Classical first-order predicate logic

This is a powerful extension of propositional logic. It is the most important logic of all.

In the remaining lectures, we will:

- explain predicate logic syntax and semantics carefully
- do English–predicate logic translation, and see examples from computing
- generalise *arguments* and *validity* from propositional logic to predicate logic
- consider ways of establishing validity in predicate logic:
  - truth tables — they don't work
  - direct argument — very useful
  - equivalences — also useful
  - natural deduction (sorry).

Why?

Propositional logic is quite nice, but not very expressive. Statements like

- the list is ordered
- every worker has a boss
- there is someone worse off than you

need something more than propositional logic to express.

Propositional logic can't express arguments like this one of De Morgan:

- A horse is an animal.
- Therefore, the head of a horse is the head of an animal.

6. Syntax of predicate logic

6.1 New atomic formulas

Up to now, we have regarded phrases such as *the computer is a Sun* and *Frank bought grapes* as atomic, without internal structure.

Now we look inside them.

We regard being a Sun as a *property* or *attribute* that a computer (and other things) may or may not have. So we introduce:

- A *relation symbol* (or *predicate symbol)* `Sun`.
  It takes 1 argument — we say it is *unary* or its 'arity' is 1.
- We can also introduce a relation symbol `bought`.
  It takes 2 arguments — we say it is *binary,* or its arity is 2.
- *Constants,* to name objects.
  Eg, `Heron`, `Frank`, `Room-308`, `grapes`.

Then `Sun(Heron)` and `bought(Frank,grapes)` are two new atomic formulas.

6.2 Quantifiers

So what? You may think that writing

`bought(Frank,grapes)`

is not much more exciting that what we did in propositional logic — writing

`Frank bought grapes.`

But predicate logic has machinery to vary the arguments to `bought`.

This allows us to express properties of the relation '`bought`'.

The machinery is called *quantifiers.*

What are quantifiers?

A quantifier specifies a quantity (of things that have some property).

Examples

- *All* students work hard.
- *Some* students are asleep.
- *Most* lecturers are lazy.
- *Eight out of ten* cats prefer it.
- *Noone* is stupider than me.
- *At least six* students are awake.
- *There are infinitely many* prime numbers.
- *There are more* PCs than there are Macs.

Quantifiers in predicate logic

There are just two:

- $\forall$ (or `(A)`): 'for all'
- $\exists$ (or `(E)`): 'there exists' (or 'some')

Some other quantifiers can be expressed with these. (They can also express each other.) But quantifiers like *infinitely many* and *more than* cannot be expressed in first-order logic in general. (They can in, e.g., second-order logic.)

How do they work?

We've seen expressions like $\mathtt{Heron}$, $\mathtt{Frank}$, etc. These are *constants,* like π, or e.

To express 'All computers are Suns' we need *variables* that can range over all computers, not just Heron, Texel, etc.

6.3 Variables

We will use *variables* to do quantification. We fix an infinite collection (set) V of variables: eg, $x, y, z, u, v, w, x_0, x_1, x_2, \ldots$
Sometimes I write x or y to mean 'any variable'.

As well as formulas like $\mathtt{Sun}(\mathtt{Heron})$, we'll write ones like $\mathtt{Sun}(x)$.

- Now, to say 'Everything is a Sun', we'll write $\forall x\, \mathtt{Sun}(x)$.
  This is read as: 'For all x, x is a Sun'.
- 'Something is a Sun', can be written $\exists x\, \mathtt{Sun}(x)$.
  'There exists x such that x is a Sun.'
- 'Frank bought a Sun', can be written

$$\exists x(\mathtt{Sun}(x) \wedge \mathtt{bought}(\mathtt{Frank}, x)).$$

  'There is an x such that x is a Sun and Frank bought x.'
  Or: 'For some x, x is a Sun and Frank bought x.'

See how the new internal structure of atoms is used.

We will now make all of this precise.

6.4 Signatures

Definition 6.1 (signature) *A* signature *is a collection (set) of constants, and relation symbols with specified arities.*

Some call it a *similarity type*, or *vocabulary,* or (loosely) *language.*

It replaces the collection of atoms we had in propositional logic.

We usually write L to denote a signature. We often write $c, d, \ldots$ for constants, and $P, Q, R, S, \ldots$ for relation symbols.

A simple signature

Which symbols we put in L depends on what we want to say.

For illustration, we'll use a handy signature L consisting of:

- constants $\texttt{Frank}$, $\texttt{Susan}$, $\texttt{Tony}$, $\texttt{Heron}$, $\texttt{Texel}$, $\texttt{Clyde}$, $\texttt{Room-308}$, and c
- unary relation symbols $\texttt{Sun}$, $\texttt{human}$, $\texttt{lecturer}$ (arity 1)
- a binary relation symbol $\texttt{bought}$ (arity 2).

Warning: things in L are just symbols — syntax. They don't come with any meaning. To give them meaning, we'll need to work out (later) what a *situation* in predicate logic should be.

6.5 Terms

Definition 6.2 (term) *Fix a signature L.*

1. *Any constant in L is an L-term.*
2. *Any variable is an L-term.*
3. *Nothing else is an L-term.*

A closed term or (as computer people say) ground term is one that doesn't involve a variable.

Examples of terms

$\texttt{Frank}$, $\texttt{Heron}$ (ground terms)
x, y, x_{56} (not ground terms)

Terms are for *naming objects.*
Terms are not true or false.

Later (§9), we'll throw in function symbols.

6.6 Formulas of first-order logic

Definition 6.3 (formula) *Fix L as before.*

1. *If R is an n-ary relation symbol in L, and $t_1, \ldots, t_n$ are L-terms, then $R(t_1, \ldots, t_n)$ is an atomic L-formula.*
2. *If t, t' are L-terms then $t = t'$ is an atomic L-formula. (Equality — very useful!)*
3. *$\top, \bot$ are atomic L-formulas.*
4. *If A, B are L-formulas then so are $(\neg A)$, $(A \wedge B)$ $(A \vee B)$, $(A \rightarrow B)$, and $(A \leftrightarrow B)$.*
5. *If A is an L-formula and x a variable, then $(\forall x\, A)$ and $(\exists x\, A)$ are L-formulas.*
6. *Nothing else is an L-formula.*

Binding conventions: as for propositional logic, plus: $\forall x, \exists x$ have same strength as $\neg$.

Examples of formulas

Below, we write them as the cognoscenti do. Use binding conventions to disambiguate.

- $\texttt{bought}(\texttt{Frank}, x)$
  We read this as: 'Frank bought x.'
- $\exists x\, \texttt{bought}(\texttt{Frank}, x)$
  'Frank bought something.'
- $\forall x(\texttt{lecturer}(x) \rightarrow \texttt{human}(x))$
  'Every lecturer is human.' [Important eg!]
- $\forall x(\texttt{bought}(\texttt{Tony}, x) \rightarrow \texttt{Sun}(x))$
  'Everything Tony bought is a Sun.'

More examples

- $\forall x(\texttt{bought}(\texttt{Tony}, x) \rightarrow \texttt{bought}(\texttt{Susan}, x))$
  'Susan bought everything that Tony bought.'
- $\forall x\, \texttt{bought}(\texttt{Tony}, x) \rightarrow \forall x\, \texttt{bought}(\texttt{Susan}, x)$
  'If Tony bought everything, so did Susan.' Note the difference!
- $\forall x \exists y\, \texttt{bought}(x, y)$
  'Everything bought something.'
- $\exists x \forall y\, \texttt{bought}(x, y)$
  'Something bought everything.'

You can see that predicate logic is rather powerful — and terse.

7. Semantics of predicate logic

7.1 Structures (situations in predicate logic)

Definition 7.1 (structure) *Let L be a signature. An L-structure (or sometimes (loosely) a model) M is a thing that*

- *identifies a non-empty collection (set) of objects (the domain or universe of M, written $dom(M)$),*
- *specifies what the symbols of L mean in terms of these objects.*

The interpretation in M of a constant is an *object in $dom(M)$*. The interpretation in M of a relation symbol is a *relation on $dom(M)$*. You will soon see relations in Discrete Mathematics I, course 142.

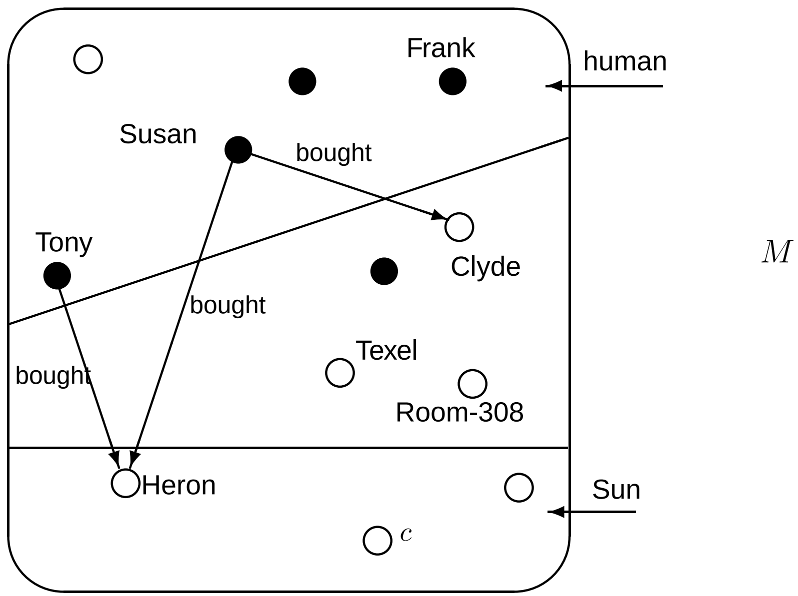
For our handy L, an L-structure should say:

- which objects are in its domain
- which of its objects are Tony, Susan, ...
- which objects are Suns, lecturers, human
- which objects bought which.

Example of a structure

Below is a diagram of a particular L-structure, called M (say).
There are 12 objects (the 12 dots) in the domain of M.
Some are labelled (eg 'Frank') to show the meanings of the constants of L (eg `Frank`).
The interpretations (meanings) of `Sun`, `human` are drawn as regions.
The interpretation of `lecturer` is indicated by the black dots.
The interpretation of `bought` is shown by the arrows between objects.

The structure M



Tony or `Tony`?

Do not confuse the object ● marked 'Tony' in $dom(M)$ with the constant `Tony` in L.
(I use different fonts, to try to help.)

They are quite different things. `Tony` is syntactic, while ● is semantic. In the context of M, `Tony` is a *name* for the object ● marked 'Tony'.

The following notation helps to clarify:

Notation 7.2 *Let M be an L-structure and c a constant in L. We write c^M for the interpretation of c in M. It is the object in $dom(M)$ that c names in M.*

So $\texttt{Tony}^M$ = the object ● marked 'Tony'.
In a different structure, `Tony` may name (mean) something else.

The meaning of a constant c *IS* the object c^M assigned to it by a structure M. A constant (and any symbol of L) has as many meanings as there are L-structures.

Drawing other symbols

Our signature L has only constants and unary and binary relation symbols.

For this L, we drew an L-structure M by

- drawing a collection of objects (the domain of M)
- marking which objects are named by which constants in M
- marking which objects M says satisfy the unary relation symbols (`human`, etc)
- drawing arrows between the objects that M says satisfy the binary relation symbols. The arrow direction matters.

