5 Meta-Linguistic Abstraction, Types, and Meta-Interpreters

Chapter Objectives

A number of Prolog meta-predicates are presented, including:

`Atom`
`clause`
`univ` (`=..`)
`call`

The type system for Prolog:

Programmer implements typing as needed
Types as run time constraints rather than enforced at compile time

Unification and variable binding explained

Evaluation versus unification

`is` versus =
Difference lists demonstrated

Chapter Contents

5.1 Meta-Interpreters, Types, and Unification

Meta-Logical Predicates

In this chapter we first consider a set of powerful Prolog predicates, called *meta-predicates*. These predicates take as their scope other predicates in the Prolog environment. Thus they offer tools for building *meta-interpreters*, interpreters in a language that are able to interpret specifications in that language. An example will be to build a rule interpreter in Prolog, an interpreter that can manipulate and interpret rule sets, specified in Prolog syntax. These interpreters can also be used to query the user, offer explanations of the interpreter's decisions, implement multi-valued or fuzzy logics, and run any Prolog code.

In Section 5.1 we introduce a useful set of meta-predicates. In Section 5.2 we discuss data typing for Prolog and describe how type constraints can be added to a prolog system. An example of a typed relational database in Prolog is given. Finally, in Section 5.3, we discuss unification and demonstrate with difference lists how powerful this can be.

Meta-logical constructs extend the expressive power of any programming environment. We refer to these predicates as *meta* because they are designed to match, query, and manipulate other predicates that make up the specifications of the problem domain. That is, they can be used to reason about Prolog predicates rather than the terms or objects these other predicates denote. We need meta-predicates in Prolog for (at least) five reasons:

To determine the "type" of an expression;
To add "type" constraints to logic programming applications;
To build, take apart, and evaluate Prolog structures;
To compare values of expressions;
To convert predicates passed as data to executable code.

We have actually seen a number of meta-predicates already. In Chapter 2 we described how global structures, which are those that can be accessed by the entire clause set, are entered into a Prolog program. The command `assert(C)` adds the clause `C` to the current set of clauses. There are dangers associated with programming with predicates such as `assert` and `retract`. Because these predicates are able to create and remove global structures, they can introduce side effects into the program, and may cause other problems associated with poorly structured programs. Yet, it is sometimes necessary to use global structures to draw on the power of Prolog's built-in database and pattern matching. We do this when creating *semantic nets* and *frames* in a Prolog environment, as in Section 2.4. We may also use global structures to describe new results as they are found with a rule-based expert system shell, as in Section 6.2. We want this information to be global so that other predicates (rules) may access it when appropriate.

Other meta-predicates that are useful for manipulating representations include:

`var(X)` succeeds only when `X` is an unbound variable.

`nonvar(X)` succeeds only when `X` is bound to a nonvariable term.

`=..` creates a list from a predicate term.

For example, `foo(a, b, c) =.. Y` unifies `Y` with `[foo, a, b, c]`. The head of the list `Y` is the predicate name, and its tail is the predicate's arguments. `=..` also can be used to bind alternative variable patterns, of course. Thus, if `X =.. [foo, a, b, c]` succeeds, then `X` has the value `foo(a, b, c)`.

`functor(A, B, C)` succeeds with `A` a term whose principal `functor` has name `B` and arity `C`.

For example, `functor(foo(a, b), X, Y)` will succeed with variables `X` = `foo` and `Y` = `2`. `functor(A, B, C)` can also be used with any of its arguments bound in order to produce the others, such as all the terms with a certain name and/or arity.

`clause(A, B)` unifies `B` with the body of a clause whose head is A.

For example, if `p(X) :- q(X)` exists in the database, then `clause(p(a), Y)` will succeed with `Y` = `q(a)`. This is useful for controlling rule chaining in an interpreter, as seen in Chapter 6.

`any_predicate(…, X, …) :- X` executes predicate `X`, the argument of any predicate.

Thus a predicate, here `X`, may be passed as a parameter and executed at any desired time. `call(X)`, where `X` is a clause, also succeeds with the execution of predicate `X`.

This short list of meta-logical predicates will be very important in building and interpreting AI data structures. Because Prolog can manipulate its own structures in a straightforward fashion, it is easy to implement interpreters that modify the Prolog semantics, as we see next.

