CO140 - Logic

Prelude

The content discussed here is part of CO140 - Logic (Computing MEng); taught by Alessandra Russo, and Ian Hodkinson, in Imperial College London during the academic year 2018/19. The notes are written for my personal use, and have no guarantee of being correct (although I hope it is, for my own sake).

Material Order

Seeing as this module isn't on Panopto, these notes are solely based off of the provided notes on CATe. This is the order in which they are uploaded (and I'd assume the order in which they are taught).

1. *Propositional Logic - Syntax.pdf*
2. *Propositional Logic - Semantics.pdf*
3. *Propositional Logic - English Correspondence.pdf*
4. *Propositional Logic - Arguments and Validity.pdf*
5. *PropositionalLogic - CheckValidity.pdf*
6. *Propositional Logic - Natural Deduction Part1.pdf*
7. *Propositional Logic - Natural Deduction Part 2.pdf*
8. *Propositional Logic- Natural Deduction Part 3.pdf*
9. *Propositional Logic - Lemmas.pdf*
10. *First-order logic.pdf*

Introduction

A logic system consists of 3 things:

1. Syntax - formal language used to express concepts
2. Semantics - meaning for the syntax
3. Proof theory - syntactic way of identifying valid statements of language

Considering the basic example in a program, we can then see the features;

```
if count > 0 and not found then
    decrement count;
    look for next entry;
end if
```

1. basic (**atomic**) statements (**propositions**) are either $\top$ or $\bot$ depending on circumstance;
    i. `count > 0`
    ii. `found`
2. **boolean operations**, such as `and`, `or`, `not`, etc. are used to build complex statements from **atomic propositions**
3. the final statement `count > 0 and not found` evaluates to either $\top$ or $\bot$

Syntax

The formal language of logic consists of three ingredients;

1. Propositional atoms (propositional variables), evaluate to a truth value of either $\top$ or $\bot$. These are represented with letters; $p, p', p_0, p_1, p_2, p_n, q, r, s, ...$
2. Boolean connectives;

• `and` is written as $p \wedge q$	p and q both hold
• `or` is written as $p \vee q$	p or q holds (or both)
• `not` is written as $\neg p$	p does not hold
• `if-then / implies` is written as $p \to q$	if p holds, then so does q
• `if-and-only-if` is written as $p \leftrightarrow q$	p holds if and only if q holds
• `truth`, and `falsity` are written as $\top$, and $\bot$ respectively.	logical constants

3. Punctuation. Similar to arithmetic, the lack of brackets can make an expression ambiguous. For example, $p_0 \vee p_1 \wedge p_2$ can be read as either $(p_0 \vee p_1) \wedge p_2$ or $p_0 \vee (p_1 \wedge p_2)$, which are different. The latter is the correct interpretation due to binding conventions.

   We can order the boolean connectives by decreasing binding strength;

   (strongest) $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$ (weakest)

   While repeated disjunctions ($\vee$), and conjunctions ($\wedge$) are fine, as $p \wedge q \wedge r$ is equivalent to $p \wedge (q \wedge r)$, and the same for $\vee$, due to associativity, the same isn't true for $\to$. Due to the ambiguity, brackets should always be used.

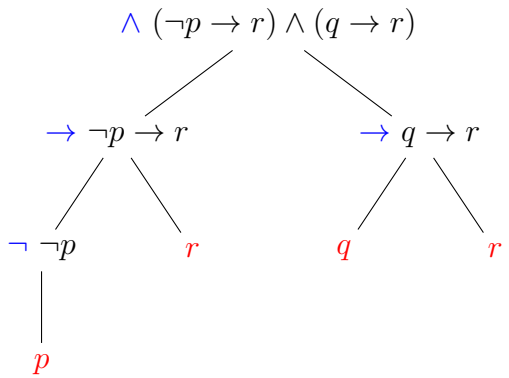
   There are also exceptions to the rule, for example with $p \to r \wedge q \to r$ - this should be $p \to (r \wedge q) \to r$ according to our binding conventions, but brackets should be used to ensure the correct interpretation.

Formulas

Something is a **well-formed formula** only if it is built from the following rules (the brackets are required);

1. a propositional atom $(p, p', p_0, p_1, p_2, p_n, q, r, s, ...)$ is a propositional formula
2. $\top$, and $\bot$ are both formulas
3. if A is a formula, then $(\neg A)$ is also a formula
4. if A, and B are both formulas, then $(A \wedge B), (A \vee B), (A \to B), (A \leftrightarrow B)$ are also formulas

We can also create a tree to parse a logical formula, for example; $(\neg p \to r) \wedge (q \to r)$



Note that this tree shows the principal connective in blue, and the propositional atoms in red. Note that $\wedge$ is the principal connective in the top layer, and it therefore has the general form $A \wedge B$, and so on going down.

Definitions

- A formula is a **negated formula** when it is in the form $\neg A$, negated atoms are sometimes called **negated-atomic**.
- $A \wedge B$, and $A \vee B$ are **conjunctions**, and **disjunctions**. A, and B, are **conjuncts**, and **disjuncts**, respectively.
- $A \to B$ is an implication. A is the **antecedent**, and B is the **consequent**

Semantics

The connectives covered above have a rough English translation. However a natural language has ambiguity, and as engineers, we need precise meanings for formulas. This is the truth table for every connective that will be used in this course (?):

p	q	$\top$	$\bot$	$p \wedge q$	$p \vee q$	$\neg p$	$p \to q$	$p \leftrightarrow q$	$p \uparrow q$
0	0	1	0	0	0	1	1	1	1
0	1	1	0	0	1	1	1	0	1
1	0	1	0	0	1	0	0	0	1
1	1	1	0	1	1	0	1	1	0

Note how we can also define new connectives (see how $A \uparrow B$ was defined in the last column); this is a NAND connective - equivalent to $\neg(A \wedge B)$.

Translation

English to Logic

- **but** means `and`

  "I will go out, but it is raining" `(i will go out)` $\wedge$ `(it is raining)`

- **unless** generally means `or`

  "I will go out unless it rains" `(i will go out)` $\vee$ `(it will rain)` (note the `will`)

  $\neg$`(it will rain)` $\to$ `i will go out`

  There is also the strong form of **unless**, but in we generally use the weak form in computing

  `(i will go out)` $\leftrightarrow \neg$`(it will rain)`

- **or** generally refers to exclusive or (strong reading) in English, but it can also refer to inclusive or (weak reading). However, we always take the weak reading in computing.

Modality

`I don't know what this means, so I'm just ignoring it for now`

Logic to English

While the others are slightly more straightforward, $\to$ is a pain to translate.

For example, `(i am the pope)` $\to$ `(i am an atheist)` evaluates to true, as falsity implies anything, however if we were to translate it into English, "If I am the Pope, then I am an atheist" is (most likely) untrue.

Another example is the following; $p \wedge q \to r$, and $(p \to r) \vee (q \to r)$ are logically equivalent, but can be translated into different meanings. For example, let p be "event A happens", let q be "event B happens", and q be "event C happens". The former can be translated to "If both A and B happens, then C happens", whereas the latter becomes "If A happens, then C happens, or if B happens, then C also happens".

Arguments

We use the double turnstile, $\vDash$ (`\vDash` in LaTeX), to mean **therefore**. For example, the *Socrates syllogism* can be expressed as `(socrates is a man)`, `(men are mortal)` $\vDash$ `(socrates is mortal)` in logic, and in English as;

- Socrates is a man
- Men are mortal
- Therefore, Socrates is mortal

The definition of a valid argument is as follows;
Given valid formulas $A_1, A_2, ..., A_n, B$, and '$A_1, ..., A_n$ therefore B', we can write it as $A_1, ..., A_n \vDash B$, if, and only if B is true in every situation where $A_1, ..., A_n$ are all true.

Examples

- $A, A \to B \vDash B$ **modus ponens**
- $A \to B, \neg B \vDash \neg A$ **modus tollens**
- $A \to B, B \nvDash A$ A can be false, as falsity implies anything

Definitions

- A propositional formula is logically **valid** if it's true in all situations ($\vDash A$), if A is **valid**
- A propositional formula is **satisfiable** if it's true in at least one situation (hence **valid** $\to$ **satisfiable**)
- Two propositonal formulas are logically **equivalent** if they are true in the same situations.

argument	validity	satisfiability	equivalence
$A \vDash B$	$A \to B$ valid	$A \wedge \neg B$ unsatisfiable	$(A \to B) \equiv \top$
$\top \vDash A$	A valid	$\neg A$ unsatisfiable	$A \equiv \top$
$A \nvDash \bot$	$\neg A$ not valid	A satisfiable	
$A \vDash B$, and $B \vDash A$		$A \leftrightarrow \neg B$ unsatisfiable	$A \equiv B$

(copied directly from *Propositional Logic - Arguments and Validity.pdf*)

Validity in Propositional Logic

The main ways used to check validity are as follows;

- Truth tables - check all possible situations, and check the results of each formula are $\top$
- Direct argument
- Equivalences - using equivalences to reduce the initial formula to $\top$
- Various proof systems - including Natural Deduction

In general, if we want to show that A is logically equivalent to B, we need to show $A \leftrightarrow B$ is **valid**.

Truth Tables

The use of truth tables to prove validity is fairly self-explanatory; as we're testing each situation, it's the easiest method (and it works for propositional logic since we have a finite number of configurations - doesn't work for first-order), however it's inelegant, and quite tedious depending on the number of propositional atoms.

For example, if we were to prove $(p \to q) \leftrightarrow (\neg p \vee q)$ is valid, we have to evaluate all of the subformulas.

p	q	$p \to q$	$\neg p$	$\neg p \vee q$	$(p \to q) \leftrightarrow (\neg p \vee q)$
0	0	1	1	1	1
0	1	1	1	1	1
1	0	0	0	0	1
1	1	1	0	1	1

(copied directly from *Propositional Logic - CheckValidity.pdf*)

As the propositional formula has 1 in all four possible configurations of p, and q, we can then say it is valid, and as such, $p \to q$ is logically equivalent to $\neg p \vee q$

Direct Argument

We can show the validity of $((p \to q) \to p) \to p$ (known as *Peirce's law*) with direct argument.

We can take an argument by cases, either p is $\top$ or p is $\bot$.

- $p \leftrightarrow \top$ - we know this is true as $A \to B$ is $\top$ whenever B is $\top$
- $p \leftrightarrow \bot$ - we have $p \to q$ evaluating to $\top$, as $A \to B$ is $\top$ whenever A is $\bot$. As such, this formula is evaluated to $(\top \to p) \to q$. However, we know that p is $\bot$, hence we have $\top \to \bot$, which we know evaluates to $\bot$ by the truth table for $\to$. As such, we have $\bot \to p$, hence it follows that it is valid, seeing as $A \to B$ is $\top$ whenever A is $\bot$.
- This is an argument by cases, known as **law of excluded middle** (you will use this often in Natural Deduction).

Equivalences

Refer to *Logic cribsheet.pdf* for a full list of equivalences

1. $A \wedge B \equiv B \wedge A$ commutativity of $\wedge$
2. $A \wedge A \equiv A$ idempotence of $\wedge$
3. $A \wedge \top \equiv A$
4. $\bot \wedge A, \neg A \wedge A \equiv \bot$
5. $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$ associativity of $\wedge$
6. $A \vee B \equiv B \wedge A$ commutativity of $\vee$
7. $A \vee A \equiv A$ idempotence of $\vee$
8. $\bot \vee A \equiv A$
9. $\top \vee A, \neg A \vee A \equiv \top$
10. $(A \vee B) \vee C \equiv A \vee (B \vee C)$ associativity of $\vee$
11. $\neg \top \equiv \bot$
12. $\neg \bot \equiv \top$
13. $\neg \neg A \equiv A$
14. $A \to A \equiv \top$

15. $\top \to A \equiv A$
16. $A \to \top \equiv \top$
17. $\bot \to A \equiv \top$
18. $A \to \bot \equiv \neg A$
19. $A \to B \equiv \neg A \lor B$
20. $A \leftrightarrow B \equiv (A \to B) \land (B \to A) \equiv (A \land B) \lor (\neg A \land \neg B) \equiv \neg A \leftrightarrow \neg B$
21. $\neg(A \leftrightarrow B) \equiv \neg A \leftrightarrow B \equiv ...$ the rest can be derived from the above
22. $\neg(A \land B) \equiv \neg A \lor \neg B$ de Morgan laws
23. $\neg(A \lor B) \equiv \neg A \land \neg B$ de Morgan laws
24. $A \land (B \lor C) \equiv (A \land B) \lor (A \land C)$
25. $A \lor (B \land C) \equiv (A \lor B) \land (A \lor C)$
26. $A \lor (A \land B) \equiv A \lor (A \land B) \equiv A$

Normal Forms

- A formula is in **disjunctive** NF (**DNF** - $\lor$) if it's a disjunction of conjunctions of literals
- A formula is in **conjunctive** NF (**CNF** - $\land$) if it's a conjunction of disjunctions of literals (a conjunction of clauses)

Rewriting

1. Get rid of $\to$, and $\leftrightarrow$

   Replace $A \to B$ with $\neg A \lor B$

   Replace $A \leftrightarrow B$ with $(A \land B) \lor (\neg A \land \neg B)$
2. Use de Morgan laws to push negations down to the atoms
3. Delete double negations (replace $\neg\neg A$ with A)
4. Rearrange with distributivity equivalences to the desired form
5. Use the equivalences which reduce two atoms to one ($\bot \lor A \equiv A$ etc.) until no further progress can be made

Example

Write $p \land q \to \neg(p \leftrightarrow \neg r)$ in DNF

1. $p \land q \to \neg(p \leftrightarrow \neg r)$
2. $\neg(p \land q) \lor \neg(p \leftrightarrow \neg r)$ remove $\to$
3. $\neg(p \land q) \lor \neg((p \land \neg r) \lor (\neg p \land r))$ remove $\leftrightarrow$
4. $\neg p \lor \neg q \lor \neg(p \land \neg r) \land \neg(\neg p \land r)$ de Morgan
5. $\neg p \lor \neg q \lor (\neg p \lor r) \land (p \lor \neg r)$ de Morgan
6. $\neg p \lor \neg q \lor ((\neg p \lor r) \land p) \lor ((\neg p \lor r) \land \neg r)$ distributivity of $\land$
7. $\neg p \lor \neg q \lor (\neg p \land p) \lor (r \land p) \lor (\neg p \land \neg r) \lor (r \land \neg r)$ distributivity of $\land$
8. $\neg p \lor \neg q \lor (r \land p) \lor (\neg p \land \neg r)$ distributivity of $\land$
9. $\neg p \lor \neg q \lor (r \land p)$ $A \lor (A \land B) \equiv A$

   While this is in DNF, we can leave it, and simplify further
10. $\neg q \lor ((r \lor \neg p) \land (p \lor \neg p))$ distributivity of $\lor$
11. $\neg q \lor r \lor \neg p$ $A \land (B \lor \neg B) \equiv A$ (combination of equivalences)

Natural Deduction

`Read Jordan Spooner's notes for this; just practice it on Pandora until you feel confident. These are just some key points from the slides, examples are excluded, as it's better to just do questions`

- If we prove A, and B, we get $A \wedge B$ ($\wedge$I)
- If we have $A \wedge B$, we get both A, and B ($\wedge$E)
- We often need to make a temporary assumption, for example, if we assume A, and in that box, we get B, then it follows that $A \to B$ ($\to$I)

  nothing in the box can be used outside of it; consider it as a scope
- If we have $A \to B$, and A, we get B ($\to$E)
- If we have A, we get $A \vee B$ ($\vee$I)

  it doesn't matter what B is, it can literally be $\bot$
- If we have $A \vee B$, and we can prove $A \to C$, and $B \to C$ (see in the *Translation* section for a similar example to $(p \vee q) \to r \equiv (p \to r) \wedge (q \to r)$), we get C ($\vee$E).
- Note that $\vdash$, and $\models$, are different. The former is syntactic, and involves proofs, