

CDM

Predicate Logic

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Predicate Logic
- Syntax
- Tarski and Truth

Predicate Logic

Why Predicate Logic?

Neither propositional nor equational logic is powerful enough to express statements such as

- The function $x \mapsto x^3$ is continuous.
- There is a prime between n and $2n$ (for $n > 1$).
- Deadlock cannot occur (in some protocol).
- Upon completion, the stack is empty.
- The algorithm terminates on all inputs.

For example, the second assertion (known as the Bertrand-Chebyshev theorem) is of importance for algorithms that need to select primes of some suitable size: there is a prime between 2^k and 2^{k+1} so primes of arbitrary size exist (this is just the tip of an iceberg).

How could we state this assertion clearly, as a formal expression that might be computer-understandable?

Predicate Logic

So how do we construct a language suitable for (most of) mathematics and computer science?

Let's start with a small fragment, say, arithmetic. A reasonable approach would be to use only

- basic arithmetic concepts (addition, multiplication, order) and
- purely logical constructs such as "and", "not" etc.

The logical constructs will include all of propositional logic so we can still combine assertions by connectives "and", "or" and the like. But there will also be quantifiers that allow one to make statements about the existence of objects with certain properties and about properties of all objects.

Terminology: This type of system is called *predicate logic* but note that we also allow for functions. Ultimately we will need to fine-tune this a bit, but for the moment this one modification is enough.

Designing a Stronger Language

We could express the statement about primes like so:

- | | |
|----------------------|--------------------------|
| • for all x | - universal quantifier |
| • there exists a y | - existential quantifier |
| • $x \leq y$ | - binary relation |
| • and | - logical connective |
| • $y \leq 2x$ | - binary relation |
| • and | - logical connective |
| • y is prime | - unary relation |

The assertion "is prime" in the last line still needs to be unraveled.

Primality, defined

Clearly, we can express primality in terms of multiplication.

x is prime iff

- for all u - universal quantifier
- for all v - universal quantifier
- $x = u \cdot v$ - binary function, equation
- implies - logical connective
- $u = 1$ - constant, equation
- or - logical connective
- $v = 1$ - constant, equation

As it turns out, this is all we need: adding quantifiers increases the expressiveness of our language enormously.

The Range of Variables

Note that the last definition tacitly assumes that we interpret the variables as ranging over the integers.

If we were dealing with the rationals the definition would, of course, no longer make sense.

We will see in a moment that the range of quantifiers is pinned down precisely when we address the question of semantics: what exactly is the meaning of a formula in predicate logic?

The Language

We keep all logical connectives. But we add

- **constants** that denote individual objects,
- **variables** that range over individual objects,
- **quantifiers** that express "for all" and "there exists",
- **function symbols** that denote functions,
- **relation symbols** that denote relations.

Notation:

- a, b, c, \dots for constants,
- x, y, z, \dots for variables,
- \forall for the **universal quantifier**,
- \exists for the **existential quantifier**,
- f, g, h, \dots for function symbols,
- R, P, Q, \dots for relation symbols.
- Always allow $=$ for equality.

Signatures and Arities

One should distinguish more carefully between function and relation symbols of different arity: nullary, unary, binary functions and so on.

In most concrete structures only a finite number of function and relation symbols are needed.

Hence, we can convey the structure of the non-logical part of the language in a **signature** or **similarity type**: a list that indicates the arities of all the function and relation symbols.

Example 1. For group theory one often uses a signature $(2, 1, 0)$: there is one binary function symbol for the group multiplication, one unary function symbol for the inverse operation and a nullary function symbol (i.e. a constant) for the neutral element.

We could also use a language of signature (2) but at the cost of slightly more complicated axioms.

Example: Fields

For the classical algebraic theory of fields one minimally uses a language

$\mathcal{L}(+, \cdot, 0, 1)$ of signature $(2, 2, 0, 0)$

- binary function symbols $+$ and \cdot for addition and multiplication,
- constants 0 and 1 for the respective neutral elements.

Of course, more functions could be added: additive inverse, subtraction, multiplicative inverse, division. Introducing only addition and multiplication is the minimalist approach here.

Some Formulae

Now consider formulae such as

$$\begin{aligned} \forall x, y (x + y = y + x) \\ \forall x (x \cdot 0 = 0) \\ \forall x \exists z ((\neg x = 0) \rightarrow z \cdot x = 1) \\ 1 + 1 = 0 \end{aligned}$$

It is intuitively clear what these formulae mean.

Except for the last one, they are all true in any field. In fact, the first two hold in any ring.

The third one is true in a field, but is false in an arbitrary ring.

And the last only holds in rings of characteristic 2. In fact, it defines rings of characteristic 2.

Example: Boolean Algebra

In Boolean algebra one uses the language

$$\mathcal{L}(\sqcup, \sqcap, \bar{}, 0, 1) \text{ of signature } (2, 2, 1, 0, 0)$$

So we have

- binary function symbols \sqcup and \sqcap (for join and meet)
- unary function symbol (for complement)
- constants 0 and 1

Some formulae:

$$\forall x, y \exists z (x \sqcup y = \bar{z})$$

$$\exists x \forall y (x \sqcap y = 0)$$

$$\forall x \exists y (x \sqcup y = 1 \wedge x \sqcap y = 0)$$

Example: Arithmetic

For arithmetic it is convenient to have a relation symbol for order:

$$\mathcal{L}(+, \cdot, 0, 1; <) \text{ of signature } (2, 2, 0, 0; 2)$$

- binary function symbols $+$ and \cdot (for integer addition and multiplication)
- constants 0 and 1 (for integers 0 and 1)
- a binary relation symbol $<$ (for the less-than relation)

Assuming we are quantifying over the natural numbers we have assertions like

$$\forall x \exists y (x < y)$$

$$\exists x \forall y (x = y \vee x < y)$$

$$\forall x (x + 0 = x)$$

Intuitively, these are all true.

