Logic Programming

Having just studied the use of Alloy as a tool for checking specifications of software systems, we turn our attention in this module to a related question: whether, once we have a sufficiently detailed specification for a software system, we might be able to simply “execute” the specification as a program that meets it.

More realistically, the question we shall explore is whether specifications for programs can be “compiled” or otherwise translated into specification-compliant executable code without any additional coding burden.

Short answer: no

We must, of course, be reasonable about our expectations!

There is no way we could expect that the brief characterization of file systems we gave in the last module provides sufficient information to derive a working file system implemented on actual hardware.

To begin with, we omitted many features possessed by real file systems: user permissions, handling of concurrent reads/writes, etc.

In principle, we could enhance the model to incorporate these.

But what about actually interacting with a disk drive? Could this behaviour be derived? Maybe, but not without a lot of help!

Long answer: read on

In module 8, we discussed the axiom schema of replacement, which speaks of binary predicates P such that if $P(x, y)$ holds, then the value of x uniquely determines the value of y. For example, we might have

$$P(x, y) = (x = 1 \wedge y = 2) \vee (x = 3 \wedge y = 4)\ .$$

Then from the knowledge that $x = 1$, we can derive $y = 2$; similarly, if we know that $x = 3$ (and also that P holds, of course), then we can conclude $y = 4$.

There is clearly a notion of computation going on here. In treating x as an "input", we can obtain y as an "output". Indeed, we noted that predicates P that behave as above could be regarded as functions.

Can we push this idea further? What if we start considering less contrived predicates? For example, what if a predicate P could be interpreted as follows:

$P(x, y) =$ x is a list of numbers, and y is the reversal of x .

Again, the value of x uniquely determines y.

Assuming we could find a formula expressing P, if we then "plug in" some list for x, could we derive its reversal in y?

If so, then we have just "executed" a formula!

Also note: in this example, the value of y also uniquely determines the value of x. Keep this in mind....

Making it happen

If we want to create a system for deriving y given x and $P(x, y)$, we first must be more precise about just what it is we would like the system to do.

In a more general setting, we are given a set $\phi_1, \ldots, \phi_m$ as premises, inputs $x_1, \ldots, x_n$, and an $(n + p)$-ary predicate P, and we wish to find a $y_1, \ldots, y_p$ such that

$$\phi_1, \ldots, \phi_m \vdash P(x_1, \ldots, x_n, y_1, \ldots, y_p) \, .$$

But there are technical challenges that stand in the way of automating the process....

To begin with, finding $y_1, \ldots, y_p$ for which

$$\phi_1, \ldots, \phi_m \vdash P(x_1, \ldots, x_n, y_1, \ldots, y_p)$$

is essentially satisfiability for predicate logic, which we know is undecidable. In other words, we cannot have both guaranteed termination and (partial) correctness for a system that finds $y_1, \ldots, y_p$.

But is this really a problem?

We should not necessarily be afraid of non-termination; after all, real-world programs can fail to terminate. Further, if we do not admit the possibility of non-termination, then our system will not be powerful enough to solve some problems that other computer programs can.

On the other hand, given that non-termination is a possibility, we will want to be careful about how we go about finding $y_1, \ldots, y_p$, so that we do not fall into any obvious loops, at least.

More specifically, we will have to employ a strict procedure for finding proofs and counterexamples.

We first observe that finding $y_1, \ldots, y_p$ such that

$$\phi_1, \ldots, \phi_m \vdash P(x_1, \ldots, x_n, y_1, \ldots, y_p)$$

is equivalent to giving a constructive proof of

$$\phi_1, \ldots, \phi_m \vdash \exists y_1 \cdots \exists y_p P(x_1, \ldots, x_n, y_1, \ldots, y_p) \, .$$

To simplify the problem, we can negate the conclusion and make it a premise. The problem is then to prove the sequent

$$\phi_1, \ldots, \phi_m, \neg(\exists y_1 \cdots \exists y_p P(x_1, \ldots, x_n, y_1, \ldots, y_p)) \vdash \bot ,$$

i.e.,

$$\phi_1, \ldots, \phi_m, \forall y_1 \cdots \forall y_p \neg P(x_1, \ldots, x_n, y_1, \ldots, y_p) \vdash \bot .$$

In other words, we can simplify the statement of the problem by rephrasing it as a search for a contradiction. In this way, regardless of the form of P, the goal is always to find a contradiction.

Note that unless the other premises $\phi_1, \ldots \phi_m$ are contradictory, finding a contradiction will necessarily involve the predicate P and particular parameters $y_1, \ldots y_p$.

The next challenge we face arises from the considerable diversity that exists among well-formed formulas. Depending on how the grammar rules are applied, there is a potential for building formulas with arbitrarily complex structure.

Related to the potential complexity of formulas is the large number of proof rules that are available to us. We have discussed over 20 proof rules in this course—how is a program to know which ones to use? And if the program must try several alternatives at each step, how will it ever terminate in a reasonable amount of time?

We can help ourselves out by reducing the variety of forms a formula can take, and the number of available proof rules.

Recall from Module 4: every propositional formula can be written as a set of **clauses** (a conjunction of clauses is what is called conjunctive normal form, or CNF). A clause has the following format:

$$p_1 \vee p_2 \vee \cdots \vee p_n \vee \neg q_1 \vee \neg q_2 \vee \cdots \vee \neg q_m$$

and can be written as

$$(q_1 \wedge q_2 \wedge \cdots \wedge q_m) \rightarrow (p_1 \vee p_2 \vee \cdots \vee p_n),$$

where all of the p_i and q_j are atomic propositions.

If $m = 0$, then we have the clause $p_1 \vee p_2 \vee \cdots \vee p_n$, which can be written as

$$\top \rightarrow p_1 \vee p_2 \vee \cdots \vee p_n.$$

If $n = 0$, then we have the clause $\neg q_1 \vee \neg q_2 \vee \cdots \vee \neg q_m$, which we can write as

$$q_1 \wedge q_2 \wedge \cdots \wedge q_m \rightarrow \bot.$$

If $m = n = 0$, then we have the empty clause $\top \rightarrow \bot$, which denotes a contradiction.

Thus, our goal of finding a contradiction is really about deriving an empty clause from a given set of clauses.

Note, however, that Module 4 only claimed that a clausal form exists for all **propositional** formulas. But what about predicate logic formulas?

With a bit of work, we can extend the result to predicate formulas as well. First note that, by applying established equivalences and renaming bound variables where needed, we can rewrite any predicate logic formula as an equivalent formula, in which all quantifiers lie at the outermost level:

$$\exists u \forall v \exists w \exists x \forall y \forall z \phi \quad \text{(for example)}$$

It remains to find a way to eliminate the quantifiers.

Eliminating existentials—Skolemization

Consider the variable u in our example formula,

$$\exists u \forall v \exists w \exists x \forall y \forall z \phi \ .$$

The formula states that there exists a value of u for which the rest of the formula holds. We don't know what value that is, but we do know that there is one. So let the constant u_0 denote that value. Then we can remove the existential and replace all occurrences of the variable u with the constant u_0:

$$\forall v \exists w \exists x \forall y \forall z \phi[u_0/u] \ .$$

The constant u_0 is called a **Skolem constant**. We saw examples of Skolem constants in the Alloy demos.

Now, what about the variable w? Here we should not expect that a single constant would suffice to represent w, as our choice of w may depend on the value of v, which is universally quantified outside w.

Since our choice of w may depend on v, we instead replace w with a function parameterized by v:

$$\forall v \exists x \forall y \forall z \phi[u_0/u][w_0(v)/w] \ ,$$

and similarly for x:

$$\forall v \forall y \forall z \phi[u_0/u][w_0(v)/w][x_0(v)/x] \ .$$

The functions w_0 and x_0 are called **Skolem functions**.

In this way, we can replace all existentially bound variables with either constants or functions. All that remain, then, are the universally bound variables.

Since all variables that remain in the formula are now universally quantified, we simply drop the quantifiers, and adopt the understanding that all variables we encounter are implicitly universally quantified. For example, our formula now becomes

$$\phi[u_0/u][w_0(v)/w][x_0(v)/x] \ ,$$

where the free variables v, y, and z are understood to be universally quantified.

The resulting formula is quantifier-free, and may be converted into a set of clauses.

