Semantics for predicate logic

Readings: Section 2.4, 2.5, 2.6.

In this module, we will precisely define the semantic interpretation of formulas in our predicate logic. In propositional logic, every formula had a fixed, finite number of models (valuations); this is not the case in predicate logic. As a consequence, we must take more care in defining notions such as satisfiability and validity, and we will see that there cannot be algorithms to decide if these properties hold or not for a given formula.

Models (2.4.1)

In the semantics of propositional logic, we assigned a truth value to each atom. In predicate logic, the smallest unit to which we can assign a truth value is a predicate $P(t_1, t_2, \ldots, t_n)$ applied to terms.

But we cannot arbitrarily assign a truth value, as we did for propositional atoms. There needs to be some consistency.

We need to assign values to variables in appropriate contexts, and meanings to functions and predicates. Intuitively, this is straightforward, but we must define such things precisely in order to ensure consistency of interpretation.

Here is the precise definition of a model $\mathcal{M}$ for a given set of predicate symbols $\mathcal{P}$ and function symbols $\mathcal{F}$:

1. A nonempty set $A^{\mathcal{M}}$ (the universe of concrete values);
2. A concrete element $f^{\mathcal{M}}$ of A for every nullary function symbol (constant) $f \in \mathcal{F}$;
3. A concrete function $f^{\mathcal{M}} : A^n \to A$ for every n-ary function symbol $f \in \mathcal{F}$;
4. A subset $P^{\mathcal{M}}$ of n-tuples over A for every n-ary predicate symbol $P \in \mathcal{P}$.

The textbook leaves off the superscript $\mathcal{M}$ on $A^{\mathcal{M}}$, which is unambiguous if the model is clear.

We will have to define what it means for a model $\mathcal{M}$ to make a formula ϕ true. We lead up to this with some intuitive examples.

Consider the formula $\phi = \exists x(P(x) \wedge Q(x, c))$.

If we let $A^{\mathcal{M}} = \mathbb{R}$, $P^{\mathcal{M}} = \{(x) \mid x \text{ rational}\}$, $Q^{\mathcal{M}} = \{(x, y) \mid x > y\}$, $c^{\mathcal{M}} = 0$, then intuitively ϕ says "There is a real number which is both rational and positive", and it is true.

On the other hand, if we change the definition of $P^{\mathcal{M}}$ to be $P^{\mathcal{M}} = \{(x) \mid x < 0\}$, then intuitively ϕ says "There is a real number which is both negative and positive", which is false.

Our universe need not consist of numbers; it could consist of other types of elements, such as strings over an alphabet.

The satisfaction relation

We would like to specify what it means for a model to satisfy a formula, or equivalently, that a formula is assigned the value true by a given model. As with propositional logic, this is a recursive definition on the structure of formulas.

There is one hitch, however. Consider trying to assign a truth value to the formula $\forall x P(x)$ in a model with universe $A^{\mathcal{M}} = \{a, b, c\}$. Intuitively, we need to substitute the values a, b, c into the formula $P(x)$ and see if they make it true.

But we have no way yet of talking about such substitutions. While $P(x)$ is a formula, something like $P(a)$ does not fit our definition of what a formula is.

The solution is to attach an **environment** (or look-up table) to the formula $P(x)$ which defines the value of x. Our notion of the value of a formula in a model is thus relative to an environment. In general, an environment is a function $\ell : var \to A^{\mathcal{M}}$.

In our definitions, we will need to modify environments, and so we define $\ell[x \mapsto a]$ as the environment that maps x to a but otherwise behaves as ℓ.

We are now ready to define what it means for a formula to be satisfied by a model relative to an environment. Given a model $\mathcal{M}$ for $(\mathcal{F}, \mathcal{P})$ and an environment ℓ, we say $\mathcal{M} \models_\ell \phi$ (read "$\mathcal{M}$ models ϕ with environment ℓ) if the following holds. (You will see that the rules that follow match the clauses in the recursive definition of a formula in predicate logic.)