5.2 Types in Prolog

For a number of problem-solving applications, the unconstrained use of unification can introduce unintended error. Prolog is an untyped language, in that unification simply matches patterns, without restricting them according to data type. For example, `append(nil, 6, 6)` can be inferred from the definition of `append`, as we will see in Chapter 10. Strongly typed languages such as Pascal have shown how type checking helps programmers avoid these problems. Researchers have proposed adding types to Prolog (Neves et al. 1986, Mycroft and O'Keefe 1984).

Typed data are particularly appropriate in a relational database (Neves et al. 1986, Malpas 1987). The rules of logic can be used as constraints on the data and the data can be typed to enforce consistent and meaningful interpretation of the queries. Suppose that a department store database has `inventory`, `suppliers`, `supplier_inventory`, and other appropriate relations. We define a database as relations with named fields that can be thought of as sets of tuples. For example, `inventory` might consist of 4-tuples, where:

```
< Pname, Pnumber, Supplier, Weight >  inventory
```

only when `Supplier` is the supplier name of an `inventory` item numbered `Pnumber` that is called `Pname` and has weight `Weight`. Suppose further:

```
< Supplier, Snumber, Status, Location >  suppliers
```

only when `Supplier` is the name of a supplier numbered `Snumber` who has status `Status` and lives in city `Location`. Suppose finally:

```
< Supplier, Pnumber, Cost, Department >
     supplier_inventory
```

only if `Supplier` is the name of a supplier of part number `Pnumber` in the amount of `Cost` to department `Department`.

We may define Prolog rules that implement various queries and perform type checking across these relationships. For instance, the query "are there suppliers of part number 1 that live in London?" is given in Prolog as:

```
?- getsuppliers(Supplier,1, london).
```

The rule:

```
getsuppliers(Supplier, Pnumber, City) :-
     cktype(City, suppliers, city),
     suppliers(Supplier, _, _,City),
     cktype(Pnumber, inventory, number),
     supplier_inventory(Supplier, Pnumber, _, _),
     cktype(Supplier, inventory, name).
```

implements this query and also enforces the appropriate constraints across the tuples of the database. First the variables Pnumber and City are bound when the query unifies with the head of the rule; the predicate cktype tests that Supplier is an element of the set of suppliers, that 1 is a legitimate inventory number, and that london is a suppliers' city.

We define cktype to take three arguments: a value, a relation name, and a field name, and to check that each value is of the appropriate type for that relation. For example, we may define lists of legal values for Supplier, Pnumber, and City and enforce data typing by requiring member checks of candidate values across these lists. Alternatively, we may define logical constraints on possible values of a type; for example, we may require that inventory numbers be less than 1000.

We should note the differences in type checking between standard languages such as Pascal and Prolog. We might define a Pascal data type for suppliers as:

```
type supplier = record
     sname: string;
     snumber: integer;
     status: boolean;
     location: string
     end
```

The Pascal programmer defines new types, here supplier, in terms of already defined types, such as boolean or integer. When the programmer uses variables of this type, the compiler automatically enforces type constraints on their values.

In Prolog, we can represent the supplier relation as instances of the form:

```
supplier(sname(Supplier),
     snumber(Snumber),
     status(Status),
     location(Location)).
```

We then implement type checking by using rules such as getsuppliers and cktype. The distinction between Pascal and Prolog type checking is clear and important: the Pascal type declaration tells the compiler the form for both the entire structure (record) and the individual components (boolean, integer, string) of the data type. In Pascal we declare variables to be of a particular type (record) and then create procedures to access these typed structures.

```
procedure changestatus (X: supplier);
     begin
          if X.status then. …
```

Because it is nonprocedural, Prolog does not separate the declaration from the use of data types, and type checking is done as the program is executing. Consider the rule:

```
supplier_name(supplier(sname(Supplier),
              snumber(Snumber),
              status(true),
              location (london))) :-
     integer(Snumber), write(Supplier).
```

supplier_name takes as argument an instance of the supplier predicate and writes the name of the Supplier. However, this rule will succeed only if the supplier's number is an integer, the status is active (true), and the supplier lives in london. An important part of this type check is handled by the unification algorithm (status and location) and the rest is the built-in system-predicate integer. Further constraints could restrict values to be from a particular list; for example, Snumber could be constrained to be from a list of supplier numbers. We define constraints on database queries using rules such as cktype and supplier_name to implement type checking when the program is executed.