If there were several binary relation symbols in L, we'd have to label the arrows.
In general, there's no easy way to draw interpretations of 3-ary or higher-arity relation symbols.
0-ary (nullary) relation symbols are the same as propositional atoms.

7.2 Truth in a structure (a rough guide)

When is a formula true in a structure?

- `Sun(Heron)` is true in M, because $\texttt{Heron}^M$ is an object ○ that M says is a `Sun`.
  We write this as $M \models \texttt{Sun(Heron)}$.
  Can read as 'M says `Sun(Heron)`'.

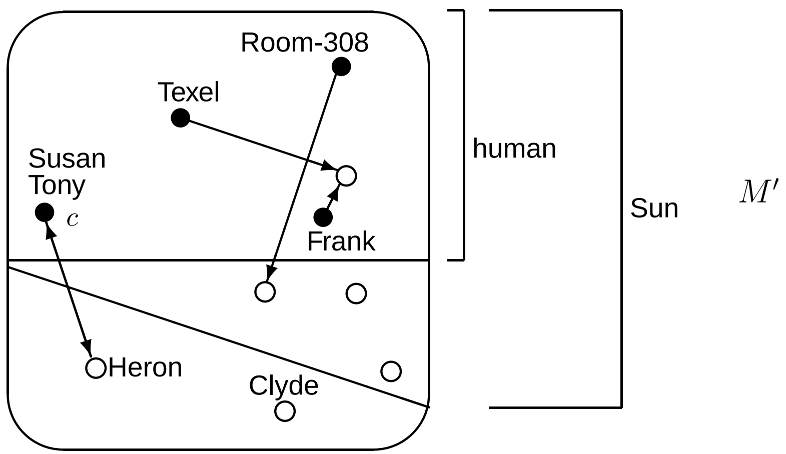
  Warning: This is a quite different use of $\models$ from definition 3.1.
  '$\models$' is *overloaded.*
- Similarly, $\texttt{bought}(\texttt{Susan}, \texttt{Clyde})$ is true in M.
  In symbols, $M \models \texttt{bought}(\texttt{Susan}, \texttt{Clyde})$.
- `bought(Susan,Susan)` is false in M, because M does not say that the constant `Susan` names an object ● that `bought` itself.
  In symbols, $M \not\models \texttt{bought}(\texttt{Susan}, \texttt{Susan})$.

From our knowledge of propositional logic,

- $M \models \neg\, \texttt{human}(\texttt{Room-308})$,
- $M \not\models \texttt{Sun}(\texttt{Tony}) \vee \texttt{bought}(\texttt{Frank}, \texttt{Clyde})$.

Another structure

Here's another L-structure, called M'.



Now, there are only 10 objects in $dom(M')$.

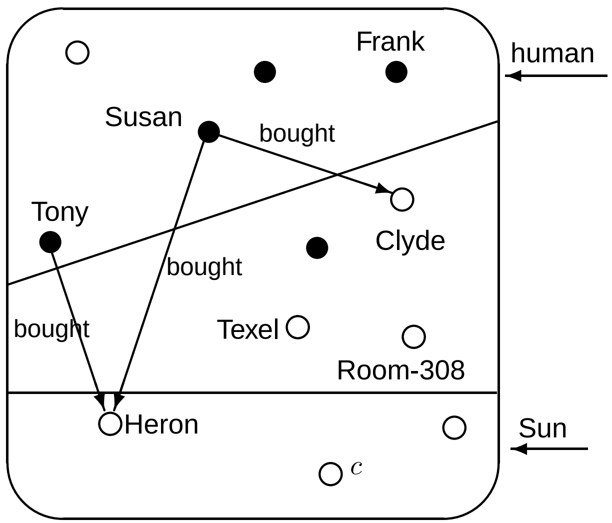
Some statements about M'

- $M' \not\models \texttt{bought}(\texttt{Susan}, \texttt{Clyde})$ this time.
- $M' \models \texttt{Susan} = \texttt{Tony}$.
- $M' \models \texttt{human}(\texttt{Texel}) \wedge \texttt{Sun}(\texttt{Texel})$.
- $M' \models \texttt{bought}(\texttt{Tony}, \texttt{Heron}) \wedge \texttt{bought}(\texttt{Heron}, c)$.

How about $\texttt{bought}(\texttt{Susan}, \texttt{Clyde}) \to \texttt{human}(\texttt{Clyde})$?
Or $\texttt{bought}(c, \texttt{Heron}) \to \texttt{Sun}(\texttt{Clyde}) \vee \neg\texttt{human}(\texttt{Texel})$?

Evaluating formulas with quantifiers

How do we work out if a formula with quantifiers is true in a structure?



M again

Evaluating quantifiers

$\exists x\, \texttt{bought}(x, \texttt{Heron})$ is true in M.
In symbols, $M \models \exists x\, \texttt{bought}(x, \texttt{Heron})$.
In English, 'something bought Heron'.

For this to be so, there must be an object x in $dom(M)$ such that $M \models \texttt{bought}(x, \texttt{Heron})$ — that is, M says that $\texttt{bought}(x, \bigcirc)$, where $\bigcirc = \texttt{Heron}^M$.

There is: we can take (eg.) x to be $\texttt{Tony}^M$.

Another example: $M \models \forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x))$

That is, 'for every object x in $dom(M)$,
$\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ is true in M'.
(We evaluate '$\to$' as in propositional logic.)

In M, there are 12 possible x. We need to check whether
$\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ is true in M for each of them.

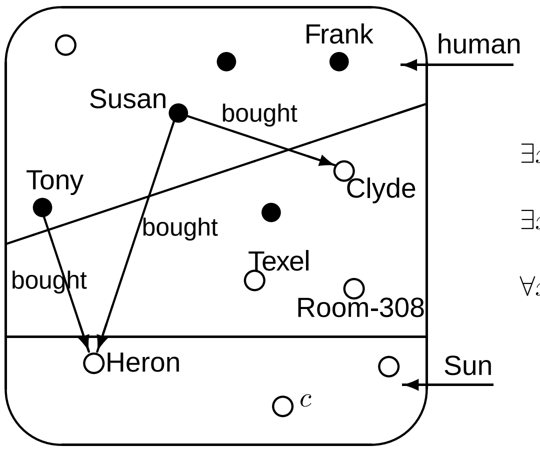
$\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ will be true in M for any object x such that $\texttt{bought}(\texttt{Tony}, x)$ is false in M. ('False $\to$ anything is true.') So we only need check the x for which $\texttt{bought}(\texttt{Tony}, x)$ is true.

The effect of '$\texttt{bought}(\texttt{Tony}, x) \to$' is to restrict the $\forall x$ to those x that Tony bought — here, just $\texttt{Heron}^M$.

For this object $\bigcirc$, $\texttt{bought}(\texttt{Susan}, \bigcirc)$ is true in M. So
$\texttt{bought}(\texttt{Tony}, \bigcirc) \to \texttt{bought}(\texttt{Susan}, \bigcirc)$ is true in M.

So $\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x)$ is true in M for *every* object x in M. Hence, $M \models \forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{bought}(\texttt{Susan}, x))$.

Exercise: which are true in M?



$\exists x(\texttt{Sun}(x) \land \texttt{bought}(\texttt{Frank}, x))$

$\exists x(\texttt{Sun}(x) \land \exists y\ \texttt{bought}(y, x))$

$\forall x(\texttt{lecturer}(x) \to \texttt{human}(x))$

7.3 Truth in a structure — formally!

We saw how to evaluate some formulas in a structure. Now we show how to evaluate arbitrary formulas.

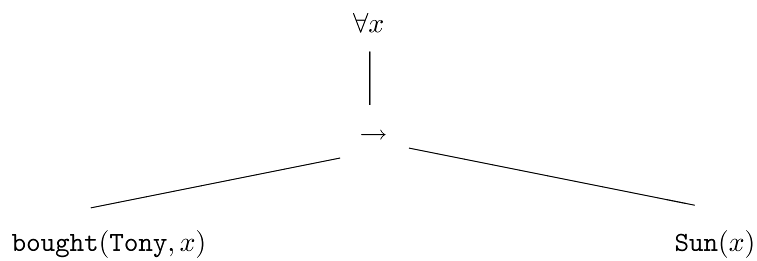
In propositional logic, we calculated the truth value of a formula in a situation by working up through its formation tree — from the atomic subformulas (leaves) up to the root.

For predicate logic, this is not so easy.
Not all formulas of predicate logic are true or false in a structure!

Example

$\forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{Sun}(x))$ is true in M (see slide 128).

Its formation tree is:



- Is $\texttt{bought}(\texttt{Tony}, x)$ true in M?!
- Is $\texttt{Sun}(x)$ true in M?!

Free and bound variables

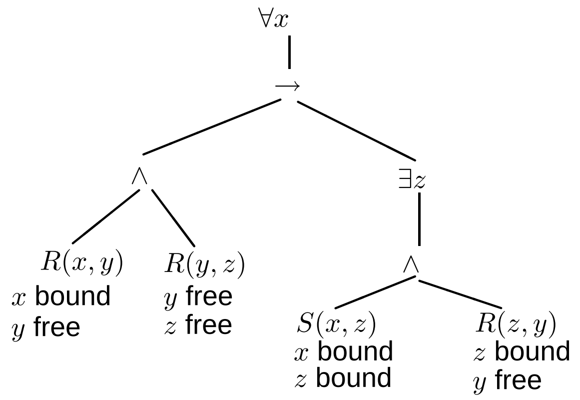
What's going on?
We'd better investigate how variables can arise in formulas.

Definition 7.3 *Let A be a formula.*

1. *An occurrence of a variable x in an atomic subformula of A is said to be* bound *if it lies under a quantifier $\forall x$ or $\exists x$ in the formation tree of A.*
2. *If not, the occurrence is said to be* free.
3. *The* free variables of A *are those variables with free occurrences in A.*

Example

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



The free variables of the formula are y, z.
Note: z has both free and bound occurrences.

Sentences

Definition 7.4 (sentence) *A* sentence *is a formula with no free variables.*

Examples

- $\forall x(\texttt{bought}(\texttt{Tony}, x) \to \texttt{Sun}(x))$ is a sentence.
- Its subformulas

$$\texttt{bought}(\texttt{Tony}, x) \to \texttt{Sun}(x),$$
$$\texttt{bought}(\texttt{Tony}, x),$$
$$\texttt{Sun}(x)$$

  are not sentences.

Which are sentences?

- $\texttt{bought}(\texttt{Frank}, \texttt{Texel})$
- $\texttt{bought}(\texttt{Susan}, x)$
- $x = x$
- $\forall x(x = y \to \exists y(y = x))$
- $\forall x \forall y(x = y \to \forall z(R(x,z) \to R(y,z)))$

The problem

Sentences are true or false in a structure.
But non-sentences are not!
A formula with free variables is neither true nor false in a structure M, because the free variables have no meaning in M. It's like asking 'is $x = 7$ true?'

We get stuck trying to evaluate a predicate formula in a structure in the same way as a propositional one, because the structure does not fix the meanings of variables that occur free. They are *variables,* after all.

Getting round the problem

So we must specify values for free variables, before evaluating a formula to true or false.
This is so even if it turns out that the values do not affect the answer (like $x = x$).