If ϕ is of the form $P(t_1, t_2, \ldots, t_n)$, then we interpret each term t_i by replacing all occurrences of a variable v with $\ell(v)$, all occurrences of $f \in \mathcal{F}$ by $f^{\mathcal{M}}$, and in this fashion compute values $a_i \in A^{\mathcal{M}}$ for $i = 1, 2, \ldots, n$. Then $\mathcal{M} \models_\ell \phi$ if and only $(a_1, a_2, \ldots, a_n) \in P^{\mathcal{M}}$.

In other words, if $(a_1, a_2, \ldots, a_n) \in P^{\mathcal{M}}$, the formula ϕ is true in the model $\mathcal{M}$ with environment ℓ. If this is not the case, then the formula ϕ is false in the model $\mathcal{M}$, and we say $\mathcal{M} \not\models_\ell \phi$ (or $\mathcal{M} \models_\ell \phi$ does not hold).

If ϕ is of the form $\forall x \psi$, then $\mathcal{M} \models_\ell \phi$ if and only if for every $a \in A^{\mathcal{M}}$, $\mathcal{M} \models_{\ell[x \mapsto a]} \psi$. In other words, we try every possible value a for x, and for each one, we require that in the environment ℓ modified to include the interpretation that x maps to a, ψ is true in $\mathcal{M}$. If this holds, then ϕ is true in $\mathcal{M}$ with environment ℓ.

This suggests that if ϕ is of the form $\forall x \psi$, and we wish to demonstrate that $\mathcal{M} \not\models_\ell \phi$, we need only find one value a for which $\mathcal{M} \not\models_{\ell[x \mapsto a]} \psi$.

Similarly, if ϕ is of the form $\exists x \psi$, then $\mathcal{M} \models_\ell \phi$ if and only if for some $a \in A^{\mathcal{M}}$, $\mathcal{M} \models_{\ell[x \mapsto a]} \psi$. So showing $\mathcal{M} \not\models_\ell \phi$ requires demonstrating that for all possible values a, $\mathcal{M} \not\models_{\ell[x \mapsto a]} \psi$.

Finally, if ϕ is of the form $\neg\psi$, $\psi_1 \vee \psi_2$, $\psi_1 \wedge \psi_2$, or $\psi_1 \rightarrow \psi_2$, we use the semantics from propositional logic to assign a truth value to ϕ. For example, $\mathcal{M} \models_\ell \psi_1 \vee \psi_2$ if and only if $\mathcal{M} \models_\ell \psi_1$, or $\mathcal{M} \models_\ell \psi_2$, or both. This concludes the recursive definition of the satisfaction relation.

It is not hard to show that for a formula ϕ and two environments ℓ, ℓ' which agree on the free variables of ϕ, $\mathcal{M} \models_\ell \phi$ if and only if $\mathcal{M} \models_{\ell'} \phi$. Thus we can neglect the environment if ϕ has no free variables.

Time for an example. (We will reuse this example in the module on Alloy.)

Consider the sentence "None of Alma's lovers' lovers love her." Here "Alma" can be a constant a, and the concept "x loves y" can be a binary predicate $L(x, y)$. We can then represent the above sentence as the following formula:

$$\forall x \forall y (L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a)).$$

Intuitively, $L(x, a)$ says that x is Alma's lover, and $L(y, x)$ says that y loves x, so $L(x, a) \wedge L(y, x)$ says that y is one of Alma's lovers' lovers. $\neg L(y, a)$ says that y does not love Alma, and the quantifiers make sure this is true for any x, y.

Consider the model $\mathcal{M}$ defined by $A^{\mathcal{M}} = \{p, q, r\}$, $a^{\mathcal{M}} = p$, $L^{\mathcal{M}} = \{(p, p), (q, p), (r, p)\}$. How can we decide if $\mathcal{M}$ models this formula?

