Declarative Programming: Handout 8

Yılmaz Kılıçaslan
December 05, 2022

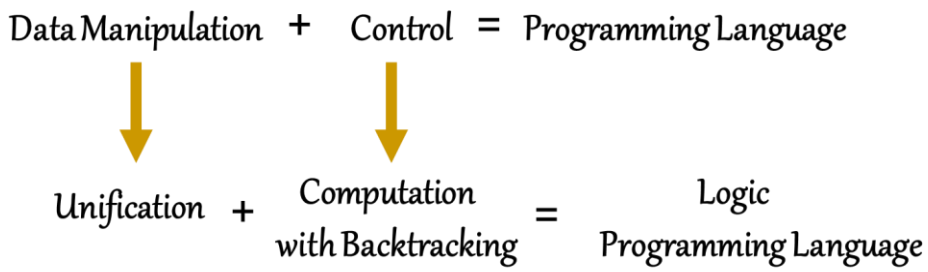
Key Concepts: Declarative/Procedural, Interpreter, Meta-Program, Unification, Substitution, Destructive Assignment.

UNIFICATION

1 INTRODUCTION

As we have pointed out several times, what characterizes a programming language is the control and data manipulation mechanisms that it uses. It is time to see how these mechanisms are implemented in logic programming. Very broadly speaking, it is via the so-called unification operation that data manipulation is achieved in logic programming; and, as for the control mechanism, the control in a logic programming language differs from the control in a conventional programming language mainly in that the former rests on a backtracking strategy, which the latter lacks. Here is a simple diagrammatic representation of the computation model of logic programs:

(1)



In what follows, in order to remove the guesswork and bring our intuitions into awareness we will examine the two components of that computation model in detail. Today, we will look at the unification operation and next week we will be examining the control mechanism used by a Prolog interpreter.

2 DECLARATIVE VERSUS PROCEDURAL INTERPRETATION OF PROLOG

A logic program can be interpreted at two levels: at a **declarative** level and at a **procedural** level. For example, the rule

(2) *α:- β1, β2, …, βn.*

can be read as:

(3) "*α is true if β1, β2, …, and βn are true*"

or as:

(4) *"to solve (execute) α, solve (execute) β1, β2, …, and βn"*.

Obviously, (3) is the declarative interpretation of (2), whereas (4) is its procedural interpretation.

In order to grasp the logic-based essence of Prolog we have, so far, focused on the declarative interpretation of our programs. However, we know that our programs were also interpreted by an interpreter. Technically, a Prolog interpreter is a meta-program that takes a prolog Program and a query as inputs and returns a logical conclusion (along with its side effects):

(5) **PROLOG INTERPRETER:**

It might sound interesting but the Prolog interpreter kept interpreting our programs procedurally. The reason behind this is simple: Our regular (i.e., non-quantum) computers are part of a world where ordering matters and, hence, interpreters, as 'translators' between programs and computers, are bound to interpret programs in a procedural way.

In fact, Prolog falls short of being a completely logical programming language. In other words, a Prolog interpreter cannot provide a programming environment where the programmer can enjoy the comfort of ignoring all procedural issues. More positively speaking, Prolog is an approximation to logic programming but one of the best. This means that besides other intellectual rewards, understanding the Prolog interpreter will help us be better Prolog programmers. Up to this point in addition to our logical knowledge we have relied on guesswork and intuitions in order to understand how the interpreter interpreted our programs. It is now time to go into the internal mechanisms of a Prolog interpreter. As already said, we will look at the unification mechanism today.

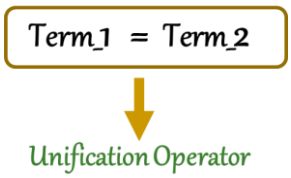
3 UNIFICATION

3.1 Some Clarification

As Sterling and Shapiro (1992) point out, unification is the heart of the computation model of logic programs and the basis of the uses of logical inference in artificial intelligence. In the next section we will examine the unification algorithm that they offer. However, some terminological and conceptual clarification is needed in order to understand that algorithm.

The key operator for unification is the unification operator, which is represented by the symbol =. A unification operation that successfully applies to two terms make them equal:

(6) **UNIFICATION AS AN EQUATION**



A set of atomic unification operations determines a substitution operation in a way that is defined below:

(7) **SUBSTITUTION:**

> A *substitution* is a finite set (possibly empty) of pairs of the form $X_i = t_i$, where X_i is a variable and t_i is a term, and $X_i \neq X_j$ for every $i \neq j$, and X_i does not occur in t_j, for any i and j.

{X = mary, Y = sarah} and {Y = sarah} are examples of substitution. A substitution can even be an empty set, as is the case at the outset of the of the unification algorithm. Substitutions can be applied to terms:

(8) **APPLICATION OF A SUBSTITUTION:**

> The result of applying a substitution θ to a term A, denoted by $A\theta$, is the term obtained by replacing every occurrence of X by t in A, for every pair $X = t$ in θ.

