CDM

First-Order Logic

Klaus Sutner
Carnegie Mellon University
Spring 2021

Designing a Logic

There are three major parts in the design of a logic:

- language (syntax)
- model theory (semantics)
- proof theory (deductions)

Describing a language is fairly straightforward, in particular if one is not interested in actually building a parser or implementing various algorithms.

Model theory (study of the corresponding structures) brushes up against set theory, at least for infinite models, and can be quite challenging. But note that there are no issues in the finite case.

Proof theory is arguably the most difficult part: while deductions are just finite data structures, understanding and organizing them in detail is hard.

Digression: Logic, the Field

According to the Handbook of Mathematical Logic, the field is organized into 4 areas:

- set theory
- recursion theory
- proof theory
- model theory

The syntax of logic languages is considered the easy part of model theory and proof theory.

Fair enough in math, but in the computational universe there are quite a few interesting algorithmic challenges.

Many Logics

There are many choices on how to organize a logic:

> propositional, equational, first-order, second-order, higher-order, modal, linear time temporal, branching time temporal, linear, intuitionist, ...

Which one is appropriate for a particular problem depends very much on the problem.

And, there is a trade-off: making the logic powerful usually results in computational complexity issues; keeping it weak limits applicability.

Convention: We are going to deal exclusively with classical first-order logic.

Wishlist

We need a logic that 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 we can select a prime with any given number of bits (this is just the tip of an iceberg).

First-Order Logic

So how do we construct a language suitable for (most of) mathematics and theoretical 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", "for all" 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 first-order logic (FOL) or predicate logic. In the 20th century, FOL emerged as the de facto workhorse in all of math and TCS.

The Language

The language of first-order logic consists of the following pieces:

constants	that denote individual objects,
variables	that range over individual objects,
relation symbols	that denote relations,
function symbols	that denote functions,
logical connectives	"and," "or," "not," "implies," ...
existential quantifiers	that express "there exists,"
universal quantifiers	that express "for all."

Notation

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

These are just conventions, there are no sacred cows here. E.g., we might also use subscripted symbols and additional connectives such as $\oplus$ or $\Leftrightarrow$.

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

The standard signature for group theory is $(2, 1, 0)$: one binary function symbol for group multiplication, one unary function symbol for the inverse operation, a nullary function symbol (constant) for the neutral element.
Alternatively, we could use signature (2) only—but at the cost of slightly more complicated axioms.

Large Languages

There are occasions when one wants to use countably many function or relation symbols. It is easy to describe languages of this form. For example, we could allow for constants c_n, $n \in \mathbb{N}$ (together with axioms $c_i \neq c_j$ for $i < j$ this would make sure that all models are infinite).

In mathematical logic one also considers languages of some higher cardinality $\kappa > \aleph_0$, but this will be of no interest to us. Note that some amount of set theory is needed just to define such a language. We don't have to worry about these.

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

$$\forall x, y\,(x + y = y + x)$$

$$\forall x\,(x \cdot 0 = 0)$$

$$\forall x\,\exists z\,((\neg x = 0) \Rightarrow z \cdot x = 1)$$

$$1 + 1 = 0$$

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 unital rings of characteristic 2 (assuming we are not in the zero ring).

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 $\bar{\ }$ (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.

Infinitude of Primes

In the language of arithmetic we can write a formula $\mathrm{prime}(x)$ that expresses the assertion "x is prime".

$$\mathrm{prime}(x) \equiv 1 < x \wedge \forall u, v\,(x = u \cdot v \Rightarrow u = 1 \vee v = 1)$$

Note that x here is a free variable (not quantified over) in $\mathrm{prime}(x)$. So, we can use free variables to define subsets of the domain.

We can now express the assertion "there are infinitely many primes."

$$\forall z\,\exists x\,(x > z \wedge \mathrm{prime}(x))$$

Warning: This is mildly misleading, FOL in general cannot express "there are infinitely many." The last formula works because we are in the context of arithmetic (where we force the domain to be a Dedekind-infinite set).