One way to do it is to apply the definition of $\models$ systematically. $\mathcal{M}$ models the formula if all of the following are true (for any environment ℓ:

$$\mathcal{M} \models_{\ell[x \mapsto p]} \forall y (L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a))$$

$$\mathcal{M} \models_{\ell[x \mapsto q]} \forall y (L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a))$$

$$\mathcal{M} \models_{\ell[x \mapsto r]} \forall y (L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a))$$

We can then apply similar reasoning to each of these. We could easily write a recursive computer program to do this, but it is tedious to do by hand. Instead, we will try to reason a little more concisely.

Intuitively, to demonstrate that $\mathcal{M} \not\models \forall x \forall y (L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a))$ in the model M where $a^{\mathcal{M}} = p$, $L^{\mathcal{M}} = \{(p, p), (q, p), (r, p)\}$, we must find interpretations of x and y making the LHS of the implication true and the RHS false.

Trying various possibilities shows us that if x is interpreted as p and y is interpreted as q, then $L(x, a)$, $L(y, x)$, and $L(y, a)$ are all assigned true. This has the effect we want of falsifying the entire formula. Now we can say this more precisely using our semantic definitions.

According to the semantics we have defined, for any environment ℓ, $\mathcal{M} \models_{\ell[x \mapsto p, y \mapsto q]} L(x, a)$, since $(p, p) \in L^{\mathcal{M}}$. Similarly, $\mathcal{M} \models_{\ell[x \mapsto p, y \mapsto q]} L(y, x)$ and $\mathcal{M} \models_{\ell[x \mapsto p, y \mapsto q]} L(y, a)$, since $(q, p) \in L^{\mathcal{M}}$.

But then $\mathcal{M} \not\models_{\ell[x \mapsto p, y \mapsto q]} L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a)$. By our discussion of the semantics of $\forall$, $\mathcal{M} \not\models_{\ell[x \mapsto p]} \forall y(L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a))$, and $\mathcal{M} \not\models_{\ell} \forall x \forall y(L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a))$. So the formula is false in $\mathcal{M}$ for any environment ℓ.

On the other hand, if we change $L^{\mathcal{M}}$ to be $\{(q, p), (r, q)\}$, then the whole formula becomes true. To see this, we note that $\mathcal{M} \models_{\ell} L(x, a) \wedge L(y, x)$ if and only if ℓ maps x to q and y to r. But for such an ℓ, $\mathcal{M} \not\models_{\ell} L(y, a)$, so $\mathcal{M} \models_{\ell} L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a)$. For all other ℓ, the left-hand side of the implication is false, so the implication is again true. Hence $\mathcal{M} \models \forall x \forall y(L(x, a) \wedge L(y, x) \rightarrow \neg L(y, a))$.

This analysis is reminiscent of the kinds of things we can do to avoid writing out a complete truth table to determine whether a formula in propositional logic is satisfiable.

Semantic entailment (2.4.2)

We would like to extend our notion $\Gamma \models \psi$ (where Γ is a set of formulas) from propositional logic to predicate logic, but the proliferation of potential models complicates things.

In predicate logic, we say $\Gamma \models \psi$ if and only if for all models $\mathcal{M}$ and all environments ℓ, whenever $\mathcal{M} \models_{\ell} \phi$ holds for all $\phi \in \Gamma$, then $\mathcal{M} \models_{\ell} \psi$ holds as well.

This seems a very strong condition: how do we check this for all possible models? We will demonstrate that this is possible, through careful reasoning, for the quantifier equivalences discussed in the last module. But first we will define notions of satisfiability, validity, and consistency.

The formula ψ is satisfiable if and only if there exists a model $\mathcal{M}$ and an environment ℓ such that $\mathcal{M} \models_{\ell} \psi$.

