Logic Programming

1 Introduction

We will investigate the connection between logic and computation from the perspective that *programs are logical theories* and that *computation is proof search*. In essence, we will turn first-order logic into a programming language. To see what this looks like, let's look at a simple example: graph coloring. The graph coloring problem is stated as follows:

> **Graph Coloring Problem.** Assume that you are given k colors and an undirected graph. The graph is in the form of two predicates: a predicate $node/1$ such that $node(x)$ indicates that x is a node and a predicate $edge/2$ such that $edge(x_1, x_2)$ indicates the presence of an edge between nodes x_1 and x_2. Your task is to assign one of the k colors to each node of the graph such that no pair of adjacent nodes have the same color.

This seems like a fairly abstract problem without any real application, but it turns out that its abstract nature makes it applicable to many real-world problems. We can think of the graph as representing some set of resources and constraints on those resources. For example, suppose that we have n jobs that need to be done, and we need to schedule them in a way that completes those jobs in minimal time. However, some of the jobs conflict with each other and can't be scheduled at the same time. We can make a graph that has one node for each job and that has an edge between two nodes if the corresponding jobs have a conflict. Then the minimal k such that the graph can be correctly colored turns out to be the optimal time to complete all of the jobs.

As a more concrete example, consider *register allocation*, a critical step in compiling programs. A computer can only operate on values that are in registers, but there are only a finite number k of registers available. Suppose that there are n variables in a program; generally n will be much greater than k. The compiler must insert code to load the variable values from memory into registers when they are needed, but it should do so in an optimal way that makes as few requests to memory as possible. We can again encode this as a graph coloring problem. Let the graph contain one node for each variable. Let there be an edge between two nodes if the corresponding variable are both *live* at the same time (i.e., the program needs to operate on both values at the same time). Then the graph coloring problem says to assign one of the k registers to each variable such that if both variables need to be in registers at the same time (thus there is an edge between them in the graph), they cannot be assigned the same register.

So how do we solve the graph coloring problem? One way is to encode the problem in terms of first-order logic. Our encoding will express the constraints of the problem as logical formulae:

> **Graph Coloring Encoding.** Let predicate $color/2$ indicate the color of a node, e.g., if node 1 is blue then $color(1, blue)$. Let predicate $equals/2$ indicate equality of colors. Then:
>
> - Every node has a color: $\forall x.node(x) \Rightarrow \exists c.color(x, c)$.
> - Every node has at most one color: $\forall x \forall c_1 \forall c_2.color(x, c_1) \wedge color(x, c_2) \Rightarrow equals(c_1, c_2)$.
> - Neighboring nodes have different colors: $\forall x_1 \forall x_2.edge(x_1, x_2) \Rightarrow \neg \exists c.color(x_1, c) \wedge color(x_2, c)$.

So, given a particular graph encoded by the *node* and *edge* predicates and the coloring problem encoded by the formulae above, we can ask the following question: what are the values of the *color* predicate that **satisfy** this encoding? Recall that in first-order logic, a formula is satisfiable if there is some way to make the formula true. Here, we are asking not only if there exists some way to make it true, but what exactly that solution is. Our encoding of the graph and of the coloring problem forms a *logical theory*; in order to determine a satisfying solution, we need to perform a *proof search*. Or, in another way of thinking about it: that logical theory is a *program*, and the proof search is a *computation*.

As we see from the above example, programs in a logic programming language take a very different form than programs in C, C++, Java, etc. Rather than giving a series of instructions that describe step by step how to compute a solution, we instead describe what a solution *looks like* and let the logic language implementation find a solution that matches our description. The former style is called **imperative programming** and the latter style is called **declarative programming**. The reality is that the imperative and

declarative styles of programming form more of a spectrum of programming styles rather than a binary either/or difference, but logic programming is pretty far on the declarative side of things.

Because most programmers learn the imperative programming style first, when they encounter declarative programming it seems very strange and difficult. It can be very hard to wrap their heads around what it means and how it works. However, there are problems that are very straightforward and simple to solve declaratively that would be extremely difficult and complex to solve imperatively. In fact, for these kinds of problems the imperative solution would essentially be an implementation of a declarative language on top of the imperative language being used. Declarative programming (and specifically logic programming) is used in a number of places in industry, such as civil engineering, mechanical engineering, digital circuit verification, automated timetabling, air traffic control, finance, and more.