Streamlining Notation

Writing arithmetic statements in predicate logic may seem overly terse, but consider the following classical statement:

There do not exist four numbers, the last being larger than two, such that the sum of the first two, both raised to the power of the fourth, are equal to the third, also raised to the power of the fourth.

In slightly more modern parlance this turns into

There are no positive integers x, y, z and n , where $n > 2$, such that $x^n + y^n = z^n$.

and is now recognizable as Fermat's Last Theorem.

This combination of ordinary language and formal expressions is the de facto gold-standard in mathematics and computer science; essentially all the literature is written this way.

Streamlining Notation, II

We can take one more step and translate into predicate logic. Starting from the last sentence, this translation is not hard:

$$\neg \exists x, y, z, n \in \mathbb{N}^+ (x^n + y^n = z^n \wedge n > 2)$$

The variables here are assumed to range over the positive integers as indicated by the annotation at the quantifier.

Even if you prefer the standard notation over the more compact one (and most people do), the latter is clearly better suited as input to any kind of algorithm – it is much easier to parse. Always remember the Entscheidungsproblem.

Expressiveness: Induction

As another example of the expressiveness of predicate logic consider the venerable Principle of Induction. We can write it down as a formula as follows.

$$R(0) \wedge \forall x (R(x) \rightarrow R(x+1)) \rightarrow \forall x R(x)$$

Here we have added another unary relation symbol R to our language. So $R(x)$ asserts that number x has some unspecified property.

The Principle of Induction then simply asserts that is formula is true over the natural numbers. Note, though, that in many applications $R(x)$ would actually be replaced by a formula of arithmetic such as

$$\sum_{i \leq x} i = x(x+1)/2$$

Dissecting the Induction Axiom

The formal expression reflects very nicely the various components of an induction argument.

$$\begin{array}{ll} R(0) \wedge & \text{base case} \\ \forall x (R(x) & \text{induction hypothesis} \\ \rightarrow R(x+1)) & \text{induction step} \\ \implies & \\ \forall x R(x) & \text{conclusion} \end{array}$$

As usual, the hard part is to show that $R(x) \rightarrow R(x+1)$ without any further assumptions about x .

Primality Formula

In the language of arithmetic we can write a formula $prime(x)$ that expresses the assertion “ x is prime”.

$$1 < x \wedge \forall u, v (x = u \cdot v \rightarrow u = 1 \vee v = 1)$$

Note that x here is a **free variable** (not quantified over) and thus is not bound to any particular value or range of values.

If we replace x by $3 = 1 + 1 + 1$ we get a true statement. But if we replace x by $4 = 1 + 1 + 1 + 1$ we get a false statement.

So, we can use free variables to pick out elements with certain properties.

In the assertion that there are infinitely many primes x appears as a bound variable.

$$\forall z \exists x (x > z \wedge prime(x))$$

Syntax

More Syntax

While we are not interested in writing a parser or theorem prover, we still need to be a bit more careful about the syntax of our language of predicate logic. The components of a formula can be organized into a taxonomy like so:

- variables, constants and terms,
- equations,
- atomic formulae,
- propositional connectives, and
- quantifiers.

Every programming language has a defining report (which no one ever reads, other than perhaps compiler writers, it's the document that uses the imperative “shall” a lot), so think of this as the defining report for predicate logic.

Only the quantification part is really new; propositional connectives will be the same as in propositional logic while terms and equations will be the same as in equational logic.

Graded Alphabet

We need a supply of variables Var as well as function symbols and predicate symbols. These form a graded alphabet $\Sigma = \Sigma_0 \cup \Sigma_1$, where every function symbol and relation symbol has a fixed number of arguments, determined by a **arity** map:

$$ar : \Sigma \rightarrow \mathbb{N}$$

We write $\mathcal{L}(\Sigma)$ for the language constructed from signature Σ .

Function symbols of arity 0 are constants and relation symbols of arity 0 are Boolean values (true or false) and will always be written \top and \perp .

In any concrete application, Σ will be finite. However, in the literature you will often find a big system approach that introduces a countable supply of function and relation symbols for each arity. For our purposes a signature custom-designed for a particular application is more helpful.

Syntax: Terms

Definition 1. The set $\mathcal{T} = \mathcal{T}(\Sigma) = \mathcal{T}(\text{Var}, \Sigma)$ of all **terms** is defined by

- Every variable is a term.
- If f is an n -ary function symbol, and t_1, \dots, t_n are terms, $n \geq 0$, then $f(t_1, \dots, t_n)$ is also a term.

A **ground term** is a term that contains no variables.

Note that $f()$ is a term for each constant (0-ary function symbol) f . For clarity, we write a, b, c and the like for constants.

The idea is that every ground term corresponds to a specific element in the underlying structure. For arbitrary terms we first have to replace all variables by constants. For example, in arithmetic the term $(1 + 1 + 1) \cdot (1 + 1)$ corresponds to the natural number 6. We could introduce constants for all the natural numbers, but there is no need to do so: we can build a corresponding term from the constant 1 and the binary operation $+$.