The formula ψ is valid if and only if for all models $\mathcal{M}$ and all environments ℓ, $\mathcal{M} \models_{\ell} \psi$.

The set Γ is consistent (or satisfiable) if and only if there is a model $\mathcal{M}$ and an environment ℓ such that for all $\phi \in \Gamma$, $\mathcal{M} \models_{\ell} \phi$.

In the previous module, we proved $\forall x(P(x) \vee Q(x)), \exists x(\neg P(x)) \vdash \exists x Q(x)$. Now we will give an argument that $\forall x(P(x) \vee Q(x)), \exists x(\neg P(x)) \models \exists x Q(x)$.

Consider a model $\mathcal{M}$ satisfying $\forall x(P(x) \vee Q(x))$ and $\exists x(\neg P(x))$. The truth of the second formula tells us that there is an $a \in A^{\mathcal{M}}$ such that $(a) \notin P^{\mathcal{M}}$. But the truth of the first formula tells us that for an environment ℓ that maps x to a, $P(x) \vee Q(x)$ is assigned true. Since $P(x)$ isn't true in ℓ, $Q(x)$ must be. And with $Q(x)$ true in some environment, $\exists x Q(x)$ is true in $\mathcal{M}$, which shows that the original entailment holds.

The informal argument on the previous slide avoided much notation; here is a more precise though perhaps less readable version.

Suppose $\mathcal{M} \models_\ell \forall x(P(x) \vee Q(x))$ and $\mathcal{M} \models_\ell \exists x(\neg P(x))$. Then for some $a \in A^{\mathcal{M}}$, $\mathcal{M} \models_{\ell[x \mapsto a]} \neg P(x)$, and so $\mathcal{M} \not\models_{\ell[x \mapsto a]} P(x)$. On the other hand, $\mathcal{M} \models_\ell \forall x(P(x) \vee Q(x))$ means that $\mathcal{M} \models_{\ell[x \mapsto a]} P(x) \vee Q(x)$. Knowing that $\mathcal{M} \not\models_{\ell[x \mapsto a]} P(x)$, we must have $\mathcal{M} \models_{\ell[x \mapsto a]} Q(x)$, and thus $\mathcal{M} \models_\ell \exists x Q(x)$.

The semantics of equality (2.4.3)

We use the term **intensional equality** to mean equality in the syntactic sense, as in $t = t$ for any term t.

But we also want to talk about equality in a semantic sense. Suppose in a particular model $\mathcal{M}$, $f^{\mathcal{M}}$ maps the interpretation of a to c, and $g^{\mathcal{M}}$ maps the interpretation of b to c. We then want the formula $f(a) = g(b)$ to be assigned T.

We ensure this by mandating that $=^{\mathcal{M}}$ should always be the equality relation on the set $A^{\mathcal{M}}$. This notion is called **extensional equality**.

Soundness and completeness

Intuitively, the soundness of predicate logic, which means that $\Gamma \vdash \psi$ implies $\Gamma \models \psi$, is not surprising; the rules of natural deduction are set up to preserve truth under interpretation.

We can see how the earlier proof of the soundness of propositional logic, which proceeded by structural induction on formulas, can be extended to cover predicate logic. Formulas are a bit more complicated, and we need some arguments about models similar to the one we just did.

The completeness of predicate logic means that $\Gamma \models \psi$ implies $\Gamma \vdash \psi$. This is more surprising, because we cannot mimic our earlier proof for propositional logic. That put together information from 2^n valuations (models) of a formula to yield a long finite proof.

But for predicate logic, we may not have a finite number of models, and they may be of very different types. It is not at all clear how to put all this information together to yield a proof.

The completeness of predicate logic was proved by Kurt Gödel in his Ph.D dissertation for the University of Vienna in 1930. A number of simpler proofs have been given by others, notably Henkin and Herbrand.

Gödel is more famous for two incompleteness theorems, which we will discuss shortly. Both the completeness and incompleteness theorems are proved in detail in PMath 432.