So far, we have seen three ways that data may be typed in Prolog. First, and most powerful, is the program designer's use of unification and syntactic patterns to constrain variable assignment. Second, Prolog itself provides predicates to do limited type checking. We saw this with meta-predicates such as var(X), clause(X,Y), and integer(X). The third use of typing occurred in the inventory example where rules checked lists of legitimate Supplier, Pnumbers, and Cities to enforce type constraints.

A fourth, and more radical approach is the complete predicate and data type check proposed by Mycroft and O'Keefe (1984). Here all predicate names are typed and given a fixed arity. Furthermore, all variable names are themselves typed. A strength of this approach is that the constraints on the constituent predicates and variables of the Prolog program are themselves enforced by a (meta) Prolog program. Even though the result may be slower program execution, the security gained through total type enforcement may justify this cost.

To summarize, rather than providing built-in type checking as a default, Prolog allows run-time type checking under complete programmer control. This approach offers a number of benefits for AI programmers, including the following:

1. The programmer is not forced to adhere to strong type checking at all times. This allows us to write predicates that work across any type of object. For example, the member predicate performs general member checking, regardless of the type of elements in the list.
2. User flexibility with typing helps exploratory programming. Programmers can relax type checking in the early stages of program development and introduce it to detect errors as they come to better understand the problem.
3. AI representations seldom conform to the built-in data types of languages such as Pascal, C++, or Java. Prolog allows

types to be defined using the full power of predicate calculus. The database example showed this flexibility.

4. Because type checking is done at run time rather than compile time, the programmer determines when the program should perform a check. This allows programmers to delay type checking until it is necessary or until certain variables have become bound.

5. Programmer control of type checking at run time also supports the creation of programs that build and enforce new types during execution. This can be of use in a learning or a natural language processing program, as we see in Chapters 7, 8, and 9.

In the next section we take a closer look at unification in Prolog. As we noted earlier, unification is the technical name for pattern matching, especially when applied to expressions in the Predicate Calculus. The details for implementing this algorithm may be found in Luger (2009, Section 2.3). In Prolog, unification is implemented with backtracking that supports full systematic instantiation of all values defined for the problem domain. To master the art of Prolog programming the sequential actions of the interpreter, sometimes referred to as Prolog's "procedural semantics" must be fully understood.

5.3 Unification, Engine of Variable Binding and Evaluation

An important feature of Prolog programming is the interpreter's behavior when considering a problem's specification and faced with a particular query. The query is matched with the set of specifications to see under what constraints it might be true. The interpreter's action, left-to-right depth first backtracking across all specified variable bindings, is a variation of the search of a resolution-based reasoning system.

But Prolog is NOT a full mathematically sound theorem prover, as it lacks several important constraints, including the occurs check, and Prolog also supports the use of cut. For details see Luger 2009, Section 14.3). The critical point is that Prolog performs a systematic search across database entries, rather than, as in traditional languages, a sequential evaluation of statements and expressions. This has an important result: variables are bound (assigned values or instantiated) by *unification* and not by an evaluation process, unless, of course, an evaluation is explicitly requested. This paradigm for programming has several implications.

The first and perhaps most important result is the relaxation of the requirement to specify variables as input or output. We saw this power briefly with the `member` predicate in Chapter 2 and will see it again with the `append` predicate in Chapter 10. `append` can either join lists together, test whether two lists are correctly appended, or break a list into parts consistent with the definition of `append`. We use unification as a constraint handler for parsing and generating natural language sentences in Chapters 7 and 8.

Unification is also a powerful technique for rule-based and frame-based expert systems. All production systems require a form of this matching, and it is often necessary to write a unification algorithm in languages that don't provide it, see, for example, Section 15.1 for a Lisp implementation of unification.