Expressiveness: Induction

As an example of the expressiveness of FOL consider the venerable Principle of Induction. We can express induction as a first-order formula as follows.

$$R(0) \wedge \forall x\,\big(R(x) \Rightarrow R(x+1)\big) \Rightarrow \forall x\,R(x)$$

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

If we wanted to quantify over all relations R we would need more horse power, say, second-order logic.

$$\forall R\,\Big(R(0) \wedge \forall x\,\big(R(x) \Rightarrow R(x+1)\big) \Rightarrow \forall x\,R(x)\Big)$$

Keeping it Simple

Second-order logic is dangerously close to set theory: exactly what are we quantifying over in $\forall R$?

The standard workaround is to use an axiom schema instead:

$$\varphi(0) \wedge \forall x\,\big(\varphi(x) \Rightarrow \varphi(x+1)\big) \Rightarrow \forall x\,\varphi(x)$$

Here $\varphi(x)$ is any formula of arithmetic with free variable x.

This works fine in basic applications, say, in number theory, but is nowhere near as powerful as second-order.

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 FOL. 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 FOL.

The quantification part is what really matters here; propositional connectives are the same as in propositional logic while terms and equations are 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:

$$\text{ar} : \Sigma \to \mathbb{N}$$

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 $\bot$.

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

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 $t_1, \ldots, t_n$ are terms, $n \geq 0$, then $f(t_1, \ldots, t_n)$ 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

An atomic formula is an expression of the form

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

where R is an n-ary relation symbol, and the $t_1, \ldots, t_n$ 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. If we wanted to be extra careful, we would select a special binary relation symbol **=** in our language, with the intent that

$$s = t$$

means that terms s and t denote the same element.

Thus, unlike with all the other relation symbols, the meaning of **=** is fixed once and for all: it is always interpreted as equality.

We will soon cave and write the standard equality symbol $=$ rather than **=**, it is always clear from context what is meant.

Fragments

The system just described is first-order logic with equality.

One can also consider first-order logic without equality (there is no direct way of asserting equality of terms).

Lastly, one can consider the fragment of FOL that has no function symbols nor equality, only relations (the actual predicate logic).

In the presence of equality, one can still fake functions to a degree by considering relations F such that

$$\forall x \exists y\, F(x,y) \wedge \forall x,u,v\, (F(x,u) \wedge F(x,v) \Rightarrow u = v)$$

Formulae

Definition

The set of formulae of FOL is defined by

- Every atomic formula is a 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.

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

Sloppy, but eminently readable, style

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

Rant: Quantifiers

When it comes to rendering quantifiers in math text, some authors appear to suffer from temporary insanity and try to make things as difficult to parse as humanly possible.

$$(\forall x\,)(\forall y\,)(\exists z\,) f(x,y) = g(z)$$

as opposed to

$$\forall x,y \exists z\, \big(f(x,y) = g(z)\big)$$

Implication

In many texts implication is written as $\varphi \to \psi$.

Alas, this notation becomes less than ideal when we also deal with functions or relations $f : A \to B$ or in the context of diagrams.

We will write $\varphi \Rightarrow \psi$ if we want to make sure the arrow is understood as implication.

Other old-fashioned notation: $\mathbf{C}(\varphi,\psi)$ and $\varphi \supset \psi$.

We assume that implication associates to the right, so

$$\varphi \Rightarrow \psi \Rightarrow \chi \quad \text{means} \quad \varphi \Rightarrow (\psi \Rightarrow \chi)$$

Sentences

Definition

A variable that is not in the range of a quantifier is free or unbound (as opposed to bound). We write $\mathrm{FV}(\varphi)$ for the set of free variables in φ. A formula without free variables is closed, or a sentence.

One often indicates free variables like so:

$\varphi(x,y)$	x and y may be free
$\exists x\, \varphi(x,y)$	only y may be free
$\forall y \exists x\, \varphi(x,y)$	closed.

Substitutions of terms for free variables are indicated like so:

$\varphi(s,t)$	replace x by s, y by t
$\varphi[s/x, t/y]$	replace x by s, y by t

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

Displaying Free Variables

The notation

$$\varphi(x_1, \ldots, x_n)$$

only expresses the fact that $\mathrm{FV}(\varphi) \subseteq \{x_1, \ldots, x_n\}$.