Rather than give the proofs of soundness and completeness, we will discuss some implications of them.

The most immediate implication is that, as with propositional logic, we have a way to show that a formula ϕ does not have a proof in natural deduction. By soundness, it suffices to demonstrate a model in which it is assigned false by our semantics.

This corresponds to the practice of demonstrating that a claim is false by showing a counterexample.

We will soon look at an automated version of this process. The software tool Alloy accepts specifications written in a language reminiscent of first-order logic, and checks whether or not assertions about the specification can be refuted with a “small” model.

As an example of the use of counterexamples to demonstrate invalidity, consider the sequent $\forall x(P(x) \lor Q(x)) \vdash \forall x P(x) \lor \forall x Q(x)$. We will show that this is invalid by giving a model which satisfies the LHS but not the RHS.

Let $A = \{a, b\}$, $P^{\mathcal{M}} = \{(a)\}$, $Q^{\mathcal{M}} = \{(b)\}$. Then $\mathcal{M} \models \forall x(P(x) \lor Q(x))$, but $\mathcal{M} \not\models \forall x P(x) \lor \forall x Q(x)$.

Thus $\forall x(P(x) \lor Q(x)) \not\models \forall x P(x) \lor \forall x Q(x)$, and by soundness, $\forall x(P(x) \lor Q(x)) \nvdash \forall x P(x) \lor \forall x Q(x)$.

Gödel’s proof of completeness was not **effective**; it did not provide a method for definitely deciding whether a formula was provable or not. Note that we have such a method for propositional logic; we can simply try all valuations.

Since a proof can be mechanically checked for validity, we can obtain a partial result; if a formula is provable, we can find a proof by generating all possible strings over the alphabet in which our proofs are written, and testing each one to see if it is a valid proof of the formula. This is not necessarily a fast or efficient algorithm, but it will find the proof. Provability is **semi-decidable**.

There does not seem to be any corresponding way to conclude that no proof exists.

By completeness, if we can find a model in which the formula is not valid, we can conclude that it is not provable. But such a model may be infinite, and we cannot check all possible interpretations of functions and predicates. In fact, there is no algorithm to test whether a formula in predicate logic is provable or not; this is **undecidable**.

To show this requires a formal model of computation. Such a model of computation, and a proof of the undecidability of provability, was first described by the American logician Alonzo Church in 1936. His model of computation was the lambda calculus, and his proof made heavy use of Gödel's incompleteness result.

Independently, a few months later, the British mathematician Alan Turing came up with a different model and a simpler proof.

Both Church's and Turing's proofs first demonstrated that there was no algorithm to answer questions about programs. Church showed it was impossible to decide whether two programs were functionally identical, and Turing showed that it was impossible to decide whether a program would halt.

They then showed that predicate logic could express the questions they were asking; hence it is also undecidable. A similar proof is given in section 2.5 of the text, where predicate logic is used to express the existence of a solution to a "correspondence problem" defined by the American mathematician Emil Post.

Post's correspondence problem is proved undecidable in CS 365; Turing's proof is given in CS 365 and 360; and Church's lambda calculus is studied in CS 442.

These results, proved in the 1930's, demonstrated limitations to computation even before electronic computers existed, and were a powerful influence on the subsequent development of hardware and software.

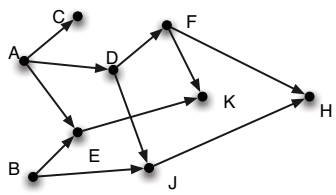
They continue to be important. We saw earlier that while satisfiability, validity, and provability are all decidable for propositional logic, they apparently cannot be determined efficiently for general formulas. Now we see that the greater expressivity of predicate logic makes these questions undecidable.

The tradeoff between expressivity and efficiency, in both theoretical and practical terms, is the subject of active research.