Rewriting as clauses dramatically reduces the variety of formulas a system must be able to handle in searching for a contradiction.

We now need proof rules to tell us how we may manipulate clausal formulas in order to derive other formulas, and ultimately, contradictions. Our first observations are obvious: that in any clausal formula

$$(q_1 \wedge q_2 \wedge \cdots \wedge q_m) \longrightarrow (p_1 \vee p_2 \vee \cdots \vee p_n),$$

we may reorder the q_i's and/or reorder the p_j's without affecting the truth of the formula. Also repeated occurrences of the same q_i, or of the same p_j, within a given formula may be coalesced into one.

Next, since the variables in the clauses are understood to be universally quantified, we may replace them (consistently within a clause) with other subexpressions as needed.

With these freedoms understood, we actually need only a single deduction rule for clauses:

$$\frac{q_1 \wedge \cdots \wedge q_m \rightarrow p_1 \vee \cdots \vee p_n \vee r \qquad r \wedge q'_1 \wedge \cdots \wedge q'_{m'} \rightarrow p'_1 \vee \cdots \vee p'_{n'}}{q_1 \wedge \cdots \wedge q_m \wedge q'_1 \wedge \cdots \wedge q'_{m'} \rightarrow p_1 \vee \cdots \vee p_n \vee p'_1 \vee \cdots \vee p'_{n'}}$$

This rule is known as **resolution**.

The rule applies when there is a proposition r that appears on the left-hand side (the tail) of one clause and the right-hand side (the head) of another. The result of the rule consists of combining the two tails and the two heads, omitting only r.

The resolution rule is complete with respect to finding contradictions. In other words, if a set of clauses contains a contradiction, it can be found by repeated application of the resolution rule.

Thus, a system for finding contradictions in a set of formulas only needs to work over sets of clauses, and only needs a single rule, i.e., the resolution rule.

However, we can actually restrict ourselves even further, without affecting the computational power of the system. For a clause

$$(q_1 \wedge q_2 \wedge \cdots \wedge q_m) \rightarrow (p_1 \vee p_2 \vee \cdots \vee p_n),$$

we call the clause a **Horn clause** if $n \leq 1$. Thus, there are two kinds of Horn clauses: the **headed** Horn clause,

$$(q_1 \wedge q_2 \wedge \cdots \wedge q_m) \rightarrow p,$$

and the **headless** Horn clause,

$$(q_1 \wedge q_2 \wedge \cdots \wedge q_m) \rightarrow \bot .$$

By restricting our attention to Horn clauses, we can simplify the resolution rule:

$$\frac{q_1 \wedge \cdots \wedge q_m \rightarrow r \qquad r \wedge q'_1 \wedge \cdots \wedge q'_{m'} \rightarrow p}{q_1 \wedge \cdots \wedge q_m \wedge q'_1 \wedge \cdots \wedge q'_{m'} \rightarrow p},$$

where p is either an atomic proposition or $\bot$ (in which case the second Horn clause in the premises is headless).

Notice that the resolution rule cannot produce a headless Horn clause unless one of the Horn clauses in the premises is headless.

Therefore, a set of Horn clauses cannot derive a headless Horn clause by resolution, unless a headless Horn clause was present to begin with.

Then, since the empty clause $\top \rightarrow \bot$ is a headless Horn clause, it follows that a set of Horn clauses that does not contain a headless Horn clause **cannot** derive a contradiction.

Recall that our goal is to find values $y_1, \ldots, y_p$ such that

$$\phi_1, \ldots, \phi_m, \neg P(x_1, \ldots, x_n, y_1, \ldots, y_p) \vdash \bot \, ,$$

or, more precisely,

$$\phi_1, \ldots, \phi_m, \neg P(x_1, \ldots, x_n, y_1, \ldots, y_p) \vdash \top \rightarrow \bot \, .$$

Since a headless Horn clause is needed to derive a contradiction, it seems reasonable to expect that the negated formula $\neg P(x_1, \ldots, x_n, y_1, \ldots, y_p)$ will contribute one or more headless Horn clauses, whereas $\phi_1, \ldots, \phi_m$ will contribute only headed Horn clauses.

