17 Logic Programming

Logic programming is based on the observation that formulas in mathematical logic can be interpreted as specifications of computations. The style of programming is declarative rather than imperative. We do not issue commands telling the computer what to do; instead, we describe the relationship between the input data and the output data, and let the computer figure out how to obtain the output from the input. To the extent that this is successful, logic programming provides a significantly higher level of abstraction, with the corresponding advantage of extremely concise programs.

There are two major abstractions that characterize logic programming. The first is that control statements such as our familiar for- and if-statements are abstracted away. Instead, the "compiler" supplies an extremely powerful control mechanism that is *uniformly* applied throughout the program. The mechanism is based on the concept of *proof* in mathematical logic: instead of a step-by-step algorithm, a program is considered to be a set of logical formulas which are assumed to be true (axioms), and a computation is an attempt to prove a formula from the axioms of the program.[1]

The second abstraction is that assignment statements and explicit pointers are no longer used; instead, a generalized pattern matching mechanism called *unification* is used to construct and decompose data structures. Implementations of unification create implicit pointers between components of data structures, but all the programmer sees is abstract data structures such as lists, records and trees.

After we discuss "pure" logic programming, we will describe the compromises introduced by Prolog, the first and still very popular, practical logic programming language.

17.1 Pure logic programming

Consider the C function that checks if one string is a substring of another, returning a pointer to the first occurrence or zero if there is no such occurrence. This is a non-trivial program with two nested loops and assorted boundary conditions to be checked, just the sort of program that is likely to have an obscure bug. A possible implementation is as follows:

C

```
    /* Is s1 a substring of s2 ? */
char *substr(char *s1, char *s2)
```

[1]More precisely, that the formula is a *logical consequence* of the program. Rather than digress to explain this concept, we will use the more intuitive notion of proof.

```
{
    char *p = s2;                  /* Used to index through s2 */
    int len1 = strlen(s1) - 1;     /* Store lengths */
    int len2 = strlen(s2) - 1;
    int i;                         /* Index through strings */

    while (*p != 0) {              /* Check for end of s2 */
        i = len1;
        if (p+i) > (s2+len2)       /* Too few characters */
            return NULL;
        while ( (i >= 0) && (*(p+i) == *(s1+i)) )
            i--;                   /* Match from end of s1 */
        if (i == -1) return p;     /* Found - matched all of s1 */
        else p++;                  /* Not found - continue loop */
    }
    return NULL;
}
```

Let us try to define exactly what it means for a string *s* to be a substring of another string *t*. We denote the substring relation by "$\sqsubseteq$", and the concatenation operator by "$\|$".

$$\begin{array}{l} t \sqsubseteq t \\ (t = t1 \,\|\, t2) \wedge (s \sqsubseteq t1) \Rightarrow (s \sqsubseteq t) \\ (t = t1 \,\|\, t2) \wedge (s \sqsubseteq t2) \Rightarrow (s \sqsubseteq t) \end{array}$$

These are three logical formulas that define substring in terms of the more primitive concept of concatenation. The first formula gives a basic *fact*, namely that every string is a substring of itself. The next two formulas state that if a string *t* is decomposed into two (smaller) strings *t1* and *t2*, and ("$\wedge$") *s* is a substring of one of these components, then ("$\Rightarrow$") *s* is a substring of *t*. Of course you know that these are not circular definitions but recursive definitions: reducing the problem to smaller and smaller problems until it can be solved by a base case.

But what does this have to do with computation? Suppose that we reverse the logical formulas:

$$\begin{array}{l} t \sqsubseteq t \\ (s \sqsubseteq t) \Leftarrow (t = t1 \,\|\, t2) \wedge (s \sqsubseteq t1) \\ (s \sqsubseteq t) \Leftarrow (t = t1 \,\|\, t2) \wedge (s \sqsubseteq t2) \end{array}$$

and suppose that we have two specific strings *s* and *t*, and we wish to check if *s* is a substring of *t*. The logical formulas tell us exactly how to do this. First check if *s* is perhaps the same as *t*; if not, decompose *t* into two substrings *t1* and *t2*, and (recursively) check if *s* is a substring of either one. Perform this check for *every* possible decomposition of *t* until you find a sequence of decompositions which shows that *s* is a substring of *t*. If no such sequence exists, then *s* is not a substring of *t*.