Note: this does not mean that they all actually occur, some or even all of them may be missing. This turns out to be preferable over the alternative.

This is really no different from saying that $2x^2 - y$ is a polynomial in variables x, y and z.

Exercise

Construct an algorithm that computes free variables and determines the scope of quantifiers.

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.

$$\forall\, x, y, z\, \big(x * (y * z) = (x * y) * z\big)$$

is somewhat less elegant and harder to read than

$$x * (y * z) = (x * y) * z$$

In algebra, the latter form is much preferred.

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.

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

Better: rename the x inside

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

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 FOL 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

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

Exercise

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

Exercise

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

Exercise

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

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 1930s 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 FOL can be evaluated.

Semantics: Structures

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

Definition

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:

$$\mathfrak{A} = \langle A; f_1, f_2, \ldots, R_1, R_2, \ldots \rangle$$

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,
- operations on these objects, and
- relations on these objects.

In the case where the carrier set is finite (actually, finite and small) we can in fact represent the whole FO structure by a suitable data structure (for example, explicit lookup tables). 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 FOL.

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).

$$f \text{ function symbol} \quad \rightsquigarrow \quad f^{\mathfrak{A}} \text{ a function in } \mathfrak{A}$$

$$R \text{ function symbol} \quad \rightsquigarrow \quad R^{\mathfrak{A}} \text{ a relation in } \mathfrak{A}$$

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

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

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

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 $+^{\mathfrak{N}}$, 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 $\mathfrak{N}$, 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

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

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

We use the numeral $\underline{n} = \underbrace{1 + 1 + \ldots + 1}_{n \text{times}}$ to represent the natural number n as a term. The set of primes is then

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

Numerals

Strictly speaking, $\underline{n} = \underbrace{1 + 1 + \ldots + 1}_{n \text{times}}$ is also sloppy, we really should define

$$\begin{aligned} \underline{0} &= 0 \qquad \text{the term, not the number} \\ \underline{n+1} &= +(\underline{n}, 1) \end{aligned}$$

by induction, so that $\underline{n}$ is a bonified term of our system.

One often avoids this level of precision, but for any algorithmic treatment there is no way around it.

The Reals

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

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

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 $\mathfrak{R}$: 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 FOL is valid if it holds over any structure of the appropriate signature.

Warning: We require to formula to hold over any first-order structure of the appropriate signature (so we can interpret the function and relation symbols).

Any, not just one or some.

So the formula $\forall x, y \, (x * y = y * x)$ is not valid, but it can be used as an axiom to filter out commutative operations.

A Convention

It is standard to assume that the carrier set of a first-order structure is not empty. Hence the formula

$$\exists x \, (x = x)$$

is valid. This turns out to be more convenient than allowing empty structures.

For example, $\forall x \, (x \neq x)$ is true in the empty structure, which looks strange.

Also, $\forall x \, (x \neq x) \wedge \bot$ has prenex normal form $\forall x \, (x \neq x \wedge \bot)$, but they are not equivalent over the empty structure: the former is false, the latter is true.

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 FOL. More precisely, let

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

be a tautology with propositional variables $p_1, p_2, \ldots, p_n$. Let $\psi_1, \ldots, \psi_n$ be arbitrary sentences of FOL. Then

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

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

Satisfiability, Informally

Again in analogy to propositional logic we can define satisfiability.

Definition

A formula of FOL 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 exactly by any structure $\mathfrak{A}$ that carries a reflexive transitive relation $R^{\mathfrak{A}}$. On the other hand,

$$\forall\, x \, (x \neq c)$$

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

$$\begin{aligned}
&\forall\, x \forall\, y \, \varphi(x, y) \Rightarrow \forall\, y \forall\, x \, \varphi(x, y) \\
&\exists\, x \exists\, y \, \varphi(x, y) \Rightarrow \exists\, y \exists\, x \, \varphi(x, y) \\
&\exists\, x \forall\, y \, \varphi(x, y) \Rightarrow \forall\, y \exists\, x \, \varphi(x, y)
\end{aligned}$$