Example

Suppose we take as premises $\phi_1 = \forall x(Q(x) \to R(f(x)))$ and $\phi_2 = \forall y(R(y) \to Q(g(y)))$, where f and g are unary function symbols.

Now, suppose we take as our predicate $P(u, v)$ the formula $Q(u) \to Q(v)$, with free variables u and v. Suppose we are given $u = a$, where a is a constant, and we wish to find the corresponding value of v.

In other words, we are looking to prove

$$\phi_1, \phi_2 \vdash \exists v P(a, v) \; .$$

Rephrasing as a search for a contradiction gives

$$\phi_1, \phi_2, \forall v \neg P(a, v) \vdash \top \rightarrow \bot \, .$$

Replacing each formula with its definition gives

$$\forall x(Q(x) \rightarrow R(f(x))), \forall y(R(y) \rightarrow Q(g(y))), \forall v \neg(Q(a) \rightarrow Q(v)) \vdash \top \rightarrow \bot \, .$$

Since there are no existential variables, we don't have to worry about Skolemization, and we can simply drop the universal quantifiers:

$$Q(x) \rightarrow R(f(x)), R(y) \rightarrow Q(g(y)), \neg(Q(a) \rightarrow Q(v)) \vdash \top \rightarrow \bot \, .$$

The two premises, $Q(x) \rightarrow R(f(x))$ and $R(y) \rightarrow Q(g(y))$, are already clauses; it remains to convert $\neg(Q(a) \rightarrow Q(v))$ to clausal form.

The formula $\neg(Q(a) \to Q(v))$ is (classically) equivalent to $\neg(\neg Q(a) \vee Q(v))$, which is (classically) equivalent to $Q(a) \wedge \neg Q(v)$. From this formula, we obtain two clauses:

$$\top \to Q(a) \qquad Q(v) \to \bot\,.$$

In total, then, we have a set of four clauses,

$$Q(x) \to R(f(x)) \quad R(y) \to Q(g(y)) \quad \top \to Q(a) \quad Q(v) \to \bot\,,$$

and we want to derive $\top \to \bot$ from them via resolution.

$$Q(x) \to R(f(x)) \quad R(y) \to Q(g(y)) \quad \top \to Q(a) \quad Q(v) \to \bot\,,$$

If we specialize x to a, we can resolve the third clause with the first:

$$\frac{\top \to Q(a) \qquad Q(a) \to R(f(a))}{\top \to R(f(a))}$$

If we specialize v to $g(y)$, we can resolve the second clause with the fourth:

$$\frac{R(y) \to Q(g(y)) \qquad Q(g(y)) \to \bot}{R(y) \to \bot}$$

Specializing y to $f(a)$, we can now resolve these two clauses:

$$\frac{\top \to R(f(a)) \qquad R(f(a)) \to \bot}{\top \to \bot}$$

We have arrived at a contradiction. In order to reach the contradiction, we had to specialize v to $g(y)$ and y to $f(a)$. Hence, v was specialized to $g(f(a))$, and therefore, $g(f(a))$ is the value of v that satisfies $P(u, v)$ for $u = a$.

In other words, we now have a proof of the sequent

$$\phi_1, \phi_2 \vdash \exists v P(a, v) \ ,$$

established by setting $v = g(f(a))$. In particular, we have shown that

$$\phi_1, \phi_2 \vdash P(a, g(f(a))) \ .$$

Thus, we have succeeded in producing the output from the input in a somewhat systematic way.

We glossed over how we chose the clauses to which we would apply the resolution rule. We must formalize this process as well.

Why? For at least two reasons.

First, it may be the case that there are **no** values $y_1, \ldots, y_p$ such that $\phi_1, \ldots, \phi_m \vdash P(x_1, \ldots, x_n, y_1, \ldots, y_p)$. In this case, we would like to be able to report that the problem has no solution, which we can only do if we know we have exhausted the search space. This requires that the search space be traversed in a systematic way.

Second, there may be **multiple** sets of values $y_1, \ldots, y_p$ that solve the problem (not possible if P denotes a function, but certainly possible in the general case). Then we would like to be able to report **all** sets of satisfying values. Again, this requires systematic search.