The reasoning is done backwards: what would have to be true in order for $s \sqsubseteq t$ to be true? Well, if:

$$(t = t1 \,\|\, t2) \wedge (s \sqsubseteq t1)$$

were true, that would solve the problem. But in order for $s \sqsubseteq t1$ to be true for some substring *t1*:

$$(t1 = t11 \parallel t12) \wedge (s \sqsubseteq t11)$$

or:

$$(t1 = t11 \parallel t12) \wedge (s \sqsubseteq t12)$$

would have to be true and so on. This recursive reasoning can be continued until a fact such as $s \sqsubseteq s$ is reached which is unconditionally true.

For example, is "wor" $\sqsubseteq$ "Hello world"? "Hello world" can be decomposed in twelve different ways:

""	"Hello world"
"H"	"ello world"
"He"	"llo world"
...	...
"Hello wor"	"ld"
"Hello worl"	"d"
"Hello world"	""

For every possible decomposition, we must check the two component strings to see if "wor" is a substring of either of them. Of course each such decomposition (except those with the null string as one of the components) results in a further set of decompositions. But eventually, we will decompose a string into "" and "wor" and then successfully conclude that "wor" is a substring of "wor". In fact, there is more than one way of solving the problem, because we could obtain the same components in several different ways, and even obtain the symmetric decomposition "wor" and "".

While the above computation is rather tedious to do by hand, it is just the type of repetitious task at which a computer excels. To execute a logic program, a set of logical formulas (the program) and a goal such as:

$$\text{"wor"} \sqsubseteq \text{"Hello world"} \ ?$$

are submitted to a software system which is called an *inference engine*, because it conducts logical inferences from one formula to another until the problem is solved. The inference engine checks if the goal formula can be proven from the axioms, the formulas of the program which are assumed to be true. The answer could be either yes or no, called *success* or *failure* in logic programming. Failure could be caused because the goal does not follow from the program, for example, "wro" is not a substring of "Hello world", or it could be caused by an incorrect program, for example if we omitted one of the formulas of the program. There is a third possibility, namely that the inference engine searches forever without deciding either way, in the same way that a while-loop in C may never terminate.

The basic concepts of logic programming are:

- The program is declarative and consists solely of formulas in mathematical logic.
- Each set of formulas for the same *predicate* (such as "$\sqsubseteq$") is interpreted as a (possibly recursive) procedure.

- A particular computation is defined by submitting a *goal*, which is a formula that is to be checked to see if it can be proven from the program.
- The compiler is an inference engine that *searches* for a possible proof of the goal from the program.

Thus every logic program has a dual reading: one as a set of formulas and another as a specification of a computation. In a sense, a logic program is a minimal program. In software engineering, you are taught to precisely specify the meaning of a program before attempting to implement it, and a precise specification uses a formal notation, usually some form of mathematical logic. If the specification *is* the program, there is nothing further to do, and thousands of programmers can be replaced by a handful of logicians. The reason that logic programming is not trivial is that pure logic is not efficient enough for practical programming, and thus there is a clear step that must be taken from the scientific theory of logic to its engineering application in programming.

There are no "control statements" in logic programming, because the control structure is uniform over all programs and consists of a search to prove a formula. Searching for solutions to a problem is, of course, not new; what is new is the suggestion that searching for solutions to computational problems be done in the general framework of logical proofs. Logic became logic programming when it was discovered that by limiting the structure of the formulas and the way that the search for proofs is done, it is possible to retain the simplicity of logical declarations and yet search for problem solutions in an efficient manner. Before explaining how this is done, we must discuss how data is handled in logic programming.

17.2 Unification

A *Horn clause* is a formula which is a *conjunction* ("and") of elementary formulas that implies a conclusion which is a single elementary formula:

$$(s \sqsubseteq t) \Leftarrow (t = t1 \parallel t2) \wedge (s \sqsubseteq t1)$$

Logic programming is based on the observation that by restricting formulas to Horn clauses, just the right balance is achieved between expressiveness and efficient inference. Facts, such as $t \sqsubseteq t$, are conclusions which are not implied by anything, that is, they are always true. The conclusion is also called the *head* of the formula because when written in this reversed form, it appears first in the formula.

To initiate a computation of a logic program, a *goal* is given:

$$\text{"wor"} \sqsubseteq \text{"Hello world"}\ ?$$

The inference engine tries to match the goal and the conclusion of a formula. In this case, there is an immediate match: "wor" matches the variable s and "Hello world" matches the variable t. This defines a *substitution* of terms (in this case constants) for the variables; the substitution is applied to all variables in the formula:

$$\text{"wor"} \sqsubseteq \text{"Hello world"} \Leftarrow (\text{"Hello world"} = t1 \parallel t2) \wedge (\text{"wor"} \sqsubseteq t1)$$

Applying backwards reasoning, we have to show that:

$$(\text{"Hello world"} = t1 \parallel t2) \land (\text{"wor"} \sqsubseteq t1)$$

is true, and this leads to another pattern match, namely trying to match "Hello world" to $t1 \parallel t2$. Here, of course, there are many possible matches and this leads to the search. For example, the inference engine can let $t1$ point to "He" and $t2$ point to "llo world"; these substitutions are then carried through the computation.

Did we say "point to"? Using pattern matching, pointers to arbitrary portions of the string can be created without any explicit notation! All the cautions about the dangers of pointers no longer apply, because pointers to components of the data structure are created and maintained automatically (and correctly) by the inference engine. Logic programming abstracts away not only the control structure, but also the necessity for explicit pointers to allocate and manipulate dynamic data.

There is one concept missing in order to make a programming language out of logical formulas, namely, how are answers returned from a computation? In the above example, we just asked if the goal (without variables) is true or not, and the result of the computation is merely success or failure, indicating that the goal is true or false. If the goal contains variables, the computation may succeed provided that certain substitutions are made for the variables. In mathematical logic, this would be expressed using an *existential quantifier*:

$$\exists s(s \sqsubseteq \text{"Hello world"})$$

which is true if *there exists* a value that can be substituted for s such that the formula is true.

Expressed as a goal:

$$s \sqsubseteq \text{"Hello world"}$$

it matches the first formula for substring $t \sqsubseteq t$, *provided* that both s and "Hello world" are substituted for t. This, of course, is possible only if "Hello world" is also substituted for s. The substitution of the string for the variable defines an answer to the goal:

$$[\text{"Hello world"} \to s]$$

Suppose we ask the inference engine to continue searching. The goal also matches the head of the second formula giving:

$$(\text{"Hello world"} = t1 \parallel t2) \land (s \sqsubseteq t1)$$

By choosing one of the possible decompositions of "Hello world", the inference engine will obtain the formula:

$$(\text{"Hello world"} = \text{"Hello w"} \parallel \text{"orld"}) \land (s \sqsubseteq \text{"Hello w"})$$

But $s \sqsubseteq$ "Hello w" matches $t \sqsubseteq t$ under the substitution:

$$[\text{"Hello w"} \to s]$$

giving another answer to the goal.

Generalized pattern matching, called *unification*, enables links to be maintained between variables and data structures (or components of data structures). Once a variable is *instantiated*, that is once a variable receives a concrete value, that value is instantly propagated to all connected variables. When a computation terminates successfully, the substitutions that have been made to the variables in the goal form the answer.

Note that unification is symmetrical in all arguments of a goal. Suppose the goal is:

$$\text{"Hello world"} \sqsubseteq t$$

This means: Is "Hello world" the substring of some string? The answer will be "Hello world", or even "Hello worldxxx". Logic programs are directionless and can be run "forwards" or "backwards", and generate multiple solutions.

Thus logic programming is based on a uniform control structure that searches for logical inferences, and on a uniform data structuring mechanism that composes and decomposes data. It is correct to say that there is only one "statement" in logic programming, namely an inference step that uses unification to match an elementary formula with a formula. It is hard to conceive that a programming language could be much simpler than this, with no bulky manuals containing dozens of rules and regulations to study! Of course, the single "statement" in the language is extremely powerful and it takes a lot of practice until you become adept at writing logic programs.

17.3 Prolog

Computation and search rules

Prolog is a logical programming language that can be used in practice to write concise, efficient programs. Prolog makes two compromises with the ideal logic programming concept that we have discussed. The first is to define *specific* search and order of computation rules on the logical formulas, so that the language can be efficiently implemented. The second is to include *non-logical predicates*, which are elementary formulas that have no logical meaning but instead are used for their side-effects, such as input and output.

Prolog programs consist of sets of Horn clauses (one elementary formula is implied by a conjunction of zero or more other formulas). Each set of Horn clauses with the same head is called a procedure:

```
substring(T, T).
substring(S, T) :– concat(T, T1, T2), substring(S, T1).
substring(S, T) :– concat(T, T1, T2), substring(S, T2).
```

The sign ":–" denotes implication, and variables must begin with upper-case letters. Given a goal:

```
?– substring("wor", "Hello world").
```

computation proceeds by attempting to unify the goal with the head of a formula; if the unification succeeds, the goal is replaced by the sequence of elementary formulas (also called goals):

```
?– concat("Hello world", T1, T2), substring("wor", T1).
```

The goal that results may consist of more than one elementary formula; the inference engine must now choose one of them to continue the attempt to find a solution. The *computation rule* of Prolog specifies that the inference engine always chooses the *leftmost* elementary formula. In the example, the computation rule requires that the concat be chosen before the recursive call to substring.

There may be more than one formula whose head matches the chosen elementary formula, and the inference engine must choose one of them to attempt unification. The Prolog *search rule* specifies that formulas be attempted in the order that they appear in the program text. When attempting to match a goal formula with the formulas of the substring procedure, the search rule requires that the fact substring(T,T) be chosen first, then the second formula with substring(S,T1), and only if these fail, the third formula with substring(S,T2).

The reason for these seemingly arbitrary requirements is that they enable Prolog to be implemented on a stack architecture just like C and Ada, and most computation in Prolog is just as efficient as it would be in imperative languages. The computation is done by *backtracking*. In the above example:

```
?– concat("Hello world", T1, T2), substring("wor", T1).
```

suppose that the computation has proven concat with the substitution:

$$["H" \rightarrow t1, "ello world" \rightarrow t2]$$

Now an attempt is made to prove substring("wor", "H") which obviously fails. The computation backtracks and attempts to find another proof of concat with a different substitution. All the data needed for the computation of substring("wor", "H") can be discarded upon backtracking. Thus the computation rule of Prolog maps naturally into an efficient stack implementation.

To further improve the efficiency of Prolog programs, the language includes a feature called the *cut* (denoted “!”), which tells the inference engine to refrain from searching part of the potential solution space. It is the programmer’s responsibility to ensure that there are no possible solutions that are “cut away”. For example, suppose that we are trying to parse an arithmetic expression which is defined as two terms separated by an operator:

```
expression(T1, OP, T2) :– term(T1), operator(OP), !, term(T2).

operator('+').
operator('–').
operator('*').
operator('/').
```

and that the goal is expression(n,'+',27). Obviously, both n and 27 are terms, and '+' is one of the operators, so the goal succeeds. If, however, the goal is expression(n,'+','>'), the computation will proceed as follows in the absence of the cut:

n is a term
'+' matches operator('+')
'>' is *not* a term
'+' does not match operator('–')
'+' does not match operator('*')
'+' does not match operator('/')

The inference engine backtracks and tries different ways to satisfy operator(OP), in the hope that a different match will also allow it to satisfy term(T2). Of course, the programmer knows that that is hopeless; the cut causes the entire formula for expression to fail if a failure occurs after the cut is passed. Of course, the cut takes Prolog further away from the ideal of declarative logic programming, but it is extensively used in practice to improve efficiency.

Non-logical formulas

To be truly practical, Prolog includes features that have nothing to do with logic programming. By definition, output statements have no logical meaning on a computation, since their only effect is on some environment outside the program. Nevertheless, output statements are necessary if we are to write programs that open files, display characters on the screen and so on.

Another area in which Prolog departs from pure logic programming is that of numerical computation. It is certainly possible to define addition in logic; in fact, that is the only way to rigorously define it:

$$N + 0 = N$$
$$N + s(M) = s(K) \Leftarrow N + M = K$$

0 is the numeral zero and $s(N)$ is the numeral that succeeds N, so for example, $s(s(s(0)))$ is the numeral 3. The formulas define "+" using the two rules: (1) a number plus zero is the number itself, and (2) N plus the successor of M is the successor of $N+M$. Obviously it would be extremely tedious to write, and inefficient to execute, the logical version of $555+777$.

Prolog includes the elementary formula:

```
Var is Expression
```

Expression is evaluated and a new variable Var is created with this value. Note that this is *not* true assignment; the value of the variable can never be assigned to again, only used as an argument in some later elementary formula.

Expression evaluation and assignment to a newly created variable can be used to simulate for-loops:

```
loop(0).
loop(N) :-
   proc,
   N1 is N - 1,
   loop(N1).
```

The following goal will execute proc ten times:

```
?- loop(10).
```

The argument is a variable used as an index. The first formula is the base case of the recursion: when the index is zero, there is nothing further to do. Otherwise, the procedure proc is executed, a *new* variable N1 is created, set to N–1, and used as an argument to the recursive call to loop. Unification creates a new variable for each use of the second formula of loop. This is not too inefficient because it can be done on the stack. Also, many Prolog compilers can do tail-recursion optimization which is a method for replacing recursion by ordinary iteration if the recursive call is the last statement executed in a procedure.

The reason that the use of is is non-logical is that it is not symmetric, that is, you cannot write:

```
28 is V1 * V2
```

or even:

```
28 is V1 * 7
```

where V1 and V2 are uninstantiated variables, that is variables which have not had values substituted in them. This would require semantic knowledge of arithmetic (how to factor and divide integers), whereas unification performs a purely syntactical matching on terms.

The Prolog database

There is no limit on the number of formulas that a Prolog program can contain, other than any limits imposed by the size of the computer's memory. In particular, there is no limit on the number of facts that can be included, so a set of Prolog facts can fulfil the function of a table in a database:[2]

```
customer(1, "Jonathan").  /* customer(Cust_ID, Name) */
customer(2, "Marilyn").
customer(3, "Robert").

salesperson(101, "Sharon")./* salesperson(Sales_ID, Name) */
salesperson(102, "Betty").
salesperson(103, "Martin").

order(103, 3, "Jaguar").    /* order(Sales_ID, Cust_ID, Article) */
order(101, 1, "Volvo").
order(102, 2, "Volvo").
order(103, 1, "Buick").
```

Ordinary Prolog goals can be interpreted as database queries. For example:

```
?– salesperson(Sales_ID, "Sharon"),          /* Sharon's ID */
   order(Sales_ID, Cust_ID, "Volvo"),        /* Order for Volvo */
   customer(Cust_ID, Name).                  /* Customer of this order */
```

[2]This type of database is known as a *relational database*.

means: “To whom did Sharon sell a Volvo?”. If the query is successful, the variable Name will receive the substitution of the name of one of those customers. Otherwise, we can conclude that Sharon did not sell any Volvos.

Sophisticated database queries become simple Prolog goals. For example: “Is there a customer who was sold a car by both Sharon and Martin?”:

```
?– salesperson(ID1, "Sharon"),       /* Sharon's ID */
   salesperson(ID2, "Martin"),       /* Martin's ID */
   order(ID1, Cust_ID, _),           /* Sharon's customer ID */
   order(ID2, Cust_ID, _).           /* Martin's customer ID */
```

Since the variable Cust_ID is common to two elementary formulas, the goal can be true only if the same customer ordered from each of the salespersons.

Is Prolog a practical alternative to specialized database software? The implementation of lists of facts is quite efficient and can easily answer queries on tables of thousands of entries. However, if your tables consist of tens of thousands of entries, more sophisticated search algorithms are needed. Also, if your database is intended for non-programmers, an appropriate user interface is needed, and your Prolog implementation may or may not be an appropriate programming language in this case.

It is important to emphasize that “this was *not* done by professionals.” That is, no new language or database concepts were introduced; this is just ordinary Prolog programming. Any programmer can create small databases just by listing the facts and then submit queries at any point during a computation.

Dynamic databases

If all sets of facts exist at the beginning of a Prolog program, the queries are perfectly declarative: they just ask for a conclusion based on a set of assumptions (the facts). However, the Prolog language includes a non-logical facility for modifying the database during the course of an inference. The elementary formula assert(F) is always true as a logical formula, but as a side-effect it adds the fact F to the database; similarly, retract(F) removes the fact F:

```
?– assert(order(102, 2, "Peugeot")),/* Betty sells a car */
   assert(order(103, 1, "BMW")),  /* Martin sells a car */
   assert(order(102, 1, "Toyota")), /* Betty sells a car */
   assert(order(102, 3, "Fiat")),   /* Betty sells a car */
   retract(salesperson(101, "Sharon")). /* Fire Sharon ! */
```

Database modifications can be used to simulate an assignment statement in Prolog. Assume that a fact count(0) exists in the program, then:

```
increment :–
   retract(count(N)),        /* Erase old value */
```

```
    N1 is N + 1,            /* New variable with new value */
    assert(count(N1)).      /* Restore new value */
```

Not one of the three elementary formulas is a logical formula!