Note, though, that the following is not valid:

$$\forall\, x \exists\, y \, \varphi(x, y) \;\; \Rightarrow \;\; \exists\, y \forall\, x \, \varphi(x, y)$$

Exercise

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 FOL? First, a formula which states that there are at most n elements.

$$\mathsf{EX}_{\leq n} = \exists\, x_1, \ldots, x_n \forall\, y \, (y = x_1 \vee \ldots \vee y = x_n)$$

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

$$\mathsf{EX}_{\geq n} = \exists\, x_1, \ldots, x_n \, (x_1 \neq x_2 \wedge x_1 \neq x_3 \wedge \ldots \wedge x_{n-1} \neq x_n)$$

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

$$\mathsf{EX}_{\geq n} \Rightarrow \mathsf{EX}_{\geq m}$$

is valid whenever $m \leq n$.

Infinity

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

$$\mathsf{EX}_{\geq 1} \wedge \mathsf{EX}_{\geq 2} \wedge \ldots \wedge \mathsf{EX}_{\geq n} \wedge \ldots$$

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

$$\forall\, n \, \mathsf{EX}_{\geq n}$$

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

Exercise

Explain precisely why these "formulae" are not admissible in FOL.

Dedekind's Trick

After some more fruitless attempts one might suspect that the statement "there are infinitely many thingies" cannot be expressed in FOL.

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

$$\varphi = \forall\, x \, (f(x) \neq c) \wedge \forall\, x, y \, (f(x) = f(y) \Rightarrow x = y).$$

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

Use a total order to produce another formulae in FOL 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 FOL formula φ.
Question:	Is φ valid?

Problem:	**Satisfiability**
Instance:	A FOL 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 the construction of an infinite structure.

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

Model Checking

In computer science one is interested in the following version of the Entscheidungsproblem, usually called the model checking problem. We are dealing with FOL here, other logics are more important in typical applications.

Problem:	**Model Checking**
Instance:	A FO formula φ, a FO structure $\mathfrak{A}$.
Question:	Is φ true over $\mathfrak{A}$?

Ideally we would like to have an algorithm ValidQ$(\mathfrak{A}, \varphi)$ that solves the problem. Alas, there are two major issues:

- In general, model checking must be undecidable.
- φ is easy to specify as a data structure, but it is entirely unclear how we could deal with $\mathfrak{A}$. As written, the problem makes no sense.

Defining Validity

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

Definition

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

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

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

Example

Over $\mathfrak{N}$ 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

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

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

Example

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

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

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

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

Logical Connectives

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

Definition (Propositional Connectives)

φ	$\mathfrak{A}_\sigma(\varphi)$
$\psi \wedge \chi$	$H_{\mathsf{and}}(\mathfrak{A}_\sigma(\psi), \mathfrak{A}_\sigma(\chi))$
$\psi \vee \chi$	$H_{\mathsf{or}}(\mathfrak{A}_\sigma(\psi), \mathfrak{A}_\sigma(\chi))$
$\neg\psi$	$H_{\mathsf{not}}(\mathfrak{A}_\sigma(\psi))$

Example

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

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

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" in σ.

Definition (Quantifiers)

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

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 (Formulae)

A formula φ is valid in $\mathfrak{A}$ under assignment σ if $\mathfrak{A}_\sigma(\varphi) = 1$.
A formula φ is valid in $\mathfrak{A}$ if it is valid in $\mathfrak{A}$ for all assignments σ. The structure $\mathfrak{A}$ 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:

$$\mathfrak{A} \models_\sigma \varphi, \qquad \mathfrak{A} \models \varphi, \qquad \models \varphi$$

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

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

Satisfiability

Definition

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

In other words, the existentially quantified formula

$$\exists x_1, \ldots, x_n \; \varphi(x_1, \ldots, x_n)$$

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

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

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

It is often more convenient to leave off the universal quantifiers.

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 $f^{\mathfrak{A}}$, 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).