Assignments to variables

An *assignment* supplies the missing values of variables.

What a structure does for constants,
an assignment does for variables.

Definition 7.5 (assignment) *Let M be a structure. An* assignment (or 'valuation') into M *is something that allocates an object in $dom(M)$ to each variable.*
For an assignment h and a variable x, we write $h(x)$ for the object assigned to x by h.

[Formally, $h : V \to dom(M)$ is a function.]

Given an L-structure M *plus* an assignment h into M, we can evaluate:

- any L-term, to an object in $dom(M)$,
- any L-formula, to true or false.

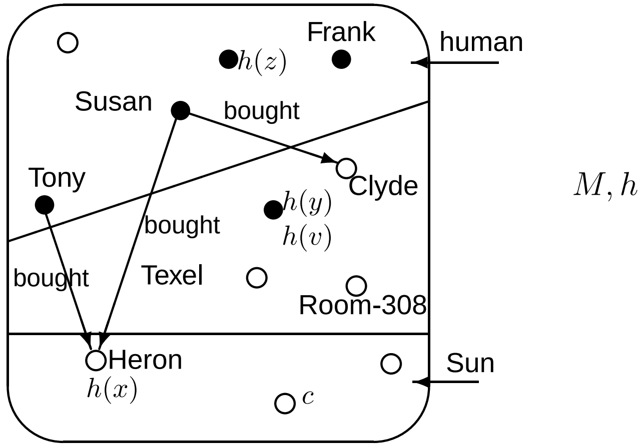
Evaluating terms (easy!)

Definition 7.6 (value of term) Let L be a signature, M an L-structure, h an assignment into M, and t an L-term.
The *value of t in M under h* is the object in M allocated to it by:

- M (if t is a constant),
- h (if t is a variable).

Example

(1) The value in M under h (below) of the term $\texttt{Tony}$ is the object ● marked 'Tony'. (From now on, I usually write just 'Tony' (or $\texttt{Tony}^M$, but NOT $\texttt{Tony}$) for it.)
(2) The value in M under h of x is Heron.



Semantics of atomic formulas

Fix an L-structure M and an assignment h. We define truth of a formula in M under h by working up the formation tree, as earlier.

Notation 7.7 ($\models$) *We write $M, h \models A$ if A is true in M under h, and $M, h \not\models A$ if not.*

Definition 7.8 (truth in M under h)

1. *Let R be an n-ary relation symbol in L, and $t_1, \ldots, t_n$ be L-terms. Suppose that the value of t_i in M under h is a_i, for each $i = 1, \ldots, n$ (see definition 7.6).*
   $M, h \models R(t_1, \ldots, t_n)$ if M says that the sequence $(a_1, \ldots, a_n)$ is in the relation R.
   If not, then $M, h \not\models R(t_1, \ldots, t_n)$.
2. *If t, t' are terms, then $M, h \models t = t'$ if t and t' have the same value in M under h.*
   If they don't, then $M, h \not\models t = t'$.
3. *$M, h \models \top$, and $M, h \not\models \bot$.*

Semantics of non-atomic formulas (definition 7.8 ctd.)

If we have already evaluated formulas A, B in M under h, then

4. *$M, h \models A \wedge B$ if $M, h \models A$ and $M, h \models B$.*
   Otherwise, $M, h \not\models A \wedge B$.
5. *$\neg A$, $A \vee B$, $A \to B$, $A \leftrightarrow B$*
   — similar: just as in propositional logic.

If x is any variable, then

6. $M, h \models \exists x A$ if there is some assignment g that agrees with h on all variables except possibly x, and such that $M, g \models A$.
   If not, then $M, h \not\models \exists x A$.
7. $M, h \models \forall x A$ if $M, g \models A$ for every assignment g that agrees with h on all variables except possibly x.
   If not, then $M, h \not\models \forall x A$.

'g agrees with h on all variables except possibly x' means that $g(y) = h(y)$ for all variables y *other than x*. (Maybe $g(x) = h(x)$ too!)

7.4 Useful notation for free variables

The books often write things like

'Let $A(x_1, \ldots, x_n)$ be a formula.'

This indicates that the free variables of A are among $x_1, \ldots, x_n$.

Note: $x_1, \ldots, x_n$ should all be different. And not all of them need actually occur free in A.

Example: if C is the formula

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

we could write it as

- $C(y, z)$
- $C(x, z, v, y)$
- C (if we're not using the useful notation)

but not as $C(x)$.

Notation for assignments

Fact 7.9 *For any formula A, whether or not $M, h \models A$ does not depend on $h(x)$ for any variable x that does not occur free in A.*

- So for a formula $A(x_1, \ldots, x_n)$, if $h(x_i) = a_i$ (each i), it's OK to write $M \models A(a_1, \ldots, a_n)$ instead of $M, h \models A$.
- Suppose we are explicitly given a formula C, such as

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

  If $h(y) = a$, $h(z) = b$, say, we can write

  $$M \models \forall x(R(x, a) \to \exists y S(y, b))$$

  instead of $M, h \models C$. Note: only the *free* occurrences of y in C are replaced by a. The bound y is unchanged.
- For a sentence S, we can just write $M \models S$, because by fact 7.9, whether $M, h \models S$ does not depend on h at all.

Working out $\models$ in this notation

Suppose we have an L-structure M, an L-formula $A(x, y_1, \ldots, y_n)$, and objects $a_1, \ldots, a_n$ in $dom(M)$.

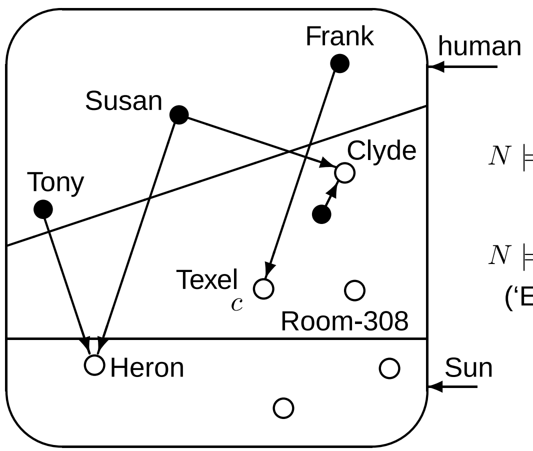
- To establish that $M \models (\forall x A)(a_1, \ldots, a_n)$ you check that $M \models A(b, a_1, \ldots, a_n)$ for each object b in $dom(M)$.

  You have to check even those b with no constants naming them in M. 'Not just Frank, Texel, ..., but all the other ○ and ● too.'
- To establish $M \models (\exists x A)(a_1, \ldots, a_n)$, you try to find some object b in the domain of M such that $M \models A(b, a_1, \ldots, a_n)$.

A is simpler than $\forall x A$ or $\exists x A$. So you can recursively work out if $M \models A(b, a_1, \ldots, a_n)$, in the same way. The process terminates.

7.5 Evaluating formulas in practice

Now we do some examples of evaluation. Let's have a new L-structure, say N. The black dots are the lecturers. The arrows indicate the interpretation in N of the relation symbol `bought`.



$N \models \exists x(\mathtt{bought}(x, \mathtt{Heron}) \wedge x = \mathtt{Susan})$?

$N \models \forall x(\mathtt{lecturer}(x) \to \exists y \mathtt{bought}(x, y))$?
('Every lecturer bought something.')

1. $N \models \exists x(\texttt{bought}(x, \texttt{Heron}) \land x = \texttt{Susan})$?

By definition 7.8, the answer is 'yes' just in case there is an object b in $dom(N)$ such that $N \models \texttt{bought}(b, \texttt{Heron}) \land b = \texttt{Susan}$.

We can find out by tabulating the results for each b in $dom(N)$. Write 1 for true, 0 for false.
$N \models \texttt{bought}(b, \texttt{Heron}) \land b = \texttt{Susan}$ if (and only if) we can find a b that makes the rightmost column a 1.

b	$B(b, \texttt{Heron})$	$b = \texttt{Susan}$	both
Tony	1	0	0
Susan	1	1	1
Frank	0	0	0
other ●	0	0	0
Room 308	0	0	0
⋮	0	0	0

But hang on, we only need *one* b making the RH column true. We already got one: Susan.

So yes, $N \models \exists x(\texttt{bought}(x, \texttt{Heron}) \land x = \texttt{Susan})$.

A bit of thought would have shown the only b with a chance is Susan. This would have shortened the work.

Moral: read the formula first!

2. $N \models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x, y))$?

$N \models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x, y))$ if and only if
$N \models \texttt{lecturer}(b) \to \exists y\ \texttt{bought}(b, y)$ for all b in $dom(N)$,
if and only if for each b in $dom(N)$, if $N \models \texttt{lecturer}(b)$ then there is d in $dom(N)$ such that $N \models \texttt{bought}(b, d)$.

So $N \models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x, y))$ if (and only if) for each b with $L(b) = 1$ in the table below, we can find a d with $B(b, d) = 1$.

$N \models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x, y))$ ctd

b	sample d	$L(b)$	$B(b, d)$
Tony	Heron	1	1
Susan	Tony	1	0
Frank	Clyde	1	0
other ●	Clyde	1	1
Heron	Susan	0	0
Room 308	Tony	0	0
⋮	⋮	0	?

$N \models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x,y))$ ctd

Reduced table with b just the lecturers:

b	sample d	$L(b)$	$B(b,d)$
Tony	Heron	1	1
Susan	Tony	1	0
Frank	Clyde	1	0
other ●	Clyde	1	1

We don't have 1s all down the RH column.

Does this mean $N \not\models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x,y))$?

$N \models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x,y))$ ctd

Not necessarily.
We might have chosen bad ds for Susan and Frank. $L(b)$ is true for them, so we must try to choose a d that b bought, so that $B(b,d)$ will be true.

And indeed, we can:

b	d	$L(b)$	$B(b,d)$
Tony	Heron	1	1
Susan	Heron	1	1
Frank	Texel	1	1
other ●	Clyde	1	1

shows $N \models \forall x(\texttt{lecturer}(x) \to \exists y\ \texttt{bought}(x,y))$.

Advice

How do we choose the 'right' d in examples like this? We have the following options:

1. In $\exists x$ or $\forall x$ cases, tabulate all possible values of d.

   In $\forall x \exists y$ cases, etc, tabulate all d *for each* b: that is, tabulate all pairs (b, d).
   (Very boring; no room to do it above.)

2. Try to see what properties d should have (above: being bought by b). Translating into English (see later on) should help. Then go for d with these properties.

3. Guess a few d and see what goes wrong. This may lead you to (2).

4. Use games. . . coming next.

How hard is it?

In most practical cases, it's easy to do the evaluation mentally, without tables, once used to it.

But in general, evaluation is hard.

Suppose that N is the natural numbers with the usual meanings of $prime$, $even$, $+$.

No-one knows whether

$$N \models \forall x(even(x) \wedge x > 2 \to \exists y \exists z(prime(y) \wedge prime(z) \wedge x = y + z)).$$

7.6 Hintikka games

Working out $\models$ by tables is clumsy. Often you can do the evaluation just by looking.

But if it's not immediately clear whether or not $M \models A$, it can help to use a game $G(M, A)$ with two players — me and you, say.