Expressiveness of predicate logic (2.6)

Because many properties of our specifications, our algorithms, and our programs can be expressed in logical terms, we naturally wish to make our logical languages as expressive as possible, though not at the cost of being unable to work with the resulting formulas.

Our example of the limited expressibility of first-order predicate logic will involve directed graphs, which are used in CS 135, CS 240, Math 239, CS 341, and many other courses. A directed graph G consists of a finite set V of vertices and an binary edge relation E. We often visualize graphs by drawing dots to represent each of the elements of V, and drawing an arrow from x to y iff $(x, y) \in E$ (in which case we say "there is an edge from x to y").



The question of reachability in directed graphs (given a graph G and two nodes u, v, is there a directed path from u to v?) is important in many areas of computer science. A program using pointer-based data structures is free of memory leaks if every allocated segment of memory can be reached from a program variable. Finding solutions to solitaire puzzles, or more generally, goal search and motion planning can be expressed in terms of reachability.

Math 239 and CS 341 cover efficient algorithms for reachability when the graph is given explicitly. But there are many computational situations where the graph is not explicit. For example, the nodes of the graph could be states of a program or system, and the edges could represent steps or transitions. Questions of freedom from error, safety, or freedom from deadlock can then be expressed in terms of reachability.

It is therefore surprising to learn that reachability cannot be expressed in predicate logic. Most of the "impossibility" results discussed in this course are only stated, not proved, but we can prove this one from completeness, by way of a couple of nice results in formal logic. First, though, we should discuss what it means to attempt to express reachability in predicate logic.

To talk about reachability using a formula, we let the variables stand for nodes (vertices), and use a binary predicate R for the edge relation. That is, if in a particular graph G, there is an edge from u to v, then $R(u, v)$ is true in the model defined by G.

A formula describing a more general relationship between u and v is one in which u and v are the only free variables. For instance, the formula $\exists x(R(u, x) \wedge R(x, v))$ is made true by a model defined by G if and only if there is a path of length 2 in G.

The difficulty comes because a path from u to v in an arbitrary graph can have unbounded (though finite) length, and we need to express reachability with a fixed (finite) formula.

We begin our proof that reachability is not expressible in predicate logic with an important and general result.

Compactness Theorem (2.24): Let Γ be a (possibly infinite) set of sentences of predicate logic. If all finite subsets of Γ are satisfiable, then so is Γ.

Proof: Suppose that all finite subsets of Γ are satisfiable, but Γ is not satisfiable. Then $\Gamma \models \bot$ (since no model makes all $\phi \in \Gamma$ true). By completeness, $\Gamma \vdash \bot$. This must have a finite proof, mentioning only a finite subset Δ of sentences from Γ. Then $\Delta \vdash \bot$, and by soundness, $\Delta \models \bot$. But this is a contradiction to the assumption that Δ, as a finite subset of Γ, is satisfiable. $\square$

As a warmup on the use of compactness, we prove one of a number of related theorems with the names of Löwenheim and Skolem on them.

Theorem (2.25): Let ψ be a sentence of predicate logic such that for any natural number $n \geq 1$, there is a model of ψ with at least n elements. Then ψ has a model with infinitely many elements.

Proof: For all n, let ϕ_n be defined as:

$$\exists x_1 \exists x_2 \ldots \exists x_n \bigwedge_{1 \leq i < j \leq n} \neg(x_i = x_j).$$

ϕ_n asserts that there are at least n elements. Now define $\Gamma = \{\psi\} \cup \{\phi_n \mid n \geq 1\}$. We will apply the compactness theorem to Γ.

To do so, we have to show that any finite subset Δ of Γ is satisfiable. Let Δ be an arbitrary finite subset of Γ, and let k be the index of the "largest" formula ϕ_n in Δ. Since there is a model of ψ with at least k elements, $\{\psi, \phi_k\}$ is satisfiable.