An important difference between unification-based computing and the use of more traditional languages is that unification performs syntactic matches (with appropriate parameter substitutions) on structures. It does *not* evaluate expressions. Suppose, for example, we wished to create a `successor` predicate that succeeds if its second argument is the arithmetic successor of its first argument. Not understanding the unification/evaluation paradigm, we might be tempted to define `successor`:

```
successor (X, Y) :- Y = X + 1.
```

This will fail because the `=` operator does not evaluate its arguments but only attempts to unify the expressions on either side. This predicate succeeds if `Y` unifies with the structure `X + 1`. Because 4 does not unify with 3 + 1, the call `successor(3, 4)` fails! On the other hand, demonstrating the power of unification, `=` can test for equivalence, as defined by determining whether substitutions exist that can make *any* two expressions equivalent. For example, whether:

```
friends (X, Y) = friends(george, kate).
```

In order to correctly define `successor` (and other related arithmetic predicates), we need to be able to evaluate arithmetic expressions. Prolog provides an operator, `is`, for just this task. `is` evaluates the expression on its right-hand side and attempts to unify the result with the object on its left. Thus:

```
X is Y + Z.
```

unifies `X` with the value of `Y` added to `Z`. Because it performs arithmetic evaluation, if `Y` and `Z` do not have values (are not bound at execution time), the evaluation of `is` causes a run-time error. Thus, `X is Y + Z` cannot (as one might think with a declarative programming language) give a value to `Y` when `X` and `Z` are bound. Therefore programs must use `is` to evaluate expressions with arithmetic operators, `+`, –, `*`, `/`, and `mod`.

Finally, as in the predicate calculus, variables in Prolog may have one and only one binding within the scope of a single expression. Once given a value, through local assignment or unification, variables can never take on a new value, except through backtracking in the and/or search space of the current interpretation. Upon backtracking, all the instances of the variable within the scope of the expression take on the new value. Thus, `is` cannot function as a traditional assignment operator; and expressions such as `X is X + 1` will always fail.

Using `is`, we now properly define `successor(X, Y)` where the second argument has a numeric value that is one more than the first:

```
successor (X, Y) :- Y is X + 1.
```

`successor` will now have the correct behavior as long as `X` is bound to a numeric value at the time that the `successor` predicate is called.

`successor` can be used either to compute `Y`, given `X`, or to test values assigned to `X` and `Y`:

```
?- successor (3, X).
X = 4
Yes
?- successor (3, 4).
Yes
?- successor (4, 2).
No
?- successor (Y, 4).
failure, error in arithmetic expression
```

since `Y` is not bound at the time that `successor` is called.

As this discussion illustrates, Prolog does not evaluate expressions as a default as in traditional languages such as C++ and Java. The programmer must explicitly call for evaluation and assignment using `is`. Explicit control of evaluation, as also found in Lisp, makes it easy to treat expressions as data, passed as parameters, and creating or modifying them as needed within the program. This feature, like the ability to manipulate predicate calculus expressions as data and execute them using `call`, greatly simplifies the development of different interpreters, such as the expert system shell of the next chapter.

We close this discussion of the power of unification-based computing with an example that does string catenation through the use of *difference lists*. As an alternative to the standard Prolog list notation, we can represent a list as the difference of two lists. For example, `[a, b]` is equivalent to `[a, b | [ ] ] — [ ]` or `[a, b, c] — [c]`. This representation has certain expressive advantages over the traditional list syntax. When the list `[a, b]` is represented as the difference `[a, b | Y] — Y`, it actually describes the potentially infinite class of all lists that have `a` and `b` as their first two elements. Now this representation has an interesting property, namely addition:

```
X — Z = X — Y + Y — Z
```

We can use this property to define the following single-clause logic program where `X — Y` is the first list, `Y — Z` is the second list, and `X — Z` is the result of catenating them, as in Figure 5.1: We create the predicate `catenate` that takes two list `X` and `Y` and creates `Z`:

```
catenate(X — Y, Y — Z, X — Z).
```

This operation joins two lists of any length in constant time by unification on the list structures, rather than by repeated assignment based on the length of the lists (as with `append`, Chapter 10). Thus, the `catenate` call gives:

```
?- catenate ([a, b  Y] — Y, [1, 2, 3] — [ ], W).
Y = [1, 2, 3]
W = [a, b, 1, 2, 3] — [ ]
```