Naming Things

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

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

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 a ground term

$$\underline{n} = 1 + 1 + \ldots + 1$$

This works since we are dealing with the naturals, but in general there is no reason why every element in a structure should be denoted by a term.

Set Theory versus Computation

For example, suppose we want to deal with the reals. The standard language has constants for 0 and 1 but nothing else.

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

Using the numeral trick, we can obtain terms for rationals (at least if we add division), but no more.

Just to write down all available facts about $\mathbb{R}$ we need to add uncountably many constants, one for each real other than the rationals. This is no problem at all in set theory.

But it wrecks the language: the new constants are not finitary data structures. We cannot even build a parser.

The Diagram of a Structure

Definition

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

In symbols: $\operatorname{diag} \mathfrak{A}$.

The point is that the validity of any formula over $\mathfrak{A}$ is completely determined by $\operatorname{diag} \mathfrak{A}$: no other information is used in our definition of truth. Hence our putative model checking algorithm could look like this:

$$\mathrm{ValidQ}(\operatorname{diag} \mathfrak{A}, \varphi)$$

If the underlying structure $\mathfrak{A}$ 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 diag $\mathfrak{A}$? Given enough constants we can simply write down a table.

E.g., for a unary function symbol f we can use a table with entries c_a and c_b provided that $b = f^{\mathfrak{A}}(a)$. Each entry corresponds to an identity $f(c_a) = c_b$ 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 $R : A^k \to \mathbb{B}$.

Finite Structures

So, the whole diagram is just a bunch of tables using special constants for all elements in the structure and Boolean values. For small finite structures this is a perfectly good representation.

For large finite structures the Cayley table approach works only in principle.

Also, often finite structures are parametrized (for example by the number of elements) and one really would like a solution uniformly in terms of the parameter. This is usually much harder than staring at a single finite structure.

We should also be wary of computational complexity. 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.

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

The (complete) diagram is the collection of all sentences in $\mathcal{L}(\Sigma_{\mathfrak{A}})$ that are valid in $\mathfrak{A}$.

$\mathfrak{A}$ is computable if its atomic diagram is decidable.

$\mathfrak{A}$ is decidable if its complete diagram is decidable.

In symbols: diag^c $\mathfrak{A}$.

Examples

Every finite structure is decidable.

The structure $\mathfrak{N}$ of arithmetic is computable, but not decidable, even for annoyingly simple formulae. diag^c $\mathfrak{N}$ is highly undecidable.

The structure $\mathbb{R}$ of the reals is decidable.

This may sound strange, but note that we are only dealing with formulae over $\mathcal{L}(+, \cdot, 0, 1; <)$. Surprisingly, for these one can check validity over $\mathbb{R}$ (Tarski, quantifier elimination).

Wild Idea

To get more interesting examples where model checking works, how about the following:

> Use structures where the carrier set and the operations are all specified by finite state machines.

So we have natural data structures that represent these so-called automatic structures.

One might suspect that they are fairly feeble, but the are actually quite powerful. More next time.

Digression: Definability

Suppose we have some first-order structure $\mathfrak{A}$ over carrier set A and a relation $R \subseteq A^n$.

Definition

R is first-order definable (over $\mathfrak{A}$) if there is a formula $\varphi(x_1, \ldots, x_n)$ such that

$$R = \{ (a_1, \ldots, a_n) \in A^n \mid \mathfrak{A} \models \varphi(a_1, \ldots, a_n) \}$$

Example

Primality is definable over $\mathfrak{N}$, the structure of arithmetic.

Reachability in graphs is not first-order definable: to assert the existence of a path requires quantification over sequences of arbitrary length (which is provably impossible in FOL).

Now consider the standard structure of arithmetic $\mathfrak{N}$. We have given a precise definition of computability of a map $f : \mathbb{N} \rightharpoonup \mathbb{N}$. Could there also be a characterization in terms of definability?

Theorem

A map $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is computable if, and only if, it can be defined by a Σ_1 formula over $\mathfrak{N}$:

$$f(x) \simeq y \Leftrightarrow \exists z\, \varphi(x, y, z)$$

where φ contains only bounded quantifiers.

Exercise

Proof the theorem.