clause of the form:

$$call(A_i) \leftarrow call(A_0), A_1, \ldots, A_{i-1}$$

can then be understood as follows:

A_i is called if A_0 is called and $A_1, \ldots, A_{i-1}$ are true.

Hence, the first clause extends each clause of the original program with a filter as described above and the second clause defines when a filter is true. The magic transformation can be said to encode a top-down computation with Prolog's computation rule. In fact, as will be illustrated below there is a close correspondence between the semi-naive evaluation of the magic program and the SLD-derivations of the original program.

Example 15.5 Let P be the program in Example 15.1. Then $magic(P)$ is the following program:

$$\begin{array}{l} path(X,Y) \leftarrow call\_path(X,Y), edge(X,Y). \\ path(X,Y) \leftarrow call\_path(X,Y), path(X,Z), edge(Z,Y). \\ edge(a,b) \leftarrow call\_edge(a,b). \\ edge(b,a) \leftarrow call\_edge(b,a). \\ \\ call\_edge(X,Y) \leftarrow call\_path(X,Y). \\ call\_path(X,Z) \leftarrow call\_path(X,Y). \\ call\_edge(Z,Y) \leftarrow call\_path(X,Y), path(X,Z). \end{array}$$

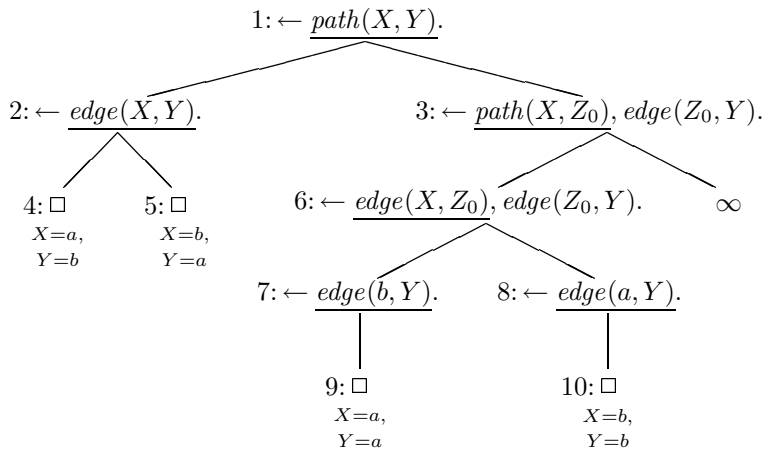


Figure 15.3: SLD-tree of $\leftarrow path(X, Y)$

For instance, note that the last clause may be read: "$edge(Z, Y)$ is called if $path(X, Y)$ is called and $path(X, Z)$ is true". Now compare this with the recursive clause of the original program in Example 15.1! ∎

Note that the program in the example does not contain any facts. Hence, no atomic formula can be a logical consequence of the program. In order to be able to use the magic program for answering a query the program has to be extended with such a fact. More precisely, in order to answer an atomic goal $\leftarrow A$ the transformed program must be extended with the fact $call(A)$. The fact may be read "A is called".

Example 15.6 Consider a goal $\leftarrow path(X, Y)$ to the program in Example 15.1. The semi-naive evaluation of the transformed program looks as follows:

$$\begin{array}{lcl}
\Delta x_0 & = & \{call_path(X, Y)\} \\
\Delta x_1 & = & \{call_edge(X, Y)\} \\
\Delta x_2 & = & \{edge(a, b), edge(b, a)\} \\
\Delta x_3 & = & \{path(a, b), path(b, a)\} \\
\Delta x_4 & = & \{call_edge(a, Y), call_edge(b, Y), path(a, a), path(b, b)\} \\
\Delta x_5 & = & \varnothing
\end{array}$$

Hence, the evaluation terminates and produces the expected answers: $path(a, a)$, $path(a, b)$, $path(b, a)$ and $path(b, b)$. ∎

It is interesting to compare the semi-naive evaluation of the magic program with the SLD-tree of the goal $\leftarrow path(X, Y)$ with respect to the original program.

The selected subgoal in the root of the SLD-tree in Figure 15.3 is $path(X, Y)$. Conceptually this amounts to a call to the procedure $path/2$. In the magic computation this corresponds to the state before the first iteration. The first iteration generates the fact $call\_edge(X, Y)$ which corresponds to the selection of the subgoal $edge(X, Y)$ in node 2 of the SLD-tree. Simultaneously, $path(X, Z_0)$ is selected in node 3. However,

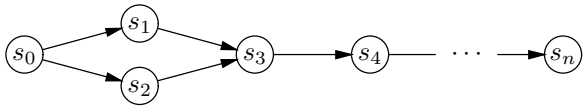


Figure 15.4: Duplicate paths

this is not explicitly visible in Δx_1 since *call_path*(X, Y) and *call_path*(X, Z_0) are renamings of each other. Iteration two yields two answers to the call to *edge*(X, Y); namely *edge*(a, b) and *edge*(b, a) corresponding to nodes 4 and 5 in the SLD-tree. These nodes also provide answers to the call *path*(X, Y) (and *path*(X, Z_0)) and correspond to the result of iteration three, and so forth.