Search strategy for resolution

We begin with the following observation: we cannot perform resolution on two headless Horn clauses. (Why not?)

Thus, each contradiction must arise from resolving some number of headed Horn clauses and **exactly one** headless Horn clause.

This suggests that we should separate the headed Horn clauses from the headless Horn clauses. Let H (for “Headed”) denote the set of headed Horn clauses, and let L (for “headLess”) denote the set of headless Horn clauses.

Partitioning the clauses in this way gives rise to the following framework for finding contradictions:

for each $l \in L$

    for each contradiction in $H \cup \{l\}$

        report the contradiction and the corresponding assignment to y_i's

In essence, we are looping over all sets of clauses that contain exactly one headless Horn clause.

But we still need to formalize how we find contradictions in $H \cup \{l\}$ for each $l \in L$.

Label the clauses in H with numbers $1, \ldots, k$. Let h_k denote the clause in H with label k.

We use a backtracking search technique to find clauses that can be resolved against l. Note that the result of resolving l with any clause in H will always be a new headless Horn clause.

Backtracking search for solutions

function findMatches(H, k, l)

Matches := $\{\}$

for j = 1 .. k

if l and h_j cannot be resolved, go to next loop iteration

l' := result of resolving l and h_j

if $l' = \top \rightarrow \bot$

Matches := Matches $\cup$ (y_i assignments that created l')

else

Matches := Matches $\cup$ findMatches(H, k, l')

return Matches

A few notes on the backtracking search function:

- the backtracking nature of the search algorithm is expressed by going to the next loop iteration after returning from the recursive call to findMatch;
- the search is “depth-first”—we recurse “down” the search tree before trying other alternatives at the same depth (i.e., other values of the loop counter at the same recursion depth);
- the depth-first algorithm may fall into an infinite recursion before all solutions have been found;
- “breadth-first” search (i.e., trying all possible loop values before recursing) is also possible, and will not infinitely recurse before all solutions are found, but requires additional memory.

Also note that, because the search algorithm accumulates all answers and returns them once the search space is exhausted, if it falls into an infinite recursion, it will never return anything at all!

A better solution would be for the function to return a single solution as soon as it is found and then “pick up where it left off” the next time it is called, in order to return the next solution in the search space.

This kind of behaviour is achievable, but is more difficult to implement, so we leave it as an enrichment exercise.

Returning to our previous example,

$$Q(x) \to R(f(x)), R(y) \to Q(g(y)), \neg(Q(a) \to Q(v)) \vdash \top \to \bot\,,$$

we can now solve the problem more systematically. We have

$$H = \{h_1 : Q(x) \to R(f(x)), h_2 : R(y) \to Q(g(y)), h_3 : \top \to Q(a)\}\,,$$

and

$$L = \{l : Q(v) \to \bot\}\,.$$

We now evaluate findMatch(H, 3, l). Since l and h_1 cannot be resolved, we go to h_2. These resolve to produce $l' = R(y) \to \bot$, where $v = g(y)$.

Next, we evaluate findMatch(H, 3, l'). l' and h_1 resolve to $l'' = Q(x) \to \bot$, where $y = f(x)$.

We then evaluate findMatch(H, 3, l''). Since l'' does not resolve with h_1 or h_2, we resolve l'' with h_3, to produce $\top \rightarrow \bot$, with $x = a$. Thus, we have a solution with $v = g(f(a))$.

Returning from the recursive call, we now attempt to resolve l' against h_2 and h_3; both attempts fail.

Returning from the previous recursive call, we attempt to resolve l against h_3; this also fails.

The function now terminates, having found a single output: $v = g(f(a))$.

A more practical example

Let us use the nullary function symbol e to denote an empty list, and the binary function symbol application $c(h, t)$ to denote the list with head h and tail (i.e., rest of the elements) t.

For example, the three-element list $[1, 2, 3]$ would look like $c(1, c(2, c(3, e)))$.

Let us consider writing a predicate that appends two lists.

We want a ternary predicate $A(x, y, z)$ that is interpreted as "z is the result of appending y onto the end of x."

Then, if we feed in x and y as input parameters, we will hopefully be able to obtain as output the result of appending them, through the output parameter z.

What do we know about A?