Syntax: Atomic Formulae

Given a few terms, we can apply a predicate to get a basic assertion (like $x + 4 < y$).

Definition 2. An **atomic formula** is an expression of the form

$$R(t_1, \dots, t_n)$$

where R is an n -ary relation symbol, and the t_1, \dots, t_n are terms.

These are basically the atomic assertions in propositional logic: once we have values for the variables that might appear in the terms, an atomic formula can be evaluated to true or false.

But note that we do need bindings for the variables, $R(x, y)$ per se has no truth value.

Equality

Equality plays a special role in our setup. We will assume that the language has a special binary relation symbol \approx that is used to denote equality:

$$s \approx t$$

where s and t are arbitrary terms.

Unlike with all the other relation symbols, the meaning of \approx is fixed once and for all: it is always interpreted as equality.

We will always assume that equality is part of the language, though predicate logic without equality is also of interest. For example, a concrete application may not use any function symbols whatsoever and only deal with relations.

We will later cave in and write the standard equality symbol $=$ but for the time being we will be careful: $s \approx t$ is just a formula in predicate logic.

Formulae

Definition 3. The set of formulae of predicate logic is defined by

- Every atomic formula is a formula.
- If φ and ψ are formulae, so are $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, and $(\varphi \rightarrow \psi)$.
- If φ is a formula and x a variable, then $(\exists x \varphi)$ and $(\forall x \varphi)$ are also formulae.

So these are compound formulae versus the atomic formulae from above.

The φ in $(\exists x \varphi)$ and $(\forall x \varphi)$ is called the *matrix* of the formula.

Note that a term by itself is not a formula: it denotes an element (if it has no free variables) rather than a truth value.

Preserving Sanity

Since we are not compilers, we omit unnecessary parentheses and write formulae such as

$$\forall x (R(x) \rightarrow \exists y S(x, y))$$

The intended meaning is: "For any x , if x has property R , then there exists a y such that x and y are related by S ."

Binary relation and function symbols are often written in infix notation, using standard mathematical symbols: As usual, one often uses infix notation for terms and atomic formulae: $x + y < z$ is easier to read than $<(+ (x, y), z)$.

One often contracts quantifiers of the same kind into one block.

$$\forall x \forall y \exists z (\approx (+ (x, z), y))$$

Sloppy, but eminently readable, style

$$\forall x, y \exists z (x + z \approx y)$$

Sentences

Definition 4. A variable that is not in the range of a quantifier is *free* or *unbound* (as opposed to *bound*). A formula without free variables is *closed*, or a *sentence*.

One often indicates free variables like so:

$$\begin{array}{ll} \varphi(x, y) & x \text{ and } y \text{ are free} \\ \exists x \varphi(x, y) & \text{only } y \text{ is free} \\ \forall y \exists x \varphi(x, y) & \text{closed.} \end{array}$$

Substitutions of terms for free variables are indicated like so:

$$\begin{array}{ll} \varphi(s, t) & \text{replace } x \text{ by } s, y \text{ by } t \\ \varphi[s/x, t/y] & \text{replace } x \text{ by } s, y \text{ by } t \end{array}$$

We will explain shortly how to compute the truth value of a sentence.

Interpreting Free Variables

A priori, a formula $\varphi(x)$ with a free variable x has no truth value associated with it: we need to replace x by a ground term to be able to evaluate.

However, taking inspiration from equational logic, it is convenient to define the truth value a formula with free variables to be the same as its *universal closure*: put universal quantifiers in front, one for each free variable.

$$\forall x, y, z (x * (y * z) \approx (x * y) * z)$$

is somewhat less elegant and harder to read than

$$x * (y * z) \approx (x * y) * z$$

Clashing Variables

Note that according to our definition it is perfectly agreeable to quantify over an already bound variable. To make the formula legible one then has to rename variables. For practical reasons it is best to simply disallow clashes between free and bound variables.

$$\forall x (R(x) \wedge \exists x \forall y S(x, y))$$

Better: rename the x inside

$$\forall x (R(x) \wedge \exists z \forall y S(z, y))$$

These issues are very similar to problems that arise in programming languages (global and local variables, scoping issues). They need to be addressed but are not of central importance.

Getting Serious

Our definition of a formula of predicate logic is sufficiently precise to reason about the logic, but it is still a bit vague if we try to actually implement an algorithm that operates on these formulae.

Exercise 1. Explain how to implement formulae in predicate logic. Describe the data structure and make sure that it can be manipulated in a reasonable way.

Exercise 2. Give a precise definition of free and bound variables by induction on the buildup of the formula.

Exercise 3. Implement a renaming algorithm that removes potential scoping clashes in the variables of a formula.

Exercise 4. Implement a substituting algorithm that replaces a free variable in a formula by a term.

Tarski and Truth

What is Truth?

How can we explain precisely what it means for an assertion to be true, to be valid? We would like some sweeping, global definition of truth that handles all areas of discourse, at least in mathematics and computer science. While this broad approach corresponds nicely to one's philosophical assumptions about the world (at least for an unreconstructed Platonist like myself) it leads to some rather dicey problems: for example, what should we do with assertions like

"This sentence is false."

We certainly would want to have the ability to declare certain assertions such as "100 is a prime number" as false.