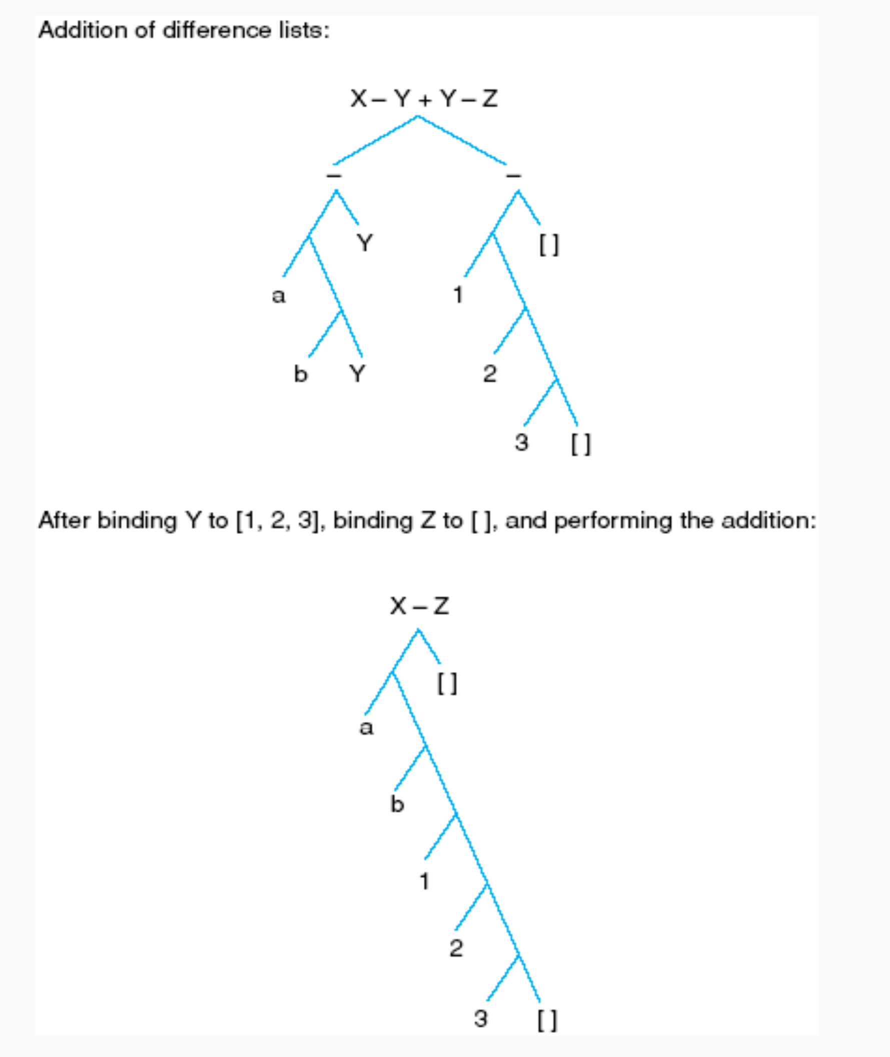
Addition of difference lists:
X – Y + Y – Z
–
–
Y
[ ]
a
1
b
Y
2
3
[ ]
After binding Y to [1, 2, 3], binding Z to [ ], and performing the addition:
X – Z
[ ]
a
b
1
2
3
[ ]

Figure 5.1 Tree diagrams: list catenation using difference lists.

As may be seen in Figure 5.1, the (subtree) value of **Y** in the second parameter is unified with both occurrences of **Y** in the first parameter of `catenate`. This demonstrates the power of unification, not simply for substituting values for variables but also for matching general structures: all occurrences of **Y** take the value of the entire subtree. The example also illustrates the advantages of an appropriate representation. Thus *difference lists* represent a whole class of lists, including the desired catenation.

In this section we have discussed a number of idiosyncrasies and advantages of Prolog's unification-based approach to computing. Unification is at the heart of Prolog's declarative semantics. For a more complete discussion of Prolog's semantics see Luger (2009, Section 14.3).

In Chapter 6 we use Prolog's declarative semantics and unification-based pattern matching to design three meta-interpreters: Prolog in Prolog, the shell for an expert system, and a planner.

Exercises

1. Create a type check that prevents the member check predicate (that checks whether an item is a member of a list of items) from crashing when called on `member(a, a)`. Will this "fix" address the `append(nil, 6, 6)` anomaly that is described in Chapter 9? Test it and see.

2. Create the "inventory supply" database of Section 5.2. Build type checks for a set of six useful queries on these data tuples.

3. Is the difference list `catenate` really a linear time `append` (Chapter 10)? Explain.

4. Explore the powers of unification. Use `trace` to see what happens when you query the Prolog interpreter as follows. Can you explain what is happening?

a: `X = X + 1`

b: `X is X + 1`

c: `X = foo(X)`