In $G(M, A)$, you are trying to show that $M \models A$, and I am testing you to see if you can.

There are two labels, '$\forall$' and '$\exists$'.

At the start, I am given the label $\forall$, and you get label $\exists$.

Playing $G(M, A)$

The game starts at the root of the formation tree of A, and works down node by node to the leaves. If the current node is:

- $\forall x$, then (the player labeled) $\forall$ chooses a value (in $dom(M)$) for the variable x
- $\exists x$, then $\exists$ chooses a value for x
- $\wedge$, then $\forall$ chooses the next node down
- $\vee$, then $\exists$ chooses the next node down
- $\neg$, then we swap labels ($\forall$ and $\exists$)
- $\rightarrow, \leftrightarrow$ — regard $A \rightarrow B$ as $\neg A \vee B$, and $A \leftrightarrow B$ as $(A \wedge B) \vee (\neg A \wedge \neg B)$.
- an atomic (or quantifier-free) formula, then we stop and evaluate it in M with the current values of variables.
  The player currently labeled $\exists$ wins the game if it's true, and the one labeled $\forall$ wins if it's false.

Winning strategies

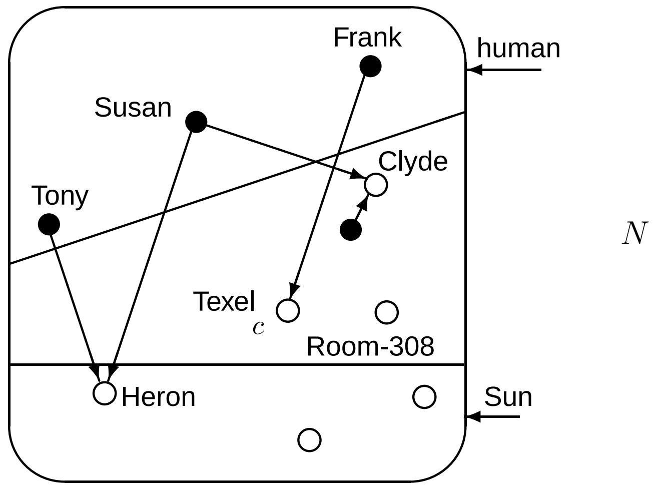
A *strategy* for a player in $G(M, A)$ is just a set of rules telling that player what to do in any position.

A strategy is *winning* if its owner wins any play (or match) of the game in which the strategy is used.

Theorem 7.10 *Let M be an L-structure. Then $M \models A$ if and only if you have a winning strategy in $G(M, A)$.*

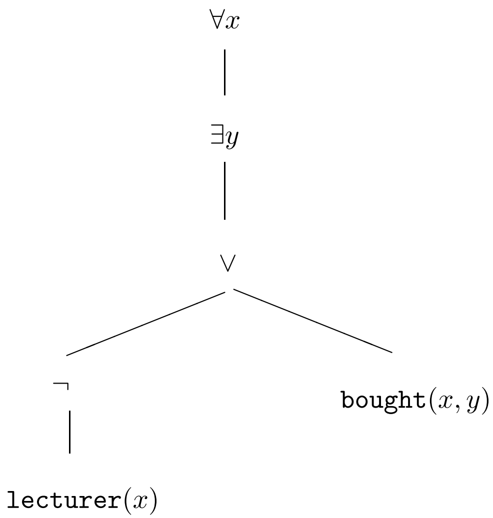
So your winning once is not enough for $M \models A$. You must be able to win however I play.

Let's play games on N

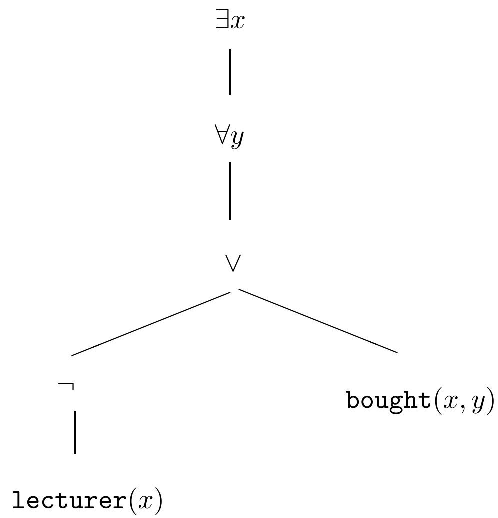


The black dots are the lecturers. The arrows indicate the interpretation in N of the relation symbol `bought`.

$N \models \forall x \exists y(\texttt{lecturer}(x) \to \texttt{bought}(x,y))$?



$N \models \exists y \forall x(\texttt{lecturer}(x) \to \texttt{bought}(x,y))$?



8. Translation into and out of logic

Translating predicate logic sentences from logic to English is not much harder than in propositional logic. But you can end up with a mess that needs careful simplifying.

$\forall x(\texttt{lecturer}(x) \wedge \neg(x = \texttt{Frank}) \to \texttt{bought}(x, \texttt{Texel}))$
'For all x, if x is a lecturer and x is not Frank then x bought Texel.'
'Every lecturer apart from Frank bought Texel.' (Maybe Frank did too.)

$\exists x \exists y \exists z(\texttt{bought}(x,y) \wedge \texttt{bought}(x,z) \wedge \neg(y = z))$
'There are x, y, z such that x bought y, x bought z, and y is not z.'
'Something bought at least two different things.'

$\forall x(\exists y \exists z(\texttt{bought}(x,y) \wedge \texttt{bought}(x,z) \wedge \neg(y = z)) \to x = \texttt{Tony})$
'For all x, if x bought two different things then x is equal to Tony.'
'Anything that bought two different things is Tony.'
CARE: it doesn't imply Tony did actually buy 2 things, just that noone else did.

English to logic

Hints for English-to-logic translation: express the sub-concepts in logic. Then build these pieces into a whole logical sentence.

Sample subconcepts:

- x buys y: $\texttt{bought}(x,y)$.
- x is bought: $\exists y\ \texttt{bought}(y,x)$.
- y is bought: $\exists z\ \texttt{bought}(z,y)$.
- x is a buyer: $\exists y\ \texttt{bought}(x,y)$.
- x buys at least two things:
  $\exists y \exists z(\texttt{bought}(x,y) \wedge \texttt{bought}(x,z) \wedge y \neq z)$.
  Here, $y \neq z$ abbreviates $\neg(y = z)$.

English-to-logic translation examples

- Every lecturer is human: $\forall x(\texttt{lecturer}(x) \to \texttt{human}(x))$.
- x is bought/has a buyer: $\exists y\, \texttt{bought}(y, x)$.
- Anything bought is not human:
  $\forall x(\exists y\, \texttt{bought}(y, x) \to \neg\, \texttt{human}(x))$.
  Note: $\exists y$ binds tighter than $\to$.
- Every Sun has a buyer: $\forall x(\texttt{Sun}(x) \to \exists y\, \texttt{bought}(y, x))$.
- Some Sun has a buyer: $\exists x(\texttt{Sun}(x) \wedge \exists y\, \texttt{bought}(y, x))$.
- All buyers are human lecturers:
  $\forall x(\underbrace{\exists y\, \texttt{bought}(x, y)}_{x \text{ is a buyer}} \to \texttt{human}(x) \wedge \texttt{lecturer}(x))$.
- No lecturer bought a Sun:
  $\neg\exists x(\texttt{lecturer}(x) \wedge \underbrace{\exists y(\texttt{bought}(x, y) \wedge \texttt{Sun}(y))}_{x \text{ bought a Sun}})$.

More examples

- Everything is a Sun or a lecturer (or both):
  $\forall x(\texttt{Sun}(x) \vee \texttt{lecturer}(x))$.
- Nothing is both a Sun and a lecturer:
  $\neg\exists x(\texttt{Sun}(x) \wedge \texttt{lecturer}(x))$, or
  $\forall x(\texttt{Sun}(x) \to \neg\texttt{lecturer}(x))$, or
  $\forall x(\texttt{lecturer}(x) \to \neg\texttt{Sun}(x))$, or
  $\forall x\neg(\texttt{Sun}(x) \wedge \texttt{lecturer}(x))$.
- Only Susan bought Clyde: $\forall x(\texttt{bought}(x, \texttt{Clyde}) \leftrightarrow x = \texttt{Susan})$.
- If Tony bought everything that Susan bought, and Tony bought a Sun, then Susan didn't buy a Sun:
  $\forall x(\texttt{bought}(\texttt{Susan}, x) \to \texttt{bought}(\texttt{Tony}, x))$
  $\wedge\ \exists y(\texttt{Sun}(y) \wedge \texttt{bought}(\texttt{Tony}, y))$
  $\to \neg\exists y(\texttt{Sun}(y) \wedge \texttt{bought}(\texttt{Susan}, y))$.
  (This may not be true! But we can still say it.)

Counting

- There is at least one Sun: $\exists x\, \texttt{Sun}(x)$.
- There are at least two Suns: $\exists x\exists y(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \wedge x \neq y)$,
  or (more deviously) $\forall x\exists y(\texttt{Sun}(y) \wedge y \neq x)$.
- There are at least three Suns:
  $\exists x\exists y\exists z(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \wedge \texttt{Sun}(z) \wedge x \neq y \wedge y \neq z \wedge x \neq z)$,
  or $\forall x\forall y\exists z(\texttt{Sun}(z) \wedge z \neq x \wedge z \neq y)$.
- There are no Suns: $\neg\exists x\, \texttt{Sun}(x)$
- There is at most one Sun: 3 ways:
  1. $\neg\exists x\exists y(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \wedge x \neq y)$
  2. $\forall x\forall y(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \to x = y)$
  3. $\exists x\forall y(\texttt{Sun}(y) \to x = y)$
- There's exactly 1 Sun: $\exists x\forall y(\texttt{Sun}(y) \leftrightarrow y = x)$.
- There are at most two Suns: 3 ways:
  1. $\neg$(there are at least 3 Suns)
  2. $\forall x\forall y\forall z(\texttt{Sun}(x) \wedge \texttt{Sun}(y) \wedge \texttt{Sun}(z) \to x = y \vee x = z \vee y = z)$
  3. $\exists x\exists y\forall z(\texttt{Sun}(z) \to z = x \vee z = y)$

Common patterns

You often need to say things like:

- 'All lecturers are human': $\forall x(\texttt{lecturer}(x) \to \texttt{human}(x))$.
  NOT $\forall x(\texttt{lecturer}(x) \wedge \texttt{human}(x))$.
  NOT $\forall x\, \texttt{lecturer}(x) \to \forall x\, \texttt{human}(x)$.
- 'All lecturers are human and not Suns':
  $\forall x(\texttt{lecturer}(x) \to \texttt{human}(x) \wedge \neg\texttt{Sun}(x))$.
- 'All human lecturers are Suns':
  $\forall x(\texttt{human}(x) \wedge \texttt{lecturer}(x) \to \texttt{Sun}(x))$.
- 'Some lecturer is a Sun': $\exists x(\texttt{lecturer}(x) \wedge \texttt{Sun}(x))$.

Patterns like $\forall x(A \to B)$, $\forall x(A \to B \wedge C)$, $\forall x(A \to B \vee C)$, and $\exists x(A \wedge B)$ are therefore common.

$\forall x(B \wedge C)$, $\forall x(B \vee C)$, $\exists x(B \wedge C)$, $\exists x(B \vee C)$ also crop up: they say every/some x is B and/or C.

$\exists x(A \to B)$ is *extremely rare.* If you write this, check to see if you've made a mistake.

9. Function symbols and sorts

— the icing on the cake.

9.1 Function symbols

In arithmetic (and Haskell) we are used to *functions,* such as $+, -, \times, \sqrt{x}, ++$, etc.

Predicate logic can do this too.

A *function symbol* is like a relation symbol or constant, but it is interpreted in a structure as a *function* (to be defined in discr math).

Any function symbol comes with a fixed arity (number of arguments).

We often write f, g for function symbols.

From now on, we adopt the following extension of definition 6.1:

Definition 9.1 (signature) *A* signature *is a collection of constants, and relation symbols and function symbols with specified arities.*

Terms with function symbols

We can now extend definition 6.2:

Definition 9.2 (term) *Fix a signature L.*

1. *Any constant of L is an L-term.*
2. *Any variable is an L-term.*
3. *If f is an n-ary function symbol of L, and $t_1, \ldots, t_n$ are L-terms, then $f(t_1, \ldots, t_n)$ is an L-term.*
4. *Nothing else is an L-term.*

Example

Let L have a constant c, a unary function symbol f, and a binary function symbol g. Then the following are L-terms:

- c
- $f(c)$
- $g(x, x)$
- $g(f(c), g(x, x))$

Semantics of function symbols

We need to extend definition 7.1 too: if L has function symbols, an L-structure must additionally define their meaning.

For any n-ary function symbol f in L, an L-structure M *must* say which object (in $dom(M)$) f associates with any sequence $(a_1, \ldots, a_n)$ of n objects in $dom(M)$. We write this object as $f^M(a_1, \ldots, a_n)$. There must be such a value.

[f^M is a function $f^M : dom(M)^n \to dom(M)$.]
A 0-ary function symbol is like a constant.

Examples

In arithmetic, M *might* say $+, \times$ are addition and multiplication of numbers: it associates 4 with $2 + 2$, 8 with 4×2, etc.

If the objects of M are vectors, M might say $+$ is addition of vectors and $\times$ is cross-product. M doesn't have to say this — it could say $\times$ is addition — but we may not want to use the symbol $\times$ in such a case.

Evaluating terms with function symbols

We can now extend definition 7.6:

Definition 9.3 (value of term) The value of an L-term t in an L-structure M under an assignment h into M is defined as follows:

- If t is a constant, then its value is the object in M allocated to it by M,
- If t is a variable, then its value is the object $h(t)$ in M allocated to it by h,
- If t is $f(t_1, \ldots, t_n)$, and the values of the terms t_i in M under h are already known to be $a_1, \ldots, a_n$, respectively, then the value of t in M under h is $f^M(a_1, \ldots, a_n)$.

So the value of a term in M under h is always *an object in $dom(M)$. Not true or false!*

Definition 7.8 needs no amendment, apart from using it with the extended definition 9.3.

We now have the standard system of first-order logic (as in books).

Arithmetic terms

A useful signature for arithmetic and for programs using numbers is the L consisting of:

- constants $\underline{\mathtt{0}}$, $\underline{\mathtt{1}}$, $\underline{\mathtt{2}}$, ... (I use underlined typewriter font to avoid confusion with actual numbers 0, 1, ...)
- binary function symbols $+, -, \times$
- binary relation symbols $<, \leq, >, \geq$.

We interpret these in a structure with domain $\{0,1,2,\ldots\}$ in the obvious way. But (eg) $34 - 61$ is unpredictable — can be any number.

We'll abuse notation by writing L-terms and formulas in infix notation:
• $x + y$, rather than $+(x, y)$,
• $x > y$, rather than $>(x, y)$.
Everybody does this, but it's breaking definitions 9.2 and 6.3, and it means we'll need to use brackets.

Some terms: $x + \underline{\mathtt{1}}$, $\quad \underline{\mathtt{2}} + (x + \underline{\mathtt{5}})$, $\quad (\underline{\mathtt{3}} \times \underline{\mathtt{7}}) + x$.
Formulas: $\underline{\mathtt{3}} \times x > \underline{\mathtt{0}}$, $\quad \forall x(x > \underline{\mathtt{0}} \to x \times x > x)$.

9.2 Many-sorted logic

As in typed programming languages, it sometimes helps to have structures with objects of different types. In logic, types are called *sorts.*

Eg some objects in a structure M may be lecturers, others may be Suns, numbers, etc.

We can handle this with unary relation symbols, or with *'many-sorted first-order logic'.*

Fix a collection $\mathrm{s}, \mathrm{s}', \mathrm{s}'', \ldots$ of sorts. How many, and what they're called, are determined by the application.

These sorts do *not* generate extra sorts, like $\mathrm{s} \to \mathrm{s}'$ or $(\mathrm{s}, \mathrm{s}')$.
If you want extra sorts like these, add them explicitly to the original list of sorts. (Their meaning would not be automatic, unlike in Haskell.)

Many-sorted terms

We adjust the definition of 'term' (definition 9.2), to give each term a sort:

- each variable and constant comes with a sort **s**, expressed as $c : \mathrm{s}$ and $x : \mathrm{s}$. There are infinitely many variables of each sort.

- each n-ary function symbol f comes with a template

$$f : (\mathrm{s}_1, \ldots, \mathrm{s}_n) \to \mathrm{s},$$

  where $\mathrm{s}_1, \ldots, \mathrm{s}_n, \mathrm{s}$ are sorts.
  Note: $(\mathrm{s}_1, \ldots, \mathrm{s}_n) \to \mathrm{s}$ is not itself a sort.

- For such an f and terms $t_1, \ldots, t_n$, if t_i has sort s_i (for each i) then $f(t_1, \ldots, t_n)$ is a term of sort s.
  Otherwise (if the t_i don't all have the right sorts), $f(t_1, \ldots, t_n)$ is not a term — it's just rubbish, like)∀)→.

Formulas in many-sorted logic

- Each n-ary relation symbol R comes with a template $R(\mathrm{s}_1, \ldots, \mathrm{s}_n)$, where $\mathrm{s}_1, \ldots, \mathrm{s}_n$ are sorts.
  For terms $t_1, \ldots, t_n$, if t_i has sort s_i (for each i) then $R(t_1, \ldots, t_n)$ is a formula. Otherwise, it's rubbish.
- $t = t'$ is a formula if the terms t, t' have the same sort. Otherwise, it's rubbish.
- Other operations ($\wedge, \neg, \forall, \exists$, etc) are unchanged. But it's polite to indicate the sort of a variable in $\forall, \exists$ by writing

$$\forall x : \mathrm{s}\ A \qquad \text{and} \qquad \exists x : \mathrm{s}\ A$$

instead of just

$$\forall x A \qquad \text{and} \qquad \exists x A$$

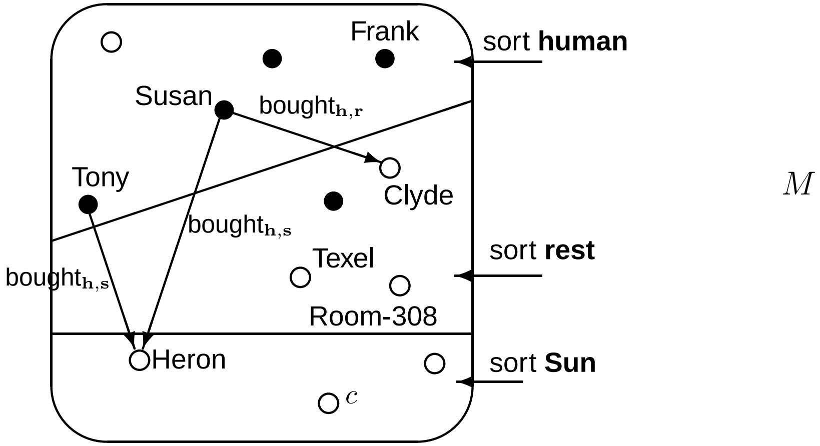
  if x has sort **s**.

This all sounds complicated, but it's very simple in practice.
Eg, you can write $\forall x : \mathbf{lecturer}\ \exists y : \mathbf{Sun}(\mathtt{bought}(x, y))$
instead of $\forall x(\mathtt{lecturer}(x) \to \exists y(\mathtt{Sun}(y) \wedge \mathtt{bought}(x, y)))$.

L-structures for many-sorted L

Let L be a many-sorted signature, with sorts $\mathrm{s}_1, \mathrm{s}_2, \ldots$

An L-structure is defined as before (definition 7.1 + slide 166), but additionally it allocates *each* object in its domain to *a single sort* (one of $\mathrm{s}_1, \mathrm{s}_2, \ldots$). So it looks like:



We need a binary relation symbol $\mathtt{bought}_{\mathrm{s},\mathrm{s}'}$ for *each* pair $(\mathrm{s}, \mathrm{s}')$ of sorts.

`lecturer` (black dots) must be implemented as 2 or 3 relation symbols, because (as in Haskell) each object has only 1 sort, not 2. (Alternative: use sorts for human lecturer, non-human lecturer, etc — all possible types of object.)

Interpretation of L-symbols

A many-sorted L-structure M must say:

- for each constant $c : \mathrm{s}$ in L, which object of sort **s** in $dom(M)$ is 'named' by c
- for each relation symbol $R : (\mathrm{s}_1, \ldots, \mathrm{s}_n)$ in L, and all objects $a_1, \ldots, a_n$ in $dom(M)$ of sorts $\mathrm{s}_1, \ldots, \mathrm{s}_n$, respectively, whether $R(a_1, \ldots, a_n)$ is true or not.

  It doesn't say anything about $R(b_1, \ldots, b_n)$ if $b_1, \ldots, b_n$ don't all have the right sorts.
- for each function symbol $f : (\mathrm{s}_1, \ldots, \mathrm{s}_n) \to \mathrm{s}$ in L and all objects $a_1, \ldots, a_n$ in $dom(M)$ of sorts $\mathrm{s}_1, \ldots, \mathrm{s}_n$, respectively, which object $f^M(a_1, \ldots, a_n)$ of sort **s** is associated with $(a_1, \ldots, a_n)$ by f.

  It doesn't say anything about $f(b_1, \ldots, b_n)$ if $b_1, \ldots, b_n$ don't all have the right sorts.

10. Application of logic: specifications

A *specification* is a description of what a program should do.

It should state the inputs and outputs (and their types).

It should include conditions on the input under which the program is guaranteed to operate. This is the *pre-condition.*

It should state what is required of the outcome in all cases (output for each input). This is the *post-condition.*

- The type (in the function header) is part of the specification.
- The pre-condition refers to the inputs (only).
- The post-condition refers to the outputs and inputs.

Precision is vital

A specification should be unambiguous. It is a *CONTRACT:*

Programmer wants pre-condition and post-condition to be the same — less work to do! The weaker the pre-condition and/or stronger the post-condition, the more work for the programmer — fewer assumptions (so more checks) and more results to produce.

Customer wants weak pre-condition and strong post-condition, for added value — less work before execution of program, more gained after execution of it.

Customer guarantees pre-condition so program will operate. Programmer guarantees post-condition, provided that the input meets the pre-condition.