How can we then avoid paradoxes as in the self-referential statement above?

Tarski's Theory of Truth

In order to succeed one needs to distinguish carefully between an object language and a meta-language. It is also helpful to restrict one's attention to truth in a particular domain such as, say, group theory or number theory. In the 1930's Alfred Tarski was the first to seriously tackle to problem of explaining truth for a sentence in a formal language.

We already have a nice formal language, though so far a formula is just a syntactic object, think of it as a string or a parse tree. We can now use Tarski's ideas to attach meaning to a formula, to establish a relationship with the world of actual objects such as numbers, functions, hash tables, algorithms, and so on.

The key is to define the truth value of a formula relative to a given structure, a mini-universe that allows us to make sense out of the components of the formula.

Semantics, Informally

Here is a simple example from basic arithmetic (over the natural numbers). Consider the formula

$$\forall x \exists y (x < y)$$

Its components are

x, y	variables, range over natural numbers
$<$	comparison, the less-than relation

The intended meaning of the formula, expressed in classical math-speak, is then

For any natural number x there exists a natural number y such that x is less than y .

So this formula is clearly true, valid.

Context Dependence

The key problem is that the formula $\forall x \exists y (x < y)$ itself does not express any of the auxiliary information:

- There is no indication that the variables range over natural numbers, and
- there is no indication that the binary relation symbol $<$ corresponds to the standard order relation on the naturals.

This would become more clear if we had written $\forall x \exists y R(x, y)$ instead, the use of a well-known symbol such as $<$ is just syntactic sugar.

Moreover, if we were to interpret the variables as ranging over the integers instead (a relatively minor change) the formula would be false, invalid.

To settle all these issues we have to consider so-called *structures* over which the formulae of predicate logic can be evaluated.

Semantics: Structures

So what exactly are these structures that we need to interpret a formula in predicate logic? Fix some language $\mathcal{L} = \mathcal{L}(\Sigma)$ where Σ is a graded alphabet as above.

Definition 5. A (first order) structure is a set together with a collection of functions and relations on that set. The signature of a first order structure is the list of arities of its functions and relations.

In order to interpret formulae in $\mathcal{L}(\Sigma)$ the signatures have to match, which we will tacitly assume from now on. So a structure in general looks like so:

$$\mathcal{A} = \langle A; f_1, f_2, \dots, R_1, R_2, \dots \rangle$$

The set A is the carrier set of the structure. Unary and binary functions and relations are by far the most important in applications, but higher arities may occur.

Abstract Data Types

Note that a first order structure is not all that different from a data type. To wit, we are dealing with a

- collection of objects (values), and
- operations on these objects.

The relations can safely be interpreted as operations: a k -ary relation is just a map $R : A^k \rightarrow \mathbb{B}$.

In the case where the carrier set is finite (actually, finite and small) we can in fact represent the whole structure by a suitable data structure. For infinite carrier sets things are a bit more complicated.

Data types (or rather, their values) are manipulated in programs, we are here interested in describing properties of structures using the machinery of predicate logic.

Interpretations

Given a formula and a structure of the same signature we can associate the function and relation symbols in the formula with real functions and relations in a structure (of the same arity).

$$\begin{aligned} f \text{ function symbol} &\rightsquigarrow f^{\mathcal{A}} \text{ a function in } \mathcal{A} \\ R \text{ function symbol} &\rightsquigarrow R^{\mathcal{A}} \text{ a relation in } \mathcal{A} \end{aligned}$$

Given this interpretation of function and relation symbols over \mathcal{A} we can determine whether a formula holds true over \mathcal{A} .

This is very different from trying to establish universal truth. All we are doing here is to confirm that, in the context of a particular structure, a certain formula is valid. As it turns out, this is all that is really needed in the real world.

Of course, when the structure changes the formula may well become false.

Example: Arithmetic

Consider arithmetic: The language is $\mathcal{L}(+, \cdot, 0, 1; <)$ and has type $(2, 2, 0, 0; 2)$.

We can interpret a formula in this language over any structure of the same type. Of course, the most important structure for arithmetic is

$$\mathcal{N} = \langle \mathbb{N}; +, \cdot, 0, 1, < \rangle$$

the set of natural numbers together with the standard operations but we will see that there are others.

Note the slight abuse of notation here (as is standard practice). More precise would be to write: The function symbol $+$ is interpreted by $+^{\mathcal{N}}$, the standard operation of addition of natural numbers.

For our purposes there is no gain in being quite so careful.

Some Arithmetic Formulae

With this interpretation over \mathcal{N} , the formula

- $0 < 1$ is true
- $\forall x \exists y (x < y)$ is true
- $\forall x, y (x < y \rightarrow \exists z (x < z \wedge z < y))$ is false

How about the primality formula

$$\varphi(x) = 1 < x \wedge \forall y, z (x \approx y \cdot z \rightarrow y \approx 1 \vee z \approx 1)$$

This formula has a free variable x , so we need to bind x (replace it by a term) before we can determine truth.

If we write $\underline{n} = \underbrace{1 + 1 + \dots + 1}_{n \text{ times}}$ for the term representing n the set of primes is just

$$\{ n \mid \varphi(\underline{n}) \text{ holds in } \mathcal{N} \}.$$

The Reals

Suppose we interpret our formulae over the structure of the real numbers instead. This is possible since it has same signature $(2, 2, 0, 0; 2)$:

$$\mathcal{R} = \langle \mathbb{R}; +, \cdot, 0, 1, < \rangle$$

Now $+$ refers to addition of reals, 0 is the real number zero, and so on. These operations are much more complicated, but as far as our formula is concerned all that matters is that they have the right arity.