The magic approach can be shown to be both sound and complete:

Theorem 15.7 (Soundness of magic) Let P be a definite program and $\leftarrow A$ an atomic goal. If $A\theta \in \mathit{naive}(\mathit{magic}(P \cup \{\leftarrow A\}))$ then $P \models \forall(A\theta)$. ∎

Theorem 15.8 (Completeness of magic) Let P be a definite program and $\leftarrow A$ an atomic goal. If $P \models \forall(A\theta)$ then there exists $A\sigma \in \mathit{naive}(\mathit{magic}(P \cup \{\leftarrow A\}))$ such that $A\theta$ is an instance of $A\sigma$. ∎

The magic approach combines advantages of naive (and semi-naive) evaluation with goal-directedness. In particular, it has a much improved termination behaviour over both SLD-resolution and naive (and semi-naive) evaluation of the original program P:

- If the SLD-tree of $\leftarrow A$ is finite then *naive*(*magic*$(P \cup \{\leftarrow A\}))$ terminates;
- If *naive*(P) terminates, then *naive*(*magic*$(P \cup \{\leftarrow A\}))$ terminates;

Moreover, the magic approach sometimes avoids repeating computations. Consider the following program and graph in Figure 15.4:

$$\mathit{path}(X, Y) \leftarrow \mathit{edge}(X, Y).$$
$$\mathit{path}(X, Y) \leftarrow \mathit{edge}(X, Z), \mathit{path}(Z, Y).$$

Even if the SLD-tree of $\leftarrow \mathit{path}(s_0, X)$ is finite the tree contains two branches which are identical up to variable renaming — one that computes all paths from s_3 via s_0 and s_1 and one branch that computes all paths from s_3 via s_0 and s_2. By using semi-naive evaluation of the transformed program this is avoided since the algorithm computes a *set* of magic templates and answers to the templates.

15.4 Optimizations

The magic transformation described in the previous section may be modified in various ways to optimize the query-answering process. We give here a brief account of some potential optimizations without going too much into technical detail.

Supplementary magic

Each iteration of the naive evaluation amounts to computing:

$$solve((A_1, \ldots, A_n), I)$$

for each program clause $A_0 \leftarrow A_1, \ldots, A_n$. This in turn amounts to finding an mgu of sets of equations of the form:

$$\{A_1 \doteq B_1, \ldots, A_n \doteq B_n\}$$

If the program contains several clauses with common subgoals it means that the same unification steps are repeated in each iteration of the evaluation. (The same can be said about semi-naive evaluation.) This is a crucial observation for evaluation of magic programs as the magic transformation of a clause $A_0 \leftarrow A_1, \ldots, A_n$ gives rise to the following sub-program:

$$\begin{array}{rcl}
call(A_1) & \leftarrow & call(A_0). \\
call(A_2) & \leftarrow & call(A_0), A_1. \\
call(A_3) & \leftarrow & call(A_0), A_1, A_2. \\
 & \vdots & \\
call(A_n) & \leftarrow & call(A_0), A_1, A_2, \ldots, A_{n-1}. \\
A_0 & \leftarrow & call(A_0), A_1, A_2, \ldots, A_{n-1}, A_n.
\end{array}$$

Hence, naive and semi-naive evaluation run the risk of having to repeat a great many unification steps in each iteration of the evaluation.

It is possible to *factor* out the common subgoals using so-called *supplementary* predicates $\triangledown_0, \ldots, \triangledown_n$; Let $\mathbf{X}$ be the sequence of all variables in the original clause and let the supplementary predicates be defined as follows:

$$\begin{array}{rcl}
\triangledown_0(\mathbf{X}) & \leftarrow & call(A_0) \\
\triangledown_1(\mathbf{X}) & \leftarrow & \triangledown_0(\mathbf{X}), A_1 \\
 & \vdots & \\
\triangledown_n(\mathbf{X}) & \leftarrow & \triangledown_{n-1}(\mathbf{X}), A_n
\end{array}$$

Intuitively, $\triangledown_i(\mathbf{X})$ describes the state of the computation in a goal-directed computation of $A_0 \leftarrow A_1, \ldots, A_n$ after the success of A_i (or before A_1 if $i = 0$). For instance, the last clause states that "if $\triangledown_{n-1}(\mathbf{X})$ is the state of the computation before calling A_n and A_n succeeds then $\triangledown_n(\mathbf{X})$ is the state of the computation after A_n".