First, if x is empty, then $z = y$:

$$\phi_1 : \forall z A(e, z, z) \ .$$

If $x = c(u, v)$, we can obtain the result by appending y to v and then adding the element u:

$$\phi_2 : \forall u \forall v \forall y \forall z (A(v, y, z) \rightarrow A(c(u, v), y, c(u, z))) \ .$$

We can use these "facts" as our premises, and now consider a specific call to append two lists:

$$\phi_1, \phi_2 \vdash A(c(1, c(2, e)), c(3, c(4, e)), r) \ ,$$

where we seek a value r that makes this sequent valid.

Rephrasing as a search for a contradiction gives

$$\phi_1, \phi_2, \neg A(c(1, c(2, e)), c(3, c(4, e)), r) \vdash \top \rightarrow \bot \, .$$

The premises ϕ_1 and ϕ_2 become clauses simply by dropping the quantifiers (and prepending "$\top \rightarrow$" to ϕ_1). The goal formula becomes the headless Horn clause

$$A(c(1, c(2, e)), c(3, c(4, e)), r) \rightarrow \bot \, .$$

Thus, we have

$$H = \{h_1 : \top \rightarrow A(e, z, z), h_2 : A(v, y, z) \rightarrow A(c(u, v), y, c(u, z))\}$$

and

$$L = \{l : A(c(1, c(2, e)), c(3, c(4, e)), r) \rightarrow \bot\} \, .$$

Since l does not resolve against h_1, we resolve against h_2, obtaining

$$u = 1 \quad v = c(2, e) \quad y = c(3, c(4, e)), r = c(u, z) = c(1, z) \ ,$$

$$l' = A(v, y, z) \to \bot = A(c(2, e), c(3, c(4, e)), z) \to \bot \ .$$

Again, l' does not resolve against h_1, so we resolve against h_2. To keep the two instances of h_2 used in the resolution separate, we first reinstantiate h_2 with fresh variables:

$$h_2 : A(v_1, y_1, z_1) \to A(c(u_1, v_1), y_1, c(u_1, z_1)) \ .$$

Then we get

$$u_1 = 2 \quad v_1 = e \quad y_1 = c(3, c(4, e)) \quad z = c(u_1, z_1) = c(2, z_1) \ ,$$

$$l'' = A(v_1, y_1, z_1) \to \bot = A(e, c(3, c(4, e)), z_1) \to \bot \ .$$

Now, l'' resolves against h_1, which we first rewrite to avoid name clashes:

$$h_1 : \top \to A(e, z_2, z_2) \; .$$

Resolving against

$$l'' = A(v_1, y_1, z_1) \to \bot = A(e, c(3, c(4, e)), z_1) \to \bot \; .$$

gives

$$z_2 = c(3, c(4, e)) \quad z_1 = z_2 = c(3, c(4, e)) \; ,$$

$$l''' = \top \to \bot \; ,$$

and we have our contradiction. Tracing through our substitutions, we have

$$r = c(1, z) \quad z = c(2, z_1) \quad z_1 = c(3, c(4, e)) \; .$$

Hence, $r = c(1, c(2, c(3, c(4, e))))$, and we see that we have automatically computed the result of appending the two original lists.

A logic programming language

The programming language Prolog is an almost direct implementation of programming with Horn clauses.

Prolog (from “PROgrammation en LOGique”) was invented in 1972 by Alain Colmerauer and Philippe Roussel.

Prolog programs consist of a set of premises, written as headed Horn clauses, that form the **database**, and a user-supplied **query**, written as a headless Horn clause.

Prolog attempts to satisfy the query, given the premises, by employing a depth-first resolution-based search for a contradiction in the database, just as in our development on previous slides.

Prolog syntax

The Horn clause

$$(q_1 \wedge \cdots \wedge q_m) \rightarrow p$$

is represented in Prolog as

```
p :- q_1, ..., q_m.
```

and read as "p if $q_1, \ldots, q_m$." The Horn clauses that make up the database are read in from a file; the headless Horn clause that constitutes the query is entered interactively.

Variables are distinguished from atoms (i.e., constants) in Prolog by case: variable names always start with an uppercase letter, while the names of constants do not.