Over the reals the density formula

$$\forall x, y (x < y \rightarrow \exists z (x < z \wedge z < y))$$

holds.

On the other hand, the primality statement $\varphi(x)$ is not interesting over \mathcal{R} : there is no binding for x that makes it true.

Validity, Informally

We can now give a first informal definition of truth or validity.

Definition. A formula of predicate logic is **valid** if it holds over any structure of the appropriate signature.

For free variables this definition is motivated by formulae such as $x * y \approx y * x$: we want this to mean that the operation $*$ is commutative. Note, though, that we apply the same approach to function and relation symbols (other than equality): any possible interpretation has to be taken into account

Of course, we can pin down some of the properties of the corresponding functions and relations in the formula itself, but that's all we can do. There is no magic that says that the symbol $+$ must always be interpreted as some sort of addition over some algebraic structure.

Some Valid Formulae

It is clear that a formula like $\varphi \rightarrow \varphi$ is valid. In fact, if we replace the propositional variables in any tautology by arbitrary sentences we obtain a valid sentence of predicate logic. More precisely, let

$$\varphi(p_1, p_2, \dots, p_n)$$

be a tautology with propositional variables p_1, p_2, \dots, p_n . Let ψ_1, \dots, ψ_n be arbitrary sentences of predicate logic. Then

$$\varphi(\psi_1, \psi_2, \dots, \psi_n)$$

is a valid sentence of predicate logic. True, but not too interesting.

Satisfiability, Informally

Again in analogy to propositional logic we can define satisfiability.

Definition 6. A formula of predicate logic is **satisfiable** if it is true for some interpretation of the variables, functions and relations. It is a **contradiction** if it is true for no interpretation of the variables, functions and relations.

For example, in the language of binary relations the formula

$$x R x \wedge (x R y \wedge y R z \rightarrow x R z)$$

is satisfied by any structure \mathcal{A} that carries an irreflexive transitive relation $R^{\mathcal{A}}$. On the other hand,

$$\forall x (x \not\approx c)$$

where c is a constant is a contradiction: we can interpret x as the element in the structure denoted by c in which case equality holds.

Quantifier Manipulations

Slightly more complicated examples for valid formulae are

$$\forall x \forall y \varphi(x, y) \rightarrow \forall y \forall x \varphi(x, y)$$

$$\exists x \exists y \varphi(x, y) \rightarrow \exists y \exists x \varphi(x, y)$$

$$\exists x \forall y \varphi(x, y) \rightarrow \forall y \exists x \varphi(x, y)$$

Note, though, that the following is not valid:

$$\forall x \exists y \varphi(x, y) \rightarrow \exists y \forall x \varphi(x, y)$$

Exercise 5. Verify that the first three formulae are true and come up with an example that shows that the last one is not.

Counting

Another good source of valid formulae are assertions about the number of elements in the underlying structure. How do we say "there are exactly n elements in the ground set" in predicate logic? First, a formula which states that there are at most n elements.

$$EX_{\leq n} = \exists x_1, \dots, x_n \forall y (y \approx x_1 \vee \dots \vee y \approx x_n)$$

Second, a formula which states that there are at least n elements.

$$EX_{\geq n} = \exists x_1, \dots, x_n (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \dots \wedge x_{n-1} \neq x_n)$$

All these formulae are clearly satisfiable. The conjunction $EX_{\leq n} \wedge EX_{\geq n}$ pins down the cardinality to exactly n . Also, a formula

$$EX_{\geq n} \rightarrow EX_{\geq m}$$

is valid whenever $m \leq n$.

Infinity

How about a formula that states that there are infinitely many elements?

$$EX_{\geq 1} \wedge EX_{\geq 2} \wedge \dots \wedge EX_{\geq n} \wedge \dots$$

does not work since it is not a finite formula. The attempt

$$\forall n EX_{\geq n}$$

also fails; we cannot quantify over formulae in our logic.

Exercise 6. Explain precisely why these "formulae" are not admissible in predicate logic.

A Trick

After some more fruitless attempts one might suspect that the statement “there are infinitely many thingies” cannot be expressed in predicate logic.

Wrong! Let f be a unary function symbol and c a constant. Consider

$$\varphi = \forall x (f(x) \not\approx c) \wedge \forall x, y (f(x) \approx f(y) \rightarrow x \approx y).$$

So φ states that f is not surjective but injective. Hence in any interpretation that makes φ true the carrier set must be infinite.

Exercise 7. Use a total order to produce another formulae in predicate logic that forces the ground set to be infinite.

And the Entscheidungsproblem

Again, there are natural decision problems associated with this classification.

Problem: Validity

Instance: A predicate logic formula φ .

Question: Is φ valid?

Problem: Satisfiability

Instance: A predicate logic formula φ .

Question: Is φ satisfiable?

As usual, there is the search version of Satisfiability: we would like to construct a satisfying interpretation if one exists. Note that this may well entail construct an infinite structure.

As one might suspect from the few examples, these problems are much harder in predicate logic than in propositional logic and will turn out to be highly undecidable in general.

Defining Validity

Time to give a precise definition of validity and satisfiability. We begin by defining assignments in the context of predicate logic.