2 Logic Programs

In this section we will look more closely at how to write logic programs using logical formulae.

2.1 Normal Form

Logic programs consist of logical formulae that encode some property of interest (e.g., the graph and the coloring problem in the example from the previous section). To simplify things, it is convenient to use a *normal form* to express those formulae. We'll use something called *clausal normal form*, which is very similar to the standard *conjunctive normal form* used in propositional logic.

A **literal** is either a predicate, a variable, a negated predicate, or a negated variable. If the literal is a negated predicate or variable it is called a **negative literal**, otherwise it is called a **positive literal**. A formula in clausal normal form has the following form:

$$(\forall X_1, \ldots, X_i \,.\, A_1 \wedge \ldots \wedge A_j \Rightarrow B_1 \vee \ldots \vee B_k) \wedge \ldots \wedge (\forall Y_1, \ldots, Y_p \,.\, C_1 \wedge \ldots \wedge C_q \Rightarrow D_1 \vee \ldots \vee D_r)$$

where the As, Bs, Cs, and Ds are positive literals. In other words, it is a conjunction of clauses such that (1) each clause has all universal quantifiers on the outside, and (2) the body of the clause is an implication with a conjunction of positive literals on the left and a disjunction of positive literals on the right. Note that this is basically conjunctive normal form except that we have the universal quantifiers and we're using the identity $\neg A \vee B \equiv A \Rightarrow B$. So, for example, we could have written the first clause above as $\forall X_1, \ldots, X_i \,.\, \neg A_1 \vee \ldots \vee \neg A_j \vee B_1 \vee \ldots \vee B_k$. We can take any formula in first-order logic and transform it into an equivalent formula (in terms of satisfiability) in clausal normal form. There is a mechanical procedure for doing so, but we won't bother going through it here. When we're doing logic programming, common practice is for the programmer to write the formulae in normal form themselves instead of writing arbitrary formulae and then transforming them into normal form.

When looking at a formula in clausal normal form, think of it as saying that *if* we can prove all of A_1 through A_j, *then* we know that at least one of B_1 through B_k must be true; and similarly for all the other clauses. Here are a couple of examples of formula in clausal normal form:

- $\forall X, Y \,.\, p(X) \wedge q(Y) \Rightarrow r(X, Y) \vee s(X, Y)$
- $\forall X, Y, Z \,.\, p(X, Y) \wedge p(Y, Z) \Rightarrow p(X, Z)$

2.2 Horn Clauses

When implementing logic programming, we run into a fundamental problem: computing satisfiability for the full first-order logic is only *semi-decidable* (i.e., if a formula is satisfiable we can figure that out in finite time, but if it is unsatisfiable we may end up waiting forever). This makes logic programming somewhat impractical, and therefore we need to restrict the problem, sacrificing expressiveness for tractability. The basic problem is the disjunctions on the right side of the clause bodies. Given that the left side of the implication is true, we know that *at least one* of the things on the right of the implication is true—but we don't know which ones. For example, if we're trying to prove B_1 and manage to prove all of A_1 through A_j, we still don't know if we've proven B_1 or if we've proven one of the other literals B_2 through B_k.

The solution is to restrict the clauses so that there must be exactly one literal on the right of every implication. Formulae in clausal normal form that meet this restriction are called **definite Horn clauses**. In the two examples of formulae in clausal normal form given above, the first one is *not* a definite Horn clause but the second one *is* a definite Horn clause. This simple change has a profound impact; it makes logic programming *goal-directed*. This means that, given a specific predicate P we want to prove, we just need to look for those clauses that have P on the right-hand side of their implication. When we find such a clause, we now have new set of goals: for each predicate Q on the left-hand side of the implication, we look for those clauses that have Q on the right-hand side. We recursively follow this strategy until we've reduced the problem to a set of predicates that are all trivially true (i.e., they are *axioms*) or we've found a predicate that cannot be proved (in which case the original formula is unsatisfiable). Later, we'll see how to turn this basic strategy into an implementation of an interpreter for logic programming.