Using supplementary predicates the magic transformation may be reformulated as follows:

$$\begin{array}{rcll}
call(A_{i+1}) & \leftarrow & \triangledown_i(\mathbf{X}). & (0 \leq i < n) \\
A_0 & \leftarrow & \triangledown_n(\mathbf{X}). &
\end{array}$$

The transformation increases the number of clauses in the transformed program. It also increases the number of iterations in the evaluation. However, the amount of work in each iteration decreases dramatically since the clause bodies are shorter and avoids much of the redundancy due to duplicate unifications.

Note that no clause in the new sub-program contains more than two body literals. In fact, the supplementary transformation is very similar in spirit to Chomsky Normal Form used to transform context-free grammars (see Hopcroft and Ullman (1979)).

Subsumption

In general we are only interested in "most general answers" to a goal. That is to say, if the answer A_1 is a special case of A_2 then we are only interested in A_2. (This is why the definition of SLD-resolution involves only most general unifiers.) In a naive or semi-naive evaluation it may happen that the set being computed contains two atoms, A_1 and A_2, where A_1 is an instance of A_2 (i.e. there is a substitution θ such that $A_1 = A_2\theta$). Then A_1 is said to be *subsumed* by A_2. In this case A_1 is redundant and may be removed from the set without sacrificing completeness of the query-answering process. Moreover, keeping the set as small as possible also improves performance of the algorithms. In the worst case, redundancy due to subsumption may propagate leading to an explosion in the size of the set.

From a theoretical perspective it is easy to extend both naive and semi-naive evaluation with a normalization procedure which removes redundancy from the set. However, checking for subsumption may be so expensive from the computational point of view (and is so rarely needed), that it is often not used in practice.

Sideways information passing

As commented the magic transformation presented in Definition 15.4 encodes a goal-directed computation where the subgoals are solved in the left to right order. Hence, given a clause:

$$A_0 \leftarrow A_1, \ldots, A_n$$

the transformed program contains a clause:

$$call(A_i) \leftarrow call(A_0), A_1, \ldots, A_{i-1}$$

In addition to imposing an ordering on the body atoms, the clause also propagates bindings of $call(A_0)$ and $A_1, \ldots, A_{i-1}$ to $call(A_i)$. Now two objections may be raised:

- The left to right goal ordering is not necessarily the most efficient way of answering a query;
- Some of the bindings may be of no use for $call(A_i)$. And even if they are of use, we may not necessarily want to propagate all bindings to it.

Consider the following sub-program which checks if two nodes (e.g. in a tree) are on the *same depth*. That is, if they have a common ancestor the same number of generations back.

$$sd(X, X).$$
$$sd(X, Y) \leftarrow child(X, Z), child(Y, W), sd(Z, W).$$

Note that there is no direct flow of bindings between $child(X, Z)$ and $child(Y, W)$ in a goal-directed computation. Hence the two subgoals may be solved in parallel. However, the recursive call $sd(X, Z)$ relies on bindings from the previous two, and should probably await the success of the other subgoals. Now, the magic transformation imposes a linear ordering on the subgoals by generating:

$$call\_child(X, Z) \leftarrow call\_sd(X, Y).$$
$$call\_child(Y, W) \leftarrow call\_sd(X, Y), child(X, Z).$$
$$call\_sd(Z, W) \leftarrow call\_sd(X, Y), child(X, Z), child(Y, W).$$

In this particular case it would probably be more efficient to emit the clauses:

$$call\_child(X, Z) \leftarrow call\_sd(X, Y).$$
$$call\_child(Y, W) \leftarrow call\_sd(X, Y).$$
$$call\_sd(Z, W) \leftarrow call\_sd(X, Y), child(X, Z), child(Y, W).$$

Intuitively this means that the two calls to $child/2$ are carried out "in parallel" as soon as $sd/2$ is called. The recursive call, on the other hand, goes ahead only if the two calls to $child/2$ succeed.

This example illustrates that there are variations of the magic transformation which potentially yield more efficient programs. However, in order to exploit such variations the transformation must be parameterized by the strategy for solving the subgoals of each clause. Which strategy to use relies on the flow of data between atoms in the program and may require global analysis of the program. Such strategies are commonly called sip's (sideways information passing strategies) and the problem of generating efficient strategies is an active area of research (see Ullman (1989) for further reading).

Exercises

15.1 Transform the following program using Definition 15.4:

$$expr(X, Z) \leftarrow expr(X, [+|Y]), expr(Y, Z).$$
$$expr([id|Y], Y).$$

Then use naive and semi-naive evaluation to "compute" answers to the goal:

$$\leftarrow expr([id, +, id], X).$$

What happens if the goal is evaluated using SLD-resolution and the original program?

15.2 Consider the program $sd/2$ on p. 238 and the following "family tree":

$$child(b, a). \quad child(c, a). \quad child(d, b). \quad child(e, b). \quad child(f, c).$$
$$child(g, d). \quad child(h, d). \quad child(i, e). \quad child(j, f). \quad child(k, f).$$

Transform the program using (a) magic templates (b) supplementary magic. Then compute the answers to the goal:

$$\leftarrow sd(d, X)$$