Definition 7. An **assignment** or **valuation** (over a structure \mathcal{A}) associates variables of the language with elements in the ground set A .

Given an assignment $\sigma : \text{Var} \rightarrow A$, we can associate an element $\sigma(t)$ in A with each term t .

- $t = x$: then $\sigma(t) = \sigma(x)$
- $t = f(r_1, \dots, r_n)$: then $\sigma(t) = f^{\mathcal{A}}(\sigma(r_1), \dots, \sigma(r_n))$

Example 2. Over \mathcal{N} let $\sigma(x) = 3$. Then $\sigma(x \cdot (1 + 1)) = 6$ whereas $\sigma(x) = 0$ produces $\sigma(x \cdot (1 + 1)) = 0$.

Atomic Formulae

Once we have an assignment for all the free variables in an atomic formula we can determine a truth value for it.

Definition 8. Let σ be an assignment over a structure \mathcal{A} and $\varphi = R(t_1, \dots, t_n)$ an atomic formula. Define the **truth value of φ (under σ over \mathcal{A})** to be

$$\mathcal{A}_\sigma(\varphi) = \begin{cases} \text{tt} & \text{if } R^{\mathcal{A}}(\sigma(t_1), \dots, \sigma(t_n)) \text{ holds,} \\ \text{ff} & \text{otherwise.} \end{cases}$$

Example 3. Over the natural numbers \mathcal{N} suppose $\sigma(x) = 0$ and $\sigma(y) = 1$. Then

$$\mathcal{N}_\sigma(x + y < 1 + 1) = \mathcal{N}_\sigma(0 + 1 < 1 + 1) = \text{tt}$$

but for $\sigma(x) = \sigma(y) = 1$ we get

$$\mathcal{N}_\sigma(x + y < 1 + 1) = \mathcal{N}_\sigma(1 + 1 < 1 + 1) = \text{ff}$$

Logical Connectives

Once we have a truth value for atomic formulae, we can extend this evaluation to compound formulae without quantifiers.

Definition 9. *Connectives*

- $\varphi = \psi \wedge \chi$: then $\mathcal{A}_\sigma(\varphi) = H_{\text{and}}(\mathcal{A}_\sigma(\psi), \mathcal{A}_\sigma(\chi))$
- $\varphi = \psi \vee \chi$: then $\mathcal{A}_\sigma(\varphi) = H_{\text{or}}(\mathcal{A}_\sigma(\psi), \mathcal{A}_\sigma(\chi))$
- $\varphi = \neg\psi$: then $\mathcal{A}_\sigma(\varphi) = H_{\text{not}}(\mathcal{A}_\sigma(\psi))$

Example 4. Suppose $\sigma(x) = 0$ and $\sigma(y) = 1$. Then

$$\begin{aligned} \mathcal{N}_\sigma(x < y \vee y < x) &= H_{\text{or}}(\mathcal{N}_\sigma(x < y), \mathcal{N}_\sigma(y < x)) \\ &= H_{\text{or}}(\mathcal{N}(0 < 1), \mathcal{N}(1 < 0)) \\ &= H_{\text{or}}(\text{tt}, \text{ff}) \\ &= \text{tt} \end{aligned}$$

Quantifiers

For an assignment σ , let us write $\sigma[a/x]$ for the assignment that is the same as σ everywhere, except that $\sigma[a/x](x) = a$. Think of this as substituting “ a for x ”.

Definition 10. *Quantifiers*

- $\varphi = \exists x \psi$:
Then $\mathcal{A}_\sigma(\varphi) = \text{tt}$ if there is an a in A such that $\mathcal{A}_{\sigma[a/x]}(\psi) = \text{tt}$.
- $\varphi = \forall x \psi$:
Then $\mathcal{A}_\sigma(\varphi) = \text{tt}$ if for all a in A $\mathcal{A}_{\sigma[a/x]}(\psi) = \text{tt}$.

Note that σ only needs to be defined on the free variables of φ to produce a truth value for φ , the values anywhere else do not matter. If φ is a sentence, σ can be totally undefined.

Validity and Models

Definition 11.

A formula φ is valid in \mathcal{A} under assignment σ if $\mathcal{A}_\sigma(\varphi) = 1$.

A formula φ is valid in \mathcal{A} if it is valid in \mathcal{A} for all assignments σ . The structure \mathcal{A} is then said to be a model for φ or to satisfy φ .

A sentence is valid (or true) if it is valid over any structure (of the appropriate signature).

Notation:

$$\mathcal{A} \models_\sigma \varphi, \quad \mathcal{A} \models \varphi, \quad \models \varphi$$

One uses the same notation for sets of formulae Γ . So $\mathcal{A} \models \Gamma$ means that $\mathcal{A} \models \varphi$ for all $\varphi \in \Gamma$.

Note the condition for validity: the formula has to hold in all structures.

Satisfiability

Definition 12. A formula is **satisfiable** if there is some structure \mathcal{A} and some assignment σ for all the free variables in φ such that $\mathcal{A} \models_\sigma \varphi$.

In other words, the existentially quantified formula

$$\exists x_1, \dots, x_n \varphi(x_1, \dots, x_n)$$

has a model, where x_1, \dots, x_n are all the free variables of φ .

In shorthand: $\exists \vec{x} \varphi(\vec{x})$.

This is analogous to validity where we insist that $\forall x_1, \dots, x_n \varphi(x_1, \dots, x_n)$ holds, or $\forall \vec{x} \varphi(\vec{x})$ in compact notation.