Recall that assignment is used to record the state of a computation. Thus the alternative to simulating assignment is to carry every state variable as an additional argument in the formulas, which can become both tedious and confusing. In practice, use of non-logical database operations in Prolog programs is acceptable, either to implement dynamic databases or to improve the readability of the program.

Sorting in Prolog

As an example of the relation between the descriptive and the procedural views of a logic program, we will discuss sort programs in Prolog. We limit ourselves to sorting lists of integers. Notation: [Head|Tail] is a list whose first element is Head and whose remaining elements form a list Tail. [ ] denotes the empty list.

Sorting is truly trivial in logic programming because all we have to do is to describe what it means for list L2 to be a sorted version of list L1: L2 consists of an arrangement (permutation) of all the elements of L1 with the restriction that the elements are ordered:

```
sort(L1, L2) :– permutation(L1, L2), ordered(L2).
```

where the formulas in the body are defined as:

```
permutation([ ], [ ]).
permutation(L, [X | Tail]) :–
    append(Left_Part, [X | Right_Part], L),
    append(Left_Part, Right_Part, Short_List),
    permutation(Short_List, Tail).

ordered([ ]).
ordered([Single]).
ordered([First, Second | Tail]) :–
    First =< Second,
    ordered([Second | Tail]).
```

Just read these descriptively:

- The empty list is a permutation of the empty list. A permutation of a non-empty list is a division of the list into an element X and two parts Left_Part and Right_Part, such that X is appended to the front of a permutation of the concatenation of the two parts. For example:

  ```
  permutation([7,2,9,3], [9|Tail])
  ```

  if Tail is a permutation of [7,2,3].

- A list with zero or one elements is ordered. A list is ordered if the first two elements are ordered, and the list consisting of all but the first element is also ordered.

Procedurally, this is not the most efficient sorting program in the world; in fact, it is usually called *slow sort* ! It simply tries (generates) all permutations of the list of numbers until it finds a sorted list. However, it is just as easy to write a descriptive version of more efficient sorting algorithms such as *insertion sort*, which we solved in ML in the previous chapter:

```
insertion_sort([], []).
insertion_sort([Head | Tail], List) :–
   insertion_sort(Tail, Tail_1),
   insert_element(Head, Tail_1, List).

insert_element(X, [], [X]).
insert_element(X, [Head | Tail], [X, Head | Tail]) :–
   X =< Head.
insert_element(X, [Head | Tail], [Head | Tail_1]) :–
   insert_element(X, Tail, Tail_1).
```

Procedurally, the program is quite efficient because it does the sort by directly manipulating sublists without aimless searching. As with functional programming, there are no indices, no for-loops, no explicit pointers and the algorithm generalizes immediately to sorting other objects.

Typing and failure

Prolog is not statically type-checked. Unfortunately, the response of the Prolog inference engine to type errors can cause serious difficulties for the programmer. Suppose that we write a procedure to compute the length of a list:

```
length([], 0).              /* Length of empty list is 0 */
length([Head | Tail], N) :– /* Length of list is */
   length(Tail, N1),        /*   length of Tail */
   N1 is N+1.               /*   plus 1 */
```

and accidentally call it with an integer value rather than with a list:

```
?- length(5, Len).
```

This is not illegal because it is certainly possible that the definition of length contains an additional matching formula.

The response of the inference engine is simply to *fail* on the call to length, which is not what you were expecting. Now length was called within some other formula p, and the failure of length will cause p to fail (which you were not expecting either), and so on back up the chain of calls. This result is uncontrolled backtracking which eventually causes the original goal to fail for no obvious reason. Finding such bugs is a very difficult matter of tracing calls step by step until the error is diagnosed.

For this reason, some dialects of Prolog are typed, requiring you to declare that an argument is either an integer or a list or some programmer defined type. In a typed Prolog, the above call would be a compilation error. The usual trade-offs apply when assessing such dialects: reduced flexibility as opposed to catching errors at compile time.