2.2 Unification of Two Terms

The unification of two terms results in their becoming identical and this might require a substitution to be applied to the terms. One of the simple cases is the one where the left-hand side of the unification operation is a variable. In this case, whatever is on the right-hand side is substituted for the variable on the left-hand side:

(9) ?- X = a, Y = func(X).
X = a,
Y = func(a)

(10) ?- X = Y, Z = func(X).
X = Y,
Z = func(Y)

(11) ?- X = Y, Z = func(X), W = func(Y).
X = Y,
Z = W, W = func(Y)

(12) ?- X = Y, Z = func(Y), W = func(X).
X = Y,
Z = W, W = func(Y)

(13) Y = X, Z = func(Y), W = func(X).
Y = X,
Z = W, W = func(X)

2.3 Unification versus Assignment

Unification is quite different from the assignment operation used in traditional programming languages. Variables in conventional programming languages (e.g., C, C#, Java) are associated with memory locations. They are treated simply as containers holding values. The content of a conventional variable may change from one value to another. This is called destructive assignment. In logic programming, variables refer to individuals rather than memory locations. Therefore, once a logical variable has been specified as referring to a particular individual, it can no longer be made to refer to another individual, as illustrated below.

(14) ?- X = a, X = b.
false.

More generally, once a variable unifies with a 'ground' term (i.e., a term where variables do not occur), it will fail to unify with a different ground term:

(15) ?- X = couple(john, mary), X = couple(john, sue).
false.

However, a variable can unify with an unlimited number of variables:

(16) ?- X = Y, X = Z, X = W, X = A.
X = Y, Y = Z, Z = W, W = A

2.4 A Unification Algorithm

A unification algorithm is expetcted to capture at least all the points stated above. Below is such an algorithm (written by Sterling and Shapiro (1992) in pseudocode):

(5) **A UNIFICATION ALGORITHM:**

Input: Two terms T_1 and T_2 to be unified

Output: θ, the mgu of T_1 and T_2, or *failure*

Algorithm: Initialize the substitution θ to be empty,
the stack to contain the equation $T_1 = T_2$,
and failure to *false*.

while stack not empty and no failure *do*
pop $X = Y$ from the stack
case
X is a variable that does not occur in Y:
substitute Y for X in the stack and in θ
add $X = Y$ to θ
Y is a variable that does not occur in X:
substitute X for Y in the stack and in θ
add $Y = X$ to θ
X and Y are identical constants or variables:
continue
X is $f(X_1, \ldots, X_n)$ and Y is $f(Y_1, \ldots, Y_n)$
for some functor f and $n > 0$:
push $X_i = Y_i, i = 1 \ldots n$, on the stack
otherwise:
failure is *true*

If failure, *then* output *failure else* output θ.

Note that the unification algorithm above treats the both sides of the unification operator in the same way. For example, the second *case* clause says that evertyhing that applies to a variable on the left-hand side (as stated in the first *case* clause) also applies to a variable on the right-hand side. In short, unlike conventional assignment, unification is a directionless operation. It is for this reason that (17) and (18), for example, yield the same results:

(17) ?- X = couple(john, mary).
X = couple(john, mary)

(18) ?- couple(john, mary) = X.
X = couple(john, mary)

As the objective of a unification operation is to make two terms identical, if the terms to be unified are identical constants or variables there remains nothing to do but to continue the algorithm.

The fourth *case* clause in the unification algorithm is devoted to cases where the terms to be unified are both compound terms. In this case, for the terms to unify they have to have the same principal functor (i.e., predicate) and each of the pairs consisting of their respective arguments/parameters must simultaneously unify. This recursive operation is achieved by pushing the pairs of arguments to be unified onto the stack. Below are some examples where the pairs of compound terms successfully unify:

(19) ?- father(john, mary) = father(john, mary).

(20) ?- father(X, mary) = father(john, mary).
X = john

(21) ?- father(X, mary) = father(john, Y).
X = john,
Y = mary

4 CONCLUSION

Data manipulation is achieved in logic programming via unificaition. Two unified terms become identical. To this effect, The unification of two terms results in their becoming identical and this might require a substitution might be applied to the terms.

5 EXERCISES

A. What makes an interpreter a meta-program?

B. Which logical conclusions can an interpreter return?

C. What side-effects does an interpreter return (along with a logical conclusion)?

D. Determine which of the following unification goals succeed?

1. **pred(f(X), b) = pred(f(Y), Z)**
2. **pred1(a, b, C, d) = pred2(a, B, c, D)**
3. **pred(pred(pred())) = pred(pred(A)).**

4. **[a, B, c, D] = [A, b, C, d].**
5. **[a|[b|C] = [a, b, c, d]**
6. **[] = [X|Xs]**
7. **[a, b, c, d] = [a|_].**
8. **A = b.**
9. **B = a.**
10. **a = b.**

E. Specify a most general unifier for each of the successful unification operations in D?

References

Sterling, L. and Shapiro E. (1986) The Art of prolog. Cambridge, Massachusetts: MIT Press.