Isn't this all circular?

The definitions of truth given here is due to A. Tarski (two seminal papers, one in 1933 and a second one in 1956, with R. Vaught).

A frequent objection to this approach is that we are using "for all" to define what a universal quantifier means.

True, but the formulae in our logic are syntactic objects, and define their meaning in terms of structures, which are not syntactic, they are real (in the world of mathematics and TCS).

Think of this as a program: you can "compute" the truth value of a formula, as long as you can perform certain operations in the structure (evaluate f^A , loop over all elements, search over all elements, . . .). This is a bit problematic over infinite structures, but for finite ones we can actually perform the computation (at least if we ignore efficiency).

Computing Truth

More precisely, suppose we wanted to construct an algorithm

$$\text{ValidQ}(\mathcal{A}, \varphi)$$

that checks if formula φ is valid over structure \mathcal{A} .

What would be the appropriate input for such an algorithm?

The easy part is the formula: any standard representation will be fine.

The real problem is the structure \mathcal{A} .

For standard structures such as the natural numbers or reals we understand (more or less) how to interpret the operations. But in the general case we need some representation.

Boosting the Language

Suppose \mathcal{A} is a structure of some signature Σ . In order to describe \mathcal{A} the first step is to augment the language $\mathcal{L}(\Sigma)$ by constant symbols c_a for each element a in the carrier set A of \mathcal{A} , obtaining a new signature $\Sigma_{\mathcal{A}}$.

\mathcal{A} is naturally also a structure of signature $\Sigma_{\mathcal{A}}$.

This step is not necessary when there already are terms in the language for all the elements of the structure. E.g., in arithmetic we can denote every natural number by

$$1 + 1 + \dots + 1$$

This is clumsy but it does the job.

But think about the reals: we have only countably many terms in our language but there are uncountably many elements in the structure. We cannot begin to write all available facts about \mathbb{R} . But given the extra constants we can provide all requisite information about addition, multiplication and order – at least in principle.

The Diagram of a Structure

Definition 13. The (atomic) diagram of a structure of some fixed signature is the set of all atomic sentences and their negations in $\mathcal{L}(\Sigma_{\mathcal{A}})$ that are valid in \mathcal{A} .

In symbols: $\text{diag}_{\mathcal{A}}$.

The point is that the validity of any formula over \mathcal{A} is completely determined by $\text{diag}_{\mathcal{A}}$: no other information is used in our definition of truth. Hence our algorithm should take as inputs the diagram and the formula and use recursion to obtain the answer:

$$\text{ValidQ}(\text{diag}_{\mathcal{A}}, \varphi)$$

If the underlying structure \mathcal{A} is finite (and thus the diagram is finite), then we can actually perform this computation: it is just recursion and copious table lookups. In fact, ValidQ will be primitive recursive given any reasonable coding.

Cayley Tables

How do we represent the atomic diagram $\text{diag } \mathcal{A}$? Given enough constants we can simply write down table.

E.g., for a unary function symbol f we can use a table with entries c_a and c_b provided that $b = f^{\mathcal{A}}(a)$. Each entry corresponds to an identity $f(c_a) \approx c_b$ in the diagram.

For binary function symbols we get a classical Cayley style "multiplication table", and higher dimensional tables for functions of higher arity.

Relations can be handled by similar tables with entries in \mathbb{B} . This corresponds to the familiar interpretation of a relation $R \subseteq A^k$ as a function $R : A^k \rightarrow \mathbb{B}$.

So, the whole diagram is just a bunch of tables using special constants for all elements in the structure and Boolean values. For small structures this is a perfectly good representation, though even for finite but large structures explicit tables are not feasible.

Infinite Structures

Pushing ahead into the realm of infinite structures things become much more complicated. If the carrier set is uncountable our machinery from classical computability theory simply does not apply – the individual elements are not finitary objects and we have no handle.

However, if the carrier set is countable computability theory does apply and we can use it to measure the complexity of the structure.

Definition 14. The (complete) diagram is the collection of all sentences in $\mathcal{L}(\Sigma_{\mathcal{A}})$ that are valid in \mathcal{A} . In symbols: $\text{diag}^c \mathcal{A}$.

Definition 15. \mathcal{A} is computable if its atomic diagram is decidable. \mathcal{A} is decidable if its complete diagram is decidable.

All finite structures are trivially decidable and even primitive recursive, though things become more complicated if we consider lower levels of the computational hierarchy.

For example, satisfiability of a propositional formula is easily expressed as a validity problem of a formula over a two-element structure, yet no polynomial time algorithm is known for this problem.

Arithmetic

The standard structure of the natural numbers is a good example for a computable structure that fails to be decidable. The atomic sentences here come down to assertions of the type

$$(1 + 1) * 3 = 6$$

and the like and are trivially decidable. Indeed, this all comes down to evaluating polynomials over the integers and can be handled very easily.

Alas, the addition of quantifiers destroys any hope for effective decision methods: even just a single existential quantifier renders the validity problem undecidable.

In fact, truth over the structure of natural numbers is separated from decidability by infinitely many levels of a hierarchy of complexity.

