

Logic Languages

11.3 Theoretical Foundations

In mathematical logic, a *predicate* is a function that maps constants (atoms) or variables to the values true and false. *Predicate calculus* provides a notation and inference rules for constructing and reasoning about *propositions* (*statements*) composed of predicate applications, *operators*, and the *quantifiers* \forall and \exists .¹ Operators include and (\wedge), or (\vee), not (\neg), implication (\rightarrow), and equivalence (\leftrightarrow). Quantifiers are used to introduce bound variables in an appended proposition, much as λ introduces variables in the lambda calculus. The *universal* quantifier, \forall , indicates that the proposition is true for all values of the variable. The *existential* quantifier, \exists , indicates that the proposition is true for at least one value of the variable. Here are a few examples:

EXAMPLE 11.39

Propositions

$$\forall C[\text{rainy}(C) \wedge \text{cold}(C) \rightarrow \text{snowy}(C)]$$

(For all cities C , if C is rainy and C is cold, then C is snowy.)

$$\forall A, \forall B[(\exists C[\text{takes}(A, C) \wedge \text{takes}(B, C)]) \rightarrow \text{classmates}(A, B)]$$

(For all students A and B , if there exists a class C such that A takes C and B takes C , then A and B are classmates.)

$$\forall N[(N > 2) \rightarrow \neg(\exists A, \exists B, \exists C[A^N + B^N = C^N])]$$

(This is Fermat's last theorem.)

¹ Strictly speaking, what we are describing here is the *first-order* predicate calculus. There exist higher-order calculi in which predicates can be applied to predicates, not just to atoms and variables. Prolog allows the user to construct higher-order predicates using `call`; the formalization of such predicates is beyond the scope of this book.

EXAMPLE 11.40

Different ways to say things

One of the interesting characteristics of predicate calculus is that there are many ways to say the same thing. For example,

$$\begin{aligned}(P_1 \rightarrow P_2) &\equiv (\neg P_1 \vee P_2) \\ (\neg \exists X[P(X)]) &\equiv (\forall X[\neg P(X)]) \\ \neg(P_1 \wedge P_2) &\equiv (\neg P_1 \vee \neg P_2)\end{aligned}$$

This flexibility of expression tends to be handy for human beings, but it can be a nuisance for automatic theorem proving. Propositions are much easier to manipulate algorithmically if they are placed in some sort of *normal form*. One popular candidate is known as *clausal form*. We consider this form below. ■

11.3.1 Clausal Form

As it turns out, clausal form is very closely related to the structure of Prolog programs: once we have a proposition in clausal form, it will be relatively easy to translate it into Prolog. We should note at the outset, however, that the translation is not perfect: there are aspects of predicate calculus that Prolog cannot capture, and there are aspects of Prolog (e.g., its imperative and database-manipulating features) that have no analogues in predicate calculus.

Clocksin and Mellish [CM03, Chap. 10] describe a five-step procedure (based heavily on an article by Martin Davis [Dav63]) to translate an arbitrary first-order predicate proposition into clausal form. We trace that procedure here.

EXAMPLE 11.41

Conversion to clausal form

In the first step, we eliminate implication and equivalence operators. As a concrete example, the proposition

$$\forall A[\neg \text{student}(A) \rightarrow (\neg \text{dorm_resident}(A) \wedge \neg \exists B[\text{takes}(A, B) \wedge \text{class}(B)])]$$

would become

$$\forall A[\text{student}(A) \vee (\neg \text{dorm_resident}(A) \wedge \neg \exists B[\text{takes}(A, B) \wedge \text{class}(B)])]$$

In the second step, we move negation inward, so that the only negated items are individual terms (predicates applied to arguments):

$$\begin{aligned}&\forall A[\text{student}(A) \vee (\neg \text{dorm_resident}(A) \wedge \forall B[\neg(\text{takes}(A, B) \wedge \text{class}(B))])] \\ \equiv &\forall A[\text{student}(A) \vee (\neg \text{dorm_resident}(A) \wedge \forall B[\neg \text{takes}(A, B) \vee \neg \text{class}(B)])]\end{aligned}$$

In the third step, we use a technique known as Skolemization (due to logician Thoralf Skolem) to eliminate existential quantifiers. We will consider this technique further in Section ©11.3.3. Our example has no existential quantifiers at this stage, so we proceed.