If customer (user) provides the pre-condition (on the inputs), then provider (programmer) will guarantee the post-condition (between inputs and outputs).

10.1 Logic for specifying Haskell programs

A very precise way to specify properties of Haskell programs is to use first-order logic.
(Logic can also be used for Java, etc)

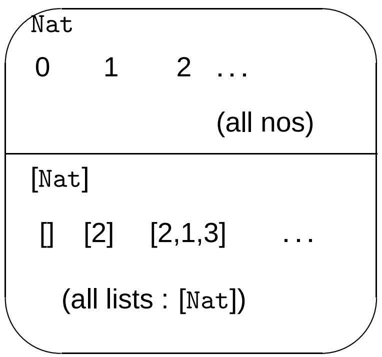
We use many-sorted logic, so we can have a sort for each Haskell type we want.

Example: lists of type [`Nat`]

Let's have a sort `Nat`, for $\{0,1,2,\ldots\}$, and a sort [`Nat`] for lists of natural numbers.
(Using the real Haskell `Int` is more longwinded: must keep saying $n \geq 0$ etc.)

The idea is that the structure's domain should look like:

2-sorted structure M



10.2 Signature for lists

The signature should be chosen to provide access to the objects in such a structure.

We want [], : (cons), ++, head, tail, $\sharp$, !!.

How do we represent these using constants, function symbols, or relation symbols?

How about a constant $[] : [\mathtt{Nat}]$ for the empty list, and function symbols

- $\mathtt{cons} : (\mathtt{Nat}, [\mathtt{Nat}]) \to [\mathtt{Nat}]$
- $++ : ([\mathtt{Nat}], [\mathtt{Nat}]) \to [\mathtt{Nat}]$
- $\mathtt{head} : [\mathtt{Nat}] \to \mathtt{Nat}$
- $\mathtt{tail} : [\mathtt{Nat}] \to [\mathtt{Nat}]$
- $\sharp : [\mathtt{Nat}] \to \mathtt{Nat}$
- $!! : ([\mathtt{Nat}], \mathtt{Nat}) \to \mathtt{Nat}$

Problem: `tail` etc are partial operations

In first-order logic, a structure *must* provide a meaning for function symbols *on all possible arguments* (of the right sorts).
What is the head or tail of the empty list? What is $xs\,!!\,\sharp(xs)$? What is $34-61$?

Two solutions (for `tail`):

1. Choose an arbitrary value (of the right sort) for $\mathtt{tail}([])$, etc.
2. Use a relation symbol `Rtail`([`Nat`],[`Nat`]) instead of a function symbol $\mathtt{tail} : [\mathtt{Nat}] \to [\mathtt{Nat}]$. Make $\mathtt{Rtail}(xs, ys)$ true just when ys is the tail of xs. If xs has no tail, $\mathtt{Rtail}(xs, ys)$ will be false for all ys.

Similarly for `head`, !!. E.g., use a function symbol $!! : ([\mathtt{Nat}], \mathtt{Nat}) \to \mathtt{Nat}$, and choose arbitrary value for $!!(xs, n)$ when $n \geq \sharp(xs)$. Or use a relation symbol $!!([\mathtt{Nat}], \mathtt{Nat}, \mathtt{Nat})$.

We'll take the function symbol option (1), as it leads to shorter formulas. But we must beware: values of functions on 'invalid' arguments are 'unpredictable'.

Lists in first-order logic: summary

Now we can define a signature L suitable for lists of type [`Nat`].

- L has constants $\underline{0}, \underline{1}, \ldots : \mathtt{Nat}$, relation symbols $<, \leq, >, \geq$ of sort (`Nat`,`Nat`), a constant $[] : [\mathtt{Nat}]$, and function symbols $+, -$, : or `cons`, ++, `head`, `tail`, $\sharp$, !!, with sorts as specified 2 slides ago.

  We write the constants as $\underline{0}$, $\underline{1}$,... to avoid confusion with actual numbers 0, 1, ... We write symbols in infix notation where appropriate.
- Let $x, y, z, k, n, m \ldots$ be variables of sort `Nat`, and $xs, ys, zs, \ldots$ variables of sort $[\mathtt{Nat}]$.
- Let M be an L-structure in which the objects of sort `Nat` are the natural numbers 0, 1, ..., the objects of sort [`Nat`] are all possible lists of natural numbers, and the L-symbols are interpreted in the natural way: ++ as concatenation of lists, etc. (Define $34-61$, $\mathtt{tail}([])$, etc. arbitrarily.)

  See figure, 3 slides ago.

10.3 Saying things about lists

Now we can say *a lot* about lists.
E.g., the following L-sentences, expressing the definitions of the function symbols, are true in M, because (as we said) the L-symbols are interpreted in M in the natural way:

$\sharp([]) = \underline{0}$

$\forall x \forall xs((\sharp(x{:}xs) = \sharp(xs) + \underline{1}) \wedge ((x{:}xs)!!\underline{0} = x))$

$\forall x \forall xs \forall n(n < \sharp(xs) \to (x{:}xs)!!(n + \underline{1}) = xs!!n)$
Note the '$n < \sharp(xs)$': $xs!!n$ could be anything if $n \geq \sharp(xs)$.

$\forall xs(\sharp(xs) = \underline{0} \vee \mathtt{head}(xs) = xs!!\underline{0})$

$\forall xs(xs \neq [] \to \sharp(\mathtt{tail}(xs)) = \sharp(xs) - \underline{1})$
$\forall xs \forall n(\underline{0} < n \wedge n < \sharp(xs) \to xs!!n = \mathtt{tail}(xs)!!(n - \underline{1}))$

$$\begin{aligned}\forall xs \forall ys \forall zs(xs = ys{+}{+}zs \leftrightarrow \\ \sharp(xs) = \sharp(ys) + \sharp(zs) \wedge \forall n(n < \sharp(ys) \to xs!!n = ys!!n) \\ \wedge \forall n(n < \sharp(zs) \to xs!!(n + \sharp(ys)) = zs!!n).\end{aligned}$$

10.4 Specifying Haskell functions

Now we know how to use logic to say things about lists, we can use logic to specify Haskell functions.

Pre-conditions in logic

These express restrictions on the arguments or parameters that can be legally passed to a function. *You write a formula* $A(a, b)$ that is true if and only if the arguments a, b satisfy the intended pre-condition (are legal).

Eg for the function $\log(x)$, you'd want a pre-condition of $x > \underline{0}$. For $\sqrt{x}$ you'd want $x \geq \underline{0}$.

Pre-conditions are usually very easy to write:

- xs is not empty: use $xs \neq []$.
- n is non-negative: use $n \geq \underline{0}$.

Type information

This is *not* part of the pre-condition.

If there are no restrictions on the arguments beyond their typing information, you can write 'none', or $\top$, as pre-condition.

This is perfectly normal and is no cause for alarm.

Post-conditions in logic

These express the required connection between the input and output of a function.

To do post-conditions in logic, *you write a formula* expressing the intended value of a function in terms of its arguments.

The formula should have free variables for the arguments, and should involve the function call so as to describe the required value. The formula should be true if and only if the output is as intended.

```
addone :: Nat -> Nat
-- pre:none
-- post:addone n = n+1
OR, in another commonly-used style,
-- post: z = n+1 where z = addone n
```


Specifying the function 'in'

```
in :: Nat -> [Nat] -> Bool
-- pre:none
-- post: in x xs <--> (E)k:Nat(k<#xs & xs!!k=x)
```

I used `(E)` and `&` as can't type $\exists, \wedge$ in Haskell.

Similarly, use `\/` for $\vee$, `(A)` for $\forall$, `~` for $\neg$.

Existence, non-uniqueness of result

Suppose you have a post-condition $A(x, y, z)$, where the variables x, y represent the input, and z represents the output.

Idea: for inputs a, b in M, the function should return some c such that $M \models A(a, b, c)$.

There is no requirement that c be unique: could have $M \models A(a, b, c) \wedge A(a, b, d) \wedge c \neq d$. Then the function could legally return c or d. It can return *any* value satisfying the post-condition.

But should arrange that $M \models \exists z A(a, b, z)$ whenever a, b meet the pre-condition: otherwise, the function cannot meet its post-condition.

So need $M \models \forall x \forall y(pre(x, y) \rightarrow \exists z\ post(x, y, z))$, for functions of 2 arguments with pre-, post-conditions given by formulas $pre, post$.

10.5 Examples

Saying something is in a list

$\exists k(k < \sharp(xs) \wedge xs!!k = n)$ says that n occurs in xs. So does $\exists ys\exists zs(xs = ys{+}{+}(n{:}zs))$.

Write $in(n, xs)$ for either of these formulas.

Then for any number a and list bs in M, we have $M \models in(a, bs)$ just when a occurs in bs.

So can specify a Haskell function for in:

```
isin :: Nat -> [Nat] -> Bool
-- pre: none
-- post: isin n xs <--> (E)ys,zs(xs=ys++n:zs)
```

The code for `isin` may in the end be very different from the post-condition(!), but `isin` should meet its post-condition.

Least entry

$in(m, xs) \wedge \forall n(n < \sharp(xs) \to xs!!n \geq m)$
expresses that (is true in M iff) m is the least entry in list xs.

So could specify a function `least`:

```
least :: [Nat] -> Nat
-- pre: input is non-empty
-- post: in(m,xs) & (A)n(n<#xs -> xs!!n>=m), where m = least xs
```

Ordered (or sorted) lists

$\forall n\forall m(n < m \wedge m < \sharp(xs) \to xs!!n \leq xs!!m)$ expresses that list xs is ordered. So does $\forall ys\forall zs\forall m\forall n(xs = ys{+}{+}(m{:}(n{:}zs)) \to m{\leq}n)$.

Exercise: specify a function

```
sorted ::  [Nat] -> Bool
```

that returns true if and only if its argument is an ordered list.

Merge

Informal specification:

```
merge :: [Nat] -> [Nat] -> [Nat] -> Bool
-- pre:none
-- post:merge(xs,ys,zs) holds when xs, ys are
-- merged to give zs, the elements of xs and ys
-- remaining in the same relative order.
```

merge([1,2],[3,4,5],[1,3,4,2,5]) and
merge([1,2],[3,4,5],[3,4,1,2,5]) are true.

merge([1,2],[3,4,5],[1]) and
merge([1,2],[3,4,5],[5,4,3,2,1]) are false.

Specifying 'merge'

Quite hard to specify explicitly (challenge for you!).

But can write an implicit specification:

$\forall xs\forall zs(\texttt{merge}(xs, [], zs) \leftrightarrow xs = zs)$

$\forall ys\forall zs(\texttt{merge}([], ys, zs) \leftrightarrow ys = zs)$

$$\forall x \ldots zs[\texttt{merge}(x{:}xs, y{:}ys, z{:}zs) \leftrightarrow (x = z \wedge \texttt{merge}(xs, y{:}ys, zs) \\ \vee\ y = z \wedge \texttt{merge}(x{:}xs, ys, zs))]$$

This pins down `merge` exactly: there exists a unique way to interpret a 3-ary relation symbol `merge` in M so that these three sentences are true. So they could form a post-condition.