Example

Let us prove the following classical deduction:

“All men are mortal. Socrates is a man. Therefore, Socrates is mortal.”

The first two sentences above are premises, and will form the database. The third sentence is the conclusion, and will form the query. We encode the two premises in Prolog as follows:

```
mortal(X) :- man(X).
man(socrates).
```

We may then ask the system whether Socrates is mortal:

```
?- mortal(socrates).
```

Prolog responds:

```
Yes.
```

The answer came by resolving the headless Horn clause

```
:- mortal(socrates)
```

against the clauses in the database. Note that no variables had to be instantiated in this example. Prolog simply reports that the negation of the statement that Socrates is mortal leads to a contradiction.

If the query contains variables, then Prolog reports the values of those variables that lead to a contradiction. For example, we might query “Who is mortal?”:

```
?- mortal(X).
```

Prolog responds:

```
X=socrates
```

indicating that taking `X` equal to `socrates` creates a contradiction with the database.

If we assert that Aristotle is also a man:

```
mortal(X) :- man(X).
man(socrates).
man(aristotle).
```

then the query

```
?- mortal(X).
```

now has two solutions. Prolog responds

```
X=socrates
```

If you hit ENTER, you get the query prompt back. If you hit semicolon (;), then Prolog generates the next solution:

```
X=aristotle
```

Example with lists

We represent the empty list in Prolog as `[]`. The list with head `H` and tail `T` is written as `[H|T]`. Then we can implement the append operation for lists in Prolog as follows:

```
append([], Z, Z).
append([U|X], Y, [U|Z]) :- append(X, Y, Z).
```

If we query

```
?- append([1,2], [3,4], Z).
```

then Prolog responds

```
Z=[1,2,3,4]
```

We noted earlier that there is no particular need to regard the last argument of a predicate as the output argument. We can illustrate this fact here by reversing the roles of the parameters and asking, "What two lists append to give `[1,2,3,4]`?":

```
?- append(X, Y, [1,2,3,4]).
```

Prolog responds (with successive inputs of ; from the user)

```
X=[], Y=[1,2,3,4]
X=[1], Y=[2,3,4]
X=[1,2], Y=[3,4]
X=[1,2,3], Y=[4]
X=[1,2,3,4], Y=[]
```

We can take advantage of our ability (sometimes) to reverse the roles of arguments in implementing a function to split a list into two equal pieces (rejecting odd-length lists). We first write a predicate for length:

```
len([], 0).
len([_|Y], N) :- len(Y, M), N is M + 1.
```

The variable _ indicates that we don't care about its value; the infix predicate `is` is used for performing mathematical calculations. We can then split a list evenly as follows:

```
evenSplit(X, Y, Z) :- append(Y, Z, X), len(Y, N), len(Z, N).
```

```
?- evenSplit([1,2,3,4], Y, Z).
```

```
Y=[1,2], Z=[3,4]
```

Final example: reversing a list

The following Prolog rules encode reversing a list:

```
reverse([], []).
reverse([X|Y], Z) :- reverse(Y, YR), append(YR, [X], Z).

?- reverse([1,2,3], Z).

Z=[3,2,1]
```

Differences between Prolog and pure logic programming

Prolog is much more restrictive than pure logic programming in terms of how it allows clauses to be manipulated. In particular, Prolog will not reorder the atoms in a clause in order to create a match for resolution; it requires that the pieces of a query be resolved in the order in which they occur.

Also, Prolog will not allow multiple occurrences of the same atom in a clause or query to be coalesced into one. Rather, if an atom appears more than once, it must be resolved more than once.

This behaviour, though somewhat redundant, allows Prolog to behave in a more predictable way, which is important in a programming environment, particularly when predicates whose evaluation induces side effects (such as I/O) are involved. For example,

```
?- display(hello), display(world).
```

is very different from

```
?- display(world), display(hello).
```

Goals of this module

You will not be tested explicitly on logic programming or Prolog. However, the ideas in this module draw from several other modules in this course; thus, it is worthwhile studying this material, as it will help to solidify your understanding of some of the other topics we have covered.

SWI Prolog is installed on the undergrad machines. Type `pl` to run it and `^D` to quit. Load a database file with

`?- consult(filename).` or more concisely,

`?- [filename].`