17.4 Advanced logic programming concepts

Building on the success of Prolog, other logic programming languages have been proposed. Many languages have attempted to combine the advantages of logic programming with other programming paradigms such as object-oriented programming and functional programming. Perhaps the most intensive effort has been invested in attempts to exploit the concurrency inherent in logic programming. Recall that a logic program consists of a sequence of formulas:

$$\begin{array}{l} t \sqsubseteq t \\ (t = t1 \parallel t2) \wedge (s \sqsubseteq t1) \Rightarrow (s \sqsubseteq t) \\ (t = t1 \parallel t2) \wedge (s \sqsubseteq t2) \Rightarrow (s \sqsubseteq t) \end{array}$$

and that at any point during the computation, the inference engine is attempting to reduce a sequence of goals such as:

$$\ldots \wedge (\text{"Hel"} \parallel t1) \wedge \ldots \wedge (t1 \parallel \text{"orld"}) \wedge \ldots$$

The Prolog language computes each goal sequentially from left to right, but it is also possible to evaluate the goals concurrently. This is called *and-parallelism* because of the conjunction that connects formulas of the goal. When matching a goal with the head of a program formula, the Prolog language tries each formula sequentially in textual order, but it is also possible to try the formulas concurrently. This is called *or-parallelism* because each goal must match the first formula *or* the second formula, and so on.

Defining and implementing concurrent logic languages is difficult. The difficulties in *and*-parallelism arise from synchronization: when a single variable appears in two different goals, like $t1$ in the example, only one goal can actually instantiate (write to) the variable, and it must block other goals from reading the variable before the instantiation is done. In *or*-parallelism, several processes perform a concurrent search for a solution, one for each formula of the procedure; when a solution is found, some provision must be made to communicate this fact to other processes so that they can terminate their searches.

There has also been much effort into integrating functional and logic programming. There is a very close relationship between the mathematics of functions and logic, because:

$$y = f(x_1, \ldots, x_n)$$

is equivalent to the truth of the logical formula:

$$\forall x_1 \cdots \forall x_n \exists ! y (y = f(x_1, \ldots, x_n))$$

(where $\exists! y$ means "there exists a unique y"). The main differences between the two programming concepts are:

1. Logic programming uses (two-way) unification which is stronger than (one-way) pattern matching used in functional programming.

2. Functional programs are one-directional in that given all the arguments the program returns a value. In logic programs, any of the arguments of the goal can be left unspecified and unification is responsible for instantiating them with answers.

3. Logic programming is based on an inference engine which automatically searches for answers.

4. Functional programming naturally treats higher-level objects, since functions and types are first-class objects that can be used as data, whereas logic programming is more or less limited to formulas on ordinary data types.

5. Similarly, the high-order facilities in functional programming languages generalize naturally to modules whereas logic programming languages tend to be "flat".

A new area of research in logic programming is the extension of matching from purely syntactical unification to include semantic information. For example, if the goal specifies $4 < x < 8$ and the formula head specifies $6 \leq x < 10$, then we can solve for $6 \leq x < 8$, that is $x = 6$ or $x = 7$. Languages that include semantic information in the matching are called *constraint logic programming languages*, because values are constrained by equations. Constraint logic languages must be based on efficient algorithms to solve the underlying equations.

These advances show great promise in improving both the level of abstraction and the efficiency of logic programming languages.

17.5 Exercises

1. Compute $3+4$ using the logical definition of addition.

2. What will happen if the loop program is called with the goal loop(−1)? How can the program be fixed?

3. Write a Prolog program that does not terminate because of the specific computation rule of the language. Write a program that does not terminate because of the search rule.

4. The Prolog rules call for a *depth-first search* for solutions, because the leftmost formula is repeatedly selected even after it has been replaced. It is also possible to do a *breadth-first search* by selecting formulas sequentially from left to right, only returning to the leftmost formula when all others have been selected. What is the effect of this rule on the success of a computation?

5. Write a Prolog goal for the following query:

Is there a type of car that was sold by Sharon but not by Betty? If so, what type of car was it?

6. Study the predefined Prolog formula findall and show how it can solve the following queries:

   How many cars did Martin sell?
   Did Martin sell more cars than Sharon?

7. Write a Prolog program for appending lists and compare it with an ML program. Run the Prolog program in different "directions".

8. How can a Prolog procedure be modified so that it will catch type mismatches and write an error message?

9. What types of logic programs would benefit from and-parallelism and what types would benefit from or-parallelism?