In the fourth step, we move all universal quantifiers to the outside of the proposition (in the absence of naming conflicts, this does not change the proposition's

meaning). We then adopt the convention that all variables are universally quantified, and drop the explicit quantifiers:

$$\text{student}(A) \vee (\neg \text{dorm_resident}(A) \wedge (\neg \text{takes}(A, B) \vee \neg \text{class}(B)))$$

Finally, in the fifth step, we use the distributive, associative, and commutative rules of Boolean algebra to convert the proposition to *conjunctive normal form*, in which the operators \wedge and \vee are nested no more than two levels deep, with \wedge on the outside and \vee on the inside:

$$(\text{student}(A) \vee \neg \text{dorm_resident}(A)) \wedge (\text{student}(A) \vee \neg \text{takes}(A, B) \vee \neg \text{class}(B))$$

Our proposition is now in clausal form. Specifically, it is in conjunctive normal form, with negation only of individual terms, with no existential quantifiers, and with implied universal quantifiers for all variables (i.e., for all names that are neither constants nor predicates). The clauses are the items at the outer level: the things that are and-ed together. ■

EXAMPLE 11.42

Conversion to Prolog

To translate the proposition to Prolog, we convert each logical clause to a Prolog fact or rule. Within each clause, we use commutativity to move the negated terms to the right and the non-negated terms to the left (our example is already in this form). We then note that we can recast the disjunctions as implications:

$$\begin{aligned} & (\text{student}(A) \leftarrow \neg(\neg \text{dorm_resident}(A))) \\ & \quad \wedge (\text{student}(A) \leftarrow \neg(\neg \text{takes}(A, B) \vee \neg \text{class}(B))) \\ \equiv & (\text{student}(A) \leftarrow \text{dorm_resident}(A)) \\ & \quad \wedge (\text{student}(A) \leftarrow (\text{takes}(A, B) \wedge \text{class}(B))) \end{aligned}$$

These are Horn clauses. The translation to Prolog is trivial:

```
student(A) :- dorm_resident(A).
student(A) :- takes(A, B), class(B).
```

■

11.3.2 Limitations

We claimed at the beginning of Section 11.1 that Horn clauses could be used to capture most, though not all, of first-order predicate calculus. So what is it missing? What can go wrong in the translation? The answer has to do with the number of non-negated terms in each clause. If a clause has more than one, then if we attempt to cast it as an implication there will be a disjunction on the left-hand side of the \leftarrow symbol, something that isn't allowed in a Horn clause. Similarly, if we end up with no non-negated terms, then the result is a headless Horn clause, something that Prolog allows only as a query, not as an element of the database.

As an example of a disjunctive head, consider the statement “every living thing is an animal or a plant.” In clausal form, we can capture this as

$$\text{animal}(X) \vee \text{plant}(X) \vee \neg \text{living}(X)$$

EXAMPLE 11.43

Disjunctive left-hand side

or equivalently

$$\text{animal}(X) \vee \text{plant}(X) \leftarrow \text{living}(X)$$

Because we are restricted to a single term on the left-hand side of a rule, the closest we can come to this in Prolog is

```
animal(X) :- living(X), \+(plant(X)).
plant(X)  :- living(X), \+(animal(X)).
```

But this is not the same, because Prolog's $\backslash +$ indicates inability to prove, not falsehood. ■

EXAMPLE 11.44

Empty left-hand side

As an example of an empty head, consider Fermat's last theorem (Example ©11.39). Abstracting out the math, we might write

$$\forall N[\text{big}(N) \rightarrow \neg(\exists A, \exists B, \exists C[\text{works}(A, B, C, N)])]$$

which becomes the following in clausal form:

$$\neg \text{big}(N) \vee \neg \text{works}(A, B, C, N)$$

We can couch this as a Prolog query:

```
?- big(N), works(A, B, C, N).
```

(a query that will never terminate), but we cannot express it as a fact or a rule. ■