Count

Can use `merge` to specify other things:

```
count : Nat -> [Nat] -> Nat
-- pre:none
-- post (informal): count x xs  = number of x's in xs
-- post: (E)ys,zs(merge ys zs xs
--         & (A)n:Nat(in(n,ys) -> n=x)
--         & (A)n:Nat(in(n,zs) -> n<>x)
--         & count x xs = #ys)
```

Idea: ys takes all the x from xs, and zs takes the rest. So the number of x is $\sharp(ys)$.

Conclusion

First-order logic is a valuable and powerful way to specify programs precisely, by writing first-order formulas expressing their pre- and post-conditions.
More on this in 141 'Reasoning about Programs' next term.

11. Arguments, validity

Predicate logic is much more expressive than propositional logic. But our experience with propositional logic tells us how to define 'valid argument' etc.

Definition 11.1 (valid argument) *Let L be a signature and $A_1, \ldots, A_n, B$ be L-formulas.*
An argument '$A_1, \ldots, A_n$, therefore B' is valid if for any L-structure M and assignment h into M,
if $M, h \models A_1$, $M, h \models A_2$, ..., and $M, h \models A_n$, then $M, h \models B$.
We write $A_1, \ldots, A_n \models B$ in this case.

This says: in any situation (structure + assignment) in which $A_1, \ldots, A_n$ are all true, B must be true too.

Special case: $n = 0$. Then we write just $\models B$. It means that B is true in every L-structure under every assignment into it.

Validity, satisfiability, equivalence

These are defined as in propositional logic.

Definition 11.2 (valid formula) *A formula A is (logically) valid if for every structure M and assignment h into M, we have $M, h \models A$.*
We write '$\models A$' (as above) if A is valid.

Definition 11.3 (satisfiable formula) *A formula A is satisfiable if for some structure M and assignment h into M, we have $M, h \models A$.*

Definition 11.4 (equivalent formulas)
Formulas A, B are logically equivalent if for every structure M and assignment h into M, we have $M, h \models A$ if and only if $M, h \models B$.

The links between these (page 43) also hold for predicate logic.
So (eg) the notions of valid/satisfiable formula, and equivalence, can all be expressed in terms of valid arguments.

Which arguments are valid?

Some examples of valid arguments:

- valid propositional ones: eg, $A \wedge B \models A$.
- many new ones: eg
  $\forall x(\mathtt{lecturer}(x) \to \mathtt{human}(x))$,
  $\exists x(\mathtt{lecturer}(x) \wedge \mathtt{bought}(x, \mathtt{Texel}))$
  $\models \exists x(\mathtt{human}(x) \wedge \mathtt{bought}(x, \mathtt{Texel}))$.

  'All lecturers are human, some lecturer bought Texel
  $\models$ some human bought Texel.'

Deciding if a supposed argument $A_1, \ldots, A_n \models B$ is valid is extremely hard in general.
We cannot just check that all L-structures + assignments that make $A_1, \ldots, A_n$ true also make B true (like truth tables).
This is because there are infinitely many L-structures.

Theorem 11.5 (Church, 1935) *No computer program can be written to identify precisely the valid arguments of predicate logic.*

Useful ways of validating arguments

In spite of theorem 11.5, we can often verify in practice that an argument or formula in predicate logic is valid. Ways to do it include:

- direct reasoning (the easiest, once you get used to it)
- equivalences (also useful)
- proof systems like natural deduction

The same methods work for showing a formula is valid. (A is valid if and only if $\models A$.)

Truth tables no longer work. You can't tabulate all structures — there are infinitely many.

11.1 Direct reasoning

Let's show
$\forall x(\mathtt{lecturer}(x) \to \mathtt{human}(x)), \quad \exists x(\mathtt{lecturer}(x) \wedge \mathtt{bought}(x, \mathtt{Texel}))$
$\models \exists x(\mathtt{human}(x) \wedge \mathtt{bought}(x, \mathtt{Texel}))$.

Take any L-structure M (where L is as before). Assume that
1) $M \models \forall x(\mathtt{lecturer}(x) \to \mathtt{human}(x))$ and
2) $M \models \exists x(\mathtt{lecturer}(x) \wedge \mathtt{bought}(x, \mathtt{Texel}))$.
Show $M \models \exists x(\mathtt{human}(x) \wedge \mathtt{bought}(x, \mathtt{Texel}))$.

So we need to find an a in M such that
$M \models \mathtt{human}(a) \wedge \mathtt{bought}(a, \mathtt{Texel})$.

By (2), there is a in M such that
$M \models \mathtt{lecturer}(a) \wedge \mathtt{bought}(a, \mathtt{Texel})$.
So $M \models \mathtt{lecturer}(a)$.
By (1), $M \models \mathtt{lecturer}(a) \to \mathtt{human}(a)$.
So $M \models \mathtt{human}(a)$.
So $M \models \mathtt{human}(a) \wedge \mathtt{bought}(a, \mathtt{Texel})$, as required.

Another example

Let's show
$\forall x(\mathtt{human}(x) \to \mathtt{lecturer}(x))$,
$\forall x(\mathtt{Sun}(x) \to \mathtt{lecturer}(x))$,
$\forall x(\mathtt{human}(x) \vee \mathtt{Sun}(x)) \qquad \models \forall x\, \mathtt{lecturer}(x)$.

Take any M such that
1) $M \models \forall x(\mathtt{human}(x) \to \mathtt{lecturer}(x))$,
2) $M \models \forall y(\mathtt{Sun}(y) \to \mathtt{lecturer}(y))$,
3) $M \models \forall z(\mathtt{human}(z) \vee \mathtt{Sun}(z))$.

Show $M \models \forall x\, \mathtt{lecturer}(x)$.

Take arbitrary a in M. We require $M \models \mathtt{lecturer}(a)$.

Well, by (3), $M \models \mathtt{human}(a) \vee \mathtt{Sun}(a)$.
If $M \models \mathtt{human}(a)$, then by (1), $M \models \mathtt{lecturer}(a)$.
Otherwise, $M \models \mathtt{Sun}(a)$. Then by (2), $M \models \mathtt{lecturer}(a)$.
So either way, $M \models \mathtt{lecturer}(a)$, as required.

Direct reasoning with equality

Let's show $\forall x \forall y(x = y \wedge \exists z R(x, z) \to \exists v R(y, v))$ is valid.

Take any structure M, and objects a, b in $dom(M)$. We need to show

$$M \models a = b \wedge \exists z R(a, z) \to \exists v R(b, v).$$

So we need to show that
IF $M \models a = b \wedge \exists z R(a, z)$ THEN $M \models \exists v R(b, v)$.

But IF $M \models a = b \wedge \exists z R(a, z)$, then a, b are the same object.
So $M \models \exists z R(b, z)$.

So there is an object c in $dom(M)$ such that $M \models R(b, c)$.

Therefore, $M \models \exists v R(b, v)$. We're done.

11.2 Equivalences

As well as the propositional equivalences seen before, we have extra ones for predicate logic. A, B denote arbitrary predicate formulas.

28. $\forall x \forall y A$ is logically equivalent to $\forall y \forall x A$.
29. $\exists x \exists y A$ is (logically) equivalent to $\exists y \exists x A$.
30. $\neg \forall x A$ is equivalent to $\exists x \neg A$.
31. $\neg \exists x A$ is equivalent to $\forall x \neg A$.
32. $\forall x(A \wedge B)$ is equivalent to $\forall x A \wedge \forall x B$.
33. $\exists x(A \vee B)$ is equivalent to $\exists x A \vee \exists x B$.

Equivalences involving bound variables

34. If x does not occur free in A, then $\forall x A$ and $\exists x A$ are equivalent to A.
35. If x doesn't occur free in A, then
    $\exists x(A \wedge B)$ is equivalent to $A \wedge \exists x B$, and
    $\forall x(A \vee B)$ is equivalent to $A \vee \forall x B$.
36. If x does not occur free in A then
    $\forall x(A \to B)$ is equivalent to $A \to \forall x B$.
37. *Note:* if x does not occur free in B then
    $\forall x(A \to B)$ is equivalent to $\exists x A \to B$.
38. (Renaming bound variables)
    If Q is $\forall$ or $\exists$, y is a variable that does not occur in A, and
    B is got from A by replacing all free occurrences of x in A by y,
    then QxA is equivalent to QyB.
    Eg $\forall x \exists y\ \texttt{bought}(x, y)$ is equivalent to $\forall z \exists v\ \texttt{bought}(z, v)$.

Equivalences/validities involving equality

39. For any term t, $t = t$ is valid.
40. For any terms t, u,
    $t = u$ is equivalent to $u = t$
41. (Leibniz principle) If A is a formula in which x occurs free, y doesn't occur in A at all, and B is got from A by replacing one or more free occurrences of x by y, then

    $$x = y \to (A \leftrightarrow B)$$

    is valid.

    Example: $x = y \to (\forall z R(x, z) \leftrightarrow \forall z R(y, z))$ is valid.

Examples using equivalences

These equivalences form a toolkit for transforming formulas.

Eg: let's show that if x is not free in A then $\forall x(\exists x \neg B \to \neg A)$ is equivalent to $\forall x(A \to B)$.

Well, the following formulas are equivalent:

- $\forall x(\exists x \neg B \to \neg A)$
- $\exists x \neg B \to \neg A$ (equivalence 34, since x is not free in $\exists x \neg B \to \neg A$)
- $\neg \forall x B \to \neg A$ (equivalence 30)
- $A \to \forall x B$ (example on p. 59)
- $\forall x(A \to B)$ (equivalence 36, since x is not free in A).

Warning: non-equivalences

Depending on A, B, the following need *NOT* be logically equivalent (though the first $\models$ the second):

- $\forall x(A \to B)$ and $\forall x A \to \forall x B$
- $\exists x(A \wedge B)$ and $\exists x A \wedge \exists x B$.
- $\forall x A \vee \forall x B$ and $\forall x(A \vee B)$.

Can you find a 'countermodel' for each one? (Find suitable A, B and a structure M such that $M \models$ 2nd but $M \not\models$ 1st.)

11.3 Natural deduction for predicate logic

This is quite easy to set up. We keep the old propositional rules — e.g., $A \vee \neg A$ for any first-order sentence A ('lemma') — and add new ones for $\forall, \exists, =$.

You construct natural deduction proofs as for propositional logic: first think of a direct argument, then convert to ND.

This is *even more important than for propositional logic.* There's quite an art to it.

Validating arguments by predicate ND can sometimes be harder than for propositional ones, because the new rules give you wide choices, and at first you may make the wrong ones! If you find this depressing, remember, it's a hard problem, there's no computer program to do it (theorem 11.5)!

$\exists$-introduction, or $\exists I$

To prove a sentence $\exists x A$, you have to prove $A(t)$, for some closed term t of your choice.

	$\vdots$	
1	$A(t)$	we got this somehow...
2	$\exists x A$	$\exists I(1)$

Notation 11.6 Here, and below, $A(t)$ is the sentence got from $A(x)$ by replacing all free occurrences of x by t.

Recall a *closed term* is one with no variables — it's made with only constants and function symbols.

This rule is reasonable. If in some structure, $A(t)$ is true, then so is $\exists x A$, because there exists an object in M (namely, the value in M of t) making A true.

But choosing the 'right' t can be hard — that's why it's such a good idea to think up a 'direct argument' first!

$\exists$-elimination, $\exists E$ (tricky!)