Equality

Note that the equality symbol \approx is treated differently from other binary relation symbols: it is always interpreted as actual equality over \mathcal{A} . Hence the following formulae are all valid.

$$\forall x (x \approx x)$$

$$\forall x \forall y (x \approx y \rightarrow y \approx x)$$

$$\forall x \forall y \forall z (x \approx y \wedge y \approx z \rightarrow x \approx z)$$

$$\forall \vec{x} \forall \vec{y} (\vec{x} \approx \vec{y} \rightarrow (R(\vec{x}) \leftrightarrow R(\vec{y})))$$

$$\forall \vec{x} \forall \vec{y} (\vec{x} \approx \vec{y} \rightarrow f(\vec{x}) \approx f(\vec{y}))$$

Exercise 8. Explain what would happen if we adopt these formulae as axioms for an otherwise undistinguished binary relation symbol \approx . Describe the structures that satisfy these axioms.

Elementary Classes

Predicate Logic and Models

So why is predicate logic all that interesting? Experience has shown that the system is just expressive enough to pin down the properties of most mathematical objects. On the other hand, one can handle proofs in predicate logic rather well (see the next lecture).

To describe a class of structures one chooses an appropriate signature (function and relation symbols) and then selects a collection of formulae Γ in this language, called *axioms*.

Definition 16. For any collection Γ of sentences of predicate logic the class of models of Γ is defined to be

$$\text{Mod}(\Gamma) = \{ \mathcal{A} \mid \mathcal{A} \models \Gamma \}.$$

where all the structures have the appropriate signature.

If the axioms are chosen properly, the class $\text{Mod}(\Gamma)$ will consist precisely of the structures one is interested in.

Elementary Classes

Definition 17. A class of structures \mathcal{C} is **elementary** if there is a single sentence φ such that $\mathcal{C} = \text{Mod}(\varphi)$.

Note that this is the same as requiring a finite set Γ of sentences: we can simply form the conjunction.

As we will see, many examples from algebra such as groups and fields are elementary classes, but there are some notable exceptions, e.g., fields of characteristic 0.

And, some structures are not at all expressible as models of any set of sentences of predicate logic. One famous example is the class of all well-orderings.

Example: Monoid Axioms

Let's use the language

$$\mathcal{L}(*, e) \text{ of type } (2, 0)$$

So there is only one binary function symbol $*$, and one constant e .

We will write $*$ in infix notation to keep things readable and drop the universal quantifiers up front. There are 2 monoid axioms (take the universal closure):

$$x * (y * z) \approx (x * y) * z \quad (1)$$

$$x * e \approx x \wedge e * x \approx x \quad (2)$$

So axiom (1) says that $*$ is **associative** and axiom (2) says e is a **neutral element** for $*$.

Definition 18. A **monoid** is any model of these axioms. A **semigroup** is a model of just axiom (1).

A Small Example

In order to specify a (small) monoid, we only need to write down a **multiplication table** for the operation, and the neutral element.

Here are the Cayley tables of two simple examples that we are already familiar with:

*	0	1
0	0	1
1	1	1

*	0	1
0	0	0
1	0	1

This is just logical disjunction and conjunction and usually written as $\langle \mathbb{B}; \vee, 0 \rangle$ and $\langle \mathbb{B}; \wedge, 1 \rangle$.

Note that it is easy to determine the whole atomic diagram from the Cayley tables. E.g., $(1 * 1) * (1 * 1) = (1 * 1)$ is in the diagram.

Six Elements

A slightly larger Cayley table with elements $\{e, a, b, c, d, f\}$.

*	e	a	b	c	d	f
e	e	a	b	c	d	f
a	a	c	f	e	b	d
b	b	f	a	d	e	c
c	c	e	d	a	f	b
d	d	b	e	f	c	a
f	f	d	c	b	a	e

Clearly we can test in time cubic in the size of the structure whether it satisfies the associativity axiom. To check for the neutral element only requires linear time.

In general, given the full multiplication table we can certainly check in polynomial time if a structure is in fact a monoid.

Some Infinite Monoids

For infinite carries sets things become more interesting. Again, we have to specify a range for the variables, the operation $*$, and the constant e .

Here are some monoids:

- $\langle \mathbb{N}; +, 0 \rangle$
- $\langle \mathbb{N}^+; *, 1 \rangle$
- $\langle \mathbb{R}^+; *, 1 \rangle$
- $\langle \text{strings}; \text{concat}, \text{empty word} \rangle$
- $\langle \text{lists}; \text{join}, \text{empty list} \rangle$

Also note that it is not at all clear how one would go about verifying that these are indeed models of the monoid axioms: a brute force test is not possible here. We have to argue within a more powerful system.

Two Strange Monoids

Here is a less obvious example. Let \mathcal{F} be the collection of propositional formulae modulo equivalence. Then the following two structures are monoids.

$$\langle \mathcal{F}; \wedge, \top \rangle$$

$$\langle \mathcal{F}; \vee, \perp \rangle$$

Note that for this to work we must identify equivalent formulae: $p \wedge \top \neq p$ but certainly $p \wedge \top \equiv p$.

Exercise 9. What is the size of these monoids if we consider only formulae with n propositional variables?

Non-Models

Not just any old set with a binary operation is a monoid.

$$\langle \mathbb{N}^+; +, 1 \rangle$$

$$\langle \mathbb{R}^+; \text{exp}, 1 \rangle$$

$$\langle \text{binary trees}; \text{join}, \text{empty tree} \rangle$$

In the first example, 1 is not a neutral element for addition. The second operation, exponentiation, is not associative. Likewise, pasting together binary trees is not associative.