EXAMPLE 11.45

Theorem proving as a search for contradiction

The careful reader may have noticed that facts are entered on the left-hand side of an (implied) Prolog $:-$ sign:

```
rainy(rochester).
```

while queries are entered on the right:

```
?- rainy(rochester).
```

The former means

$$\text{rainy}(\text{rochester}) \leftarrow \text{true}$$

The latter means

$$\text{false} \leftarrow \text{rainy}(\text{rochester})$$

If we apply resolution to these two propositions, we end up with the contradiction

$$\text{false} \leftarrow \text{true}$$

This observation suggests a mechanism for automated theorem proving: if we are given a collection of axioms and we want to prove a theorem, we temporarily add the *negation* of the theorem to the database and then attempt, through a series of resolution operations, to obtain a contradiction. ■

11.3.3 Skolemization

EXAMPLE 11.46

Skolem constants

In Example ©11.41 we were able to translate a proposition from predicate calculus into clausal form without worrying about existential quantifiers. But what about a statement like this one:

$$\exists X[\text{takes}(X, \text{cs254}) \wedge \text{class_year}(X, 2)]$$

(There is at least one sophomore in cs254.) To get rid of the existential quantifier, we can introduce a *Skolem constant* x :

$$\text{takes}(x, \text{cs254}), \text{class_year}(x, 2)$$

The mathematical justification for this change is based on something called the *axiom of choice*; intuitively, we say that if there exists an X that makes the statement true, then we can simply pick one, name it x , and proceed. (If there does not exist an X that makes the statement true, then we can choose some arbitrary x , and the statement will still be false.) It is worth noting that Skolem constants are not necessarily distinct; it is quite possible, for example, for x to name the same student as some other constant y that represents a sophomore in his201. ■

Sometimes we can replace an existentially quantified variable with an arbitrary constant x . Often, however, we are constrained by some surrounding universal quantifier. Consider the following example:

EXAMPLE 11.47

Skolem functions

$$\forall X[\neg \text{dorm_resident}(X) \vee \exists A[\text{campus_address_of}(X, A)]]$$

(Every dorm resident has a campus address.) To get rid of the existential quantifier, we must choose an address for X . Since we don't know who X is (this is a general statement about all dorm residents), we must choose an address that *depends on* X :

$$\forall X[\neg \text{dorm_resident}(X) \vee \text{campus_address_of}(X, f(X))]$$

Here f is a *Skolem function*. If we used a simple Skolem constant instead, we'd be saying that there exists some single address shared by all dorm residents. ■

Whether Skolemization results in a clausal form that we can translate into Prolog depends on whether we need to know what the constant is. If we are using predicates `takes` and `class_year`, and we wish to assert as a fact that there is a sophomore in cs254, we can write:

EXAMPLE 11.48

Limitations of
Skolemization

```
takes(the_distinguished_sophomore_in_254, cs254).
class_year(the_distinguished_sophomore_in_254, 2).
```

Similarly, we can assert that every dorm resident has a campus address by writing:

```
campus_address_of(X, the_dorm_address_of(X)) :- dorm_resident(X).
```

Now we can search for classes with sophomores in them:

```
sophomore_class(C) :- takes(X, C), class_year(X, 2).
?- sophomore_class(C).
C = cs254
```

and we can search for people with campus addresses:

```
has_campus_address(X) :- campus_address_of(X, Y).
dorm_resident(li_ying).
?- has_campus_address(X).
X = li_ying
```

Unfortunately, we won't be able to identify a sophomore in cs254 by name, nor will we be able to identify the address of li_ying. ■

✓ CHECK YOUR UNDERSTANDING

15. Define the notion of *clausal form* in predicate calculus.
16. Outline the procedure to convert an arbitrary predicate calculus statement into clausal form.
17. Characterize the statements in clausal form that cannot be captured in Prolog.
18. What is *Skolemization*? Explain the difference between Skolem constants and Skolem functions.
19. Under what circumstances may Skolemization fail to produce a clausal form that can be captured usefully in Prolog?