Let $A(x)$ be a formula. If you have managed to write down $\exists x A$, you can prove a sentence B from it by

- assuming $A(c)$, where c is a *new* constant not used in B or in the proof so far,
- proving B from this assumption.

During the proof, you can use anything already established.
But once you've proved B, you cannot use any part of the proof, *including* c, later on. *I mean it!* So we isolate the proof of B from $A(c)$, in a box:

1	$\exists x A$	got this somehow
2	$A(c)$	ass
	⟨the proof⟩	hard struggle
3	B	we made it!
4	B	$\exists E(1, 2, 3)$

c is often called a Skolem constant.

Justification of $\exists E$

Basically, 'we can give any object a name'.

If $\exists x A$ is true in some structure M, then there is an object a in $dom(M)$ such that $M \models A(a)$.

Now a may not be named by a constant in M. But we can add a new constant to name it — say, c — and add the information to M that c names a.

c must be new — the other constants already in use may not name a in M.

So $A(c)$ *for new c* is really no better or worse than $\exists x A$. If we can prove B from the assumption $A(c)$, it counts as a proof of B from the already-proved $\exists x A$.

Example of $\exists$-rules

Show $\exists x(P(x) \wedge Q(x)) \vdash \exists x P(x) \wedge \exists x Q(x)$.

1	$\exists x(P(x) \wedge Q(x))$	given
2	$P(c) \wedge Q(c)$	ass
3	$P(c)$	$\wedge E(2)$
4	$\exists x P(x)$	$\exists I(3)$
5	$Q(c)$	$\wedge E(2)$
6	$\exists x Q(x)$	$\exists I(5)$
7	$\exists x P(x) \wedge \exists x Q(x)$	$\wedge I(4,6)$
8	$\exists x P(x) \wedge \exists x Q(x)$	$\exists E(1,2,7)$

In English: Assume $\exists x(P(x) \wedge Q(x))$. Then there is a with $P(a) \wedge Q(a)$.
So $P(a)$ and $Q(a)$. So $\exists x P(x)$ and $\exists x Q(x)$.
So $\exists x P(x) \wedge \exists x Q(x)$, as required.

Note: only sentences occur in ND proofs. They should never involve formulas with free variables!

$\forall$-introduction, $\forall I$

To introduce the sentence $\forall x A$, for some $A(x)$, you introduce a *new* constant, say c, not used in the proof so far, and prove $A(c)$.
During the proof, you can use anything already established.
But once you've proved $A(c)$, you can no longer use the constant c later on.
So isolate the proof of $A(c)$, in a box:

1	c	$\forall I$ const
	$\langle$the proof$\rangle$	hard struggle
2	$A(c)$	we made it!
3	$\forall x A$	$\forall I(1,2)$

This is the *only* time in ND that you write a line (1) containing a *term,* not a formula. And it's the *only* time a box doesn't start with a line labelled 'ass'.

Justification

To show $M \models \forall x A$, we must show $M \models A(a)$ for every object a in $dom(M)$.

So choose an arbitrary a, add a new constant c naming a, and prove $A(c)$. As a is arbitrary, this shows $\forall x A$.

c must be new, because the constants already in use may not name this particular a.

$\forall$-elimination, or $\forall E$

Let $A(x)$ be a formula. If you have managed to write down $\forall x A$, you can go on to write down $A(t)$ for any closed term t. (It's your choice which t!)

	$\vdots$	
1	$\forall x A$	we got this somehow...
2	$A(t)$	$\forall E(1)$

This is easily justified: if $\forall x A$ is true in a structure, then certainly $A(t)$ is true, for any closed term t.

Choosing the 'right' t can be hard — that's why it's such a good idea to think up a 'direct argument' first!

Example of $\forall$-rules

Let's show $P \to \forall x Q(x) \vdash \forall x(P \to Q(x))$.
Here, P is a 0-ary relation symbol — that is, a propositional atom.

1	$P \to \forall x Q(x)$	given
2	c	$\forall I$ const
3	P	ass
4	$\forall x Q(x)$	$\to E(3,1)$
5	$Q(c)$	$\forall E(4)$
6	$P \to Q(c)$	$\to I(3,5)$
7	$\forall x(P \to Q(x))$	$\forall I(2,6)$

In English: Assume $P \to \forall x Q(x)$. Then for any object a, if P then $\forall x Q(x)$, so $Q(a)$.

So for any object a, if P, then $Q(a)$.

That is, for any object a, we have $P \to Q(a)$. So $\forall x(P \to Q(x))$.

Example with all the quantifier rules

Show $\exists x \forall y G(x,y) \vdash \forall y \exists x G(x,y)$.

1	$\exists x \forall y G(x,y)$	given
2	d	$\forall I$ const
3	$\forall y G(c,y)$	ass
4	$G(c,d)$	$\forall E(3)$
5	$\exists x G(x,d)$	$\exists I(4)$
6	$\exists x G(x,d)$	$\exists E(1,3,5)$
7	$\forall y \exists x G(x,y)$	$\forall I(2,6)$

English: Assume $\exists x \forall y G(x,y)$. Then there is some object c such that $\forall y G(c,y)$.

So for any object d, we have $G(c,d)$, so certainly $\exists x G(x,d)$.

Since d was arbitrary, we have $\forall y \exists x G(x,y)$.

Derived rule $\forall\!\to\! E$

This is like PC: it collapses two steps into one. Useful, but not essential.

Idea: often we have proved $\forall x(A(x) \to B(x))$ and $A(t)$, for some formulas $A(x), B(x)$ and some closed term t.

We know we can derive $B(t)$ from this:

1	$\forall x(A(x) \to B(x))$	(got this somehow)
2	$A(t)$	(this too)
3	$A(t) \to B(t)$	$\forall E(1)$
4	$B(t)$	$\to E(2,3)$

So let's just do it in 1 step:

1	$\forall x(A(x) \to B(x))$	(got this somehow)
2	$A(t)$	(this too)
3	$B(t)$	$\forall\!\to\! E(2,1)$

Example of $\forall\!\to\! E$ in action

Show $\forall x \forall y (P(x,y) \to Q(x,y)), \quad \exists x P(x,a) \vdash \exists y Q(y,a)$.

1	$\forall x \forall y (P(x,y) \to Q(x,y))$	given
2	$\exists x P(x,a)$	given
3	$P(c,a)$	ass
4	$Q(c,a)$	$\forall\!\to\! E(3,1)$
5	$\exists y Q(y,a)$	$\exists I(4)$
6	$\exists y Q(y,a)$	$\exists E(2,3,5)$

We used $\forall\!\to\! E$ on 2 $\forall$s at once. This is even more useful.

Rules for equality

- Reflexivity of equality (refl).
  Whenever you feel like it, you can introduce the sentence $t = t$, for any closed L-term t and for any L you like.

	$\vdots$	bla bla bla
1	$t = t$	refl

  (Idea: any L-structure makes $t = t$ true, so this is sound.)

More rules for equality

- Substitution of equal terms (=sub).
  If $A(x)$ is a formula, t, u are closed terms, you've proved $A(t)$, and you've also proved either $t = u$ or $u = t$, you can go on to write down $A(u)$.

1	$A(t)$	got this somehow...
2	$\vdots$	yatter yatter yatter
3	$t = u$	...and this
4	$A(u)$	=sub$(1,3)$

  (Idea: if t, u are equal, there's no harm in replacing t by u as the value of x in A.)

Examples with equality...

Show $c = d \vdash d = c$. (c, d are constants.)

1	$c = d$	given
2	$d = d$	refl
3	$d = c$	=sub$(2,1)$

This is often useful, so make it a derived rule:

1	$c = d$	given
2	$d = c$	=sym(1)

More examples with equality. . .

Show $\vdash \forall x \exists y(y = f(x))$.

1	c	$\forall I$ const
2	$f(c) = f(c)$	refl
3	$\exists y(y = f(c))$	$\exists I(2)$
4	$\forall x \exists y(y = f(x))$	$\forall I(1, 3)$

English: For any object c, we have $f(c) = f(c)$ — $f(c)$ is the same as itself.

So for any c, there is something equal to $f(c)$ — namely, $f(c)$ itself!

So for any c, we have $\exists y(y = f(c))$.

Since c was arbitrary, we get $\forall x \exists y(y = f(x))$.

Harder example

Show $\exists x \forall y(P(y) \to y = x), \quad \forall x P(f(x)) \vdash \exists x(x = f(x))$.

1	$\exists x \forall y(P(y) \to y = x)$	given
2	$\forall x P(f(x))$	given
3	$\forall y(P(y) \to y = c)$	ass
4	$P(f(c))$	$\forall E(2)$
5	$f(c) = c$	$\forall{\to}E(4, 3)$
6	$c = f(c)$	$=$sym(5)
7	$\exists x(x = f(x))$	$\exists I(6)$
8	$\exists x(x = f(x))$	$\exists E(1, 3, 7)$

English: assume there is an object c such that all objects a satisfying P (if any) are equal to c, and for *any* object b, $f(b)$ satisfies P.

Taking 'b' to be c, $f(c)$ satisfies P, so $f(c)$ is equal to c.

So c is equal to $f(c)$.

As $c = f(c)$, we obviously get $\exists x(x = f(x))$.

Final remarks

Now you've done sets, relations, and functions in other courses(?), here's what an L-structure M really is.

It consists of the following items:

- a non-empty set, $dom(M)$
- for each constant $c \in L$, an element $c^M \in dom(M)$
- for each n-ary function symbol $f \in L$, an n-ary function $f^M : dom(M)^n \to dom(M)$
- for each n-ary relation symbol $R \in L$, an n-ary relation R^M on $dom(M)$ — that is, $R^M \subseteq dom(M)^n$.

Recall for a set S, S^n is $\overbrace{S \times S \times \cdots \times S}^{n \text{ times}}$.

Another name for a relation (symbol) is a *predicate (symbol).*

What we did (all can be in Xmas test!)

Propositional logic

- Syntax
  Literals, clauses (see Prolog next term!)
- Semantics
- English–logic translations
- Arguments, validity
  - †truth tables
  - direct reasoning
  - equivalences, †normal forms
  - natural deduction

Classical first-order predicate logic

same again (except †), plus

- Many-sorted logic
- Specifications, pre- and post-conditions (continued in Reasoning about Programs)

Some of what we didn't do...

- normal forms for first-order logic
- proof of soundness or completeness for natural deduction
- theories, compactness, non-standard models, interpolation
- Gödel's theorem
- non-classical logics, eg. intuitionisitic logic, linear logic, modal & temporal logic
- finite structures and computational complexity
- automated theorem proving

Do the 2nd and 4th years for some of these.

Modern logic at research level

- Advanced computing uses classical, modal, temporal, and dynamic logics. Applications in AI, to specify and verify chips, in databases, concurrent and distributed systems, multi-agent systems, protocols, knowledge representation, ... Theoretical computing (complexity, finite model theory) need logic.
- In mathematics, logic is studied in *set theory, model theory,* including *non-standard analysis,* and *recursion theory.* Each of these is an entire field, with dozens or hundreds of research workers.
- In philosophy, logic is studied for its contribution to formalising truth, validity, argument, in many settings: eg, involving time, or other possible worlds.
- Logic provides the foundation for several modern theories in linguistics. This is nowadays relevant to computing.