But since $\phi_k \rightarrow \phi_n$ is valid for any $n \leq k$, Δ must be satisfiable as well. Now we can invoke compactness and say that Γ is satisfiable by some model $\mathcal{M}$. But if $\mathcal{M}$ is finite, say of size t, then it cannot satisfy ϕ_{t+1}. Thus $\mathcal{M}$ is infinite. □

Intuitively, this theorem says that the concept of "finiteness" is not expressible in predicate logic. Next, we will use compactness to prove that reachability is not expressible in predicate logic.

Theorem (2.26): There is no formula ϕ in predicate logic with free variables u, v and the following property: ϕ is satisfied by a model defined by a directed graph G if and only if there is a path in G from the node associated with u to the node associated with v.

Proof: Suppose that there was such a ϕ. Recall that R is the edge relation for G. We use two constants c, c', and define ϕ_0 as $c = c'$ and

$$\phi_n = \exists x_1 \exists x_2 \ldots \exists x_n (R(c, x_1) \wedge R(x_1, x_2) \wedge \ldots \wedge R(x_{n-1}, c'))$$

Let $\Delta = \{\neg\phi_n \mid n \geq 0\} \cup \{\phi[c/u][c'/v]\}$. Δ is unsatisfiable, because the first set says "There is no path from c to c'" and the second set says that there is.

But any finite subset of Δ is satisfiable, since it can contain at most a finite subset of $\{\neg\phi_n \mid n \geq 0\}$, and we can construct a graph large enough so that there are vertices we can associate with c, c' such that there aren't paths short enough from c to c' to make the ϕ_i false, but there is a path long enough to make $\phi[c/u][c'/v]$ satisfiable.

This is a contradiction to the Compactness Theorem, and therefore the formula ϕ expressing reachability cannot exist. □

Since predicate logic is inadequate to express this important concept, we must go beyond it.

Existential second-order logic

The language of predicate logic we have defined is called **first-order**. Second-order logic extends first-order logic by permitting quantification not just over variables, but over predicate symbols as well. Here is non-reachability expressed in second-order logic:

$$\begin{aligned} \exists P \forall x \forall y \forall z \quad & P(x,x) \\ \wedge \quad & (P(x,y) \wedge P(y,z) \rightarrow P(x,z)) \\ \wedge \quad & (\neg P(u,v)) \\ \wedge \quad & (R(x,y) \rightarrow P(x,y)). \end{aligned}$$

Second-order logic allows arbitrary quantification over predicates. However, we only used an existential quantifier in our definition of non-reachability, and so we have a formula of **existential** second-order logic.

This is one way of restricting the full power of second-order logic. Another way that has proved useful in various branches of computer science is to allow only unary predicates; this is known as **monadic** second-order logic.

We showed how to express non-reachability in existential second-order logic; the book mentions that reachability is also expressible in existential second-order logic. This is a consequence of a more general, and quite surprising, result.

Earlier, we discussed the fact that the satisfiability problem for propositional logic was “NP-complete” and therefore thought to be hard computationally. In CS 341, you learn that this really means “complete for the class NP”. Intuitively, NP is the class of problems whose solutions are efficiently verifiable. Many problems have this property: it may be hard to find a solution, but if someone gives you one, it is easy to verify that it is a solution.

Fagin proved in 1974 that the set of graph properties in NP are exactly those which can be expressed in existential second-order logic. This is surprising because it relates a notion of efficient computation to one of description with no mention of a model of computation. The field of descriptive complexity explores similar results and their implications (e.g. in databases).

Other proof systems

The other proof systems we briefly discussed for propositional logic (semantic tableaux, sequent calculus, and transformational proofs) can all be extended to propositional logic in a fairly straightforward fashion.

Of these, the most important in practical terms is probably transformational proofs. Recall that the idea, when applied to propositional logic, was to use equivalences in an algebraic fashion. The key advantage was that we could make a substitution of one equivalent subformula for another, whereas for natural deductions, the syntactic changes only happen at the “top level”.

We extend the notion of transformational proof to predicate logic by allowing substitutions based on the quantifier equivalences discussed in this module (and summarized in Theorem 2.13 in the text).

As before, mathematical proofs involving quantification are in practice a mixture of natural deduction and transformational proofs. Notions such as proof by contradiction remain important, and new notions of natural deduction for predicate logic such as $\forall$-introduction become important.

Being able to understand and derive such mathematical proofs requires familiarity with quantifier equivalences as well as knowledge of which possible equivalences fail to hold and what parts of them might be salvageable.

For example, we know that $\forall x(P(x) \vee Q(x)) \not\models \forall x P(x) \vee \forall x Q(x)$. On the other hand, we can show that $\forall x P(x) \vee \forall x Q(x) \models \forall x(P(x) \vee Q(x))$. So a transformation in one direction is possible.

Among the most important quantifier equivalences are the "de Morgan"-style ones, such as $\neg\forall x\phi \equiv \exists x\neg\phi$.

To illustrate an extreme example of the use of such equivalences, we will quote a central theorem from CS 360 and CS 365, to examine its form (rather than its meaning or proof).

The pumping lemma describes a property of regular languages, sets of strings that are accepted by finite state machines (or equivalently, described by regular expressions). These are first studied in CS 241. It is of the form "If L is regular, then ϕ holds". ϕ happens to be describable in first-order logic using quantifiers.

Here is the form of ϕ, as typically stated in CS 360. Don't worry about the precise meaning.

$L \in \mathcal{R} \rightarrow \exists n$
$\forall s$ such that $|s| = n$
$\exists x \exists y \exists z$ such that $s = xyz$
$\forall i \geq 0\ xy^iz \in L$

The theorem states a property of regular languages, but it is almost never applied to regular languages. Instead, it is applied to languages believed nonregular, in the contrapositive. Instead of $\psi \rightarrow \phi$, it is used in the form $\neg\phi \rightarrow \neg\psi$. The contrapositive of the pumping lemma looks like this:

$\neg(\exists n$
$\forall s$ such that $|s| = n$
$\exists x \exists y \exists z$ such that $s = xyz$
$\forall i \geq 0\ xy^iz \in L) \rightarrow L \notin \mathcal{R}$

In order to work with this form, the negation has to be pushed all the way through the nested quantifiers, using the deMorgan-style transformations.

This yields the following formula:

$(\forall n$
$\quad \exists s$ such that $|s| = n$
$\quad\quad \forall x \forall y \forall z$ such that $s = xyz$
$\quad\quad\quad\quad \exists i \geq 0\ xy^i z \notin L) \rightarrow L \notin \mathcal{R}.$

This is the form in which CS 360 students actually work with the pumping lemma in order to prove that a language L is not regular. Without exposure to formal logic, it may not be clear why this form is equivalent to the original statement of the pumping lemma.

What's next?

The formulas on the previous slides did not conform to our formal definition of formulas in predicate logic, because they made intuitive use of symbols from set theory and notation for strings that cannot be interpreted arbitrarily. Notation from arithmetic and higher mathematics is also important in the proofs we encounter and the specifications we wish to write.

In the next module, we will discuss how these familiar informal notions can be made formal, and cover some rules of thumb about how to identify formal elements in informal descriptions and how to work with them.

Then we will develop proof machinery for programs, and finally, discuss the use of the software tool Alloy.

Goals of this module

You should understand the semantics of predicate logic, how to apply it for individual formulas, and how to reason about it in general.

You should be able to come up with counterexamples for invalid formulas.

You should understand the meaning of soundness and completeness for predicate logic, and the implications of it as discussed, including the compactness theorem.

You should understand why reachability is not expressible in first-order logic, and how it can be expressed in second-order logic.

You should understand the ideas of transformational proof for predicate logic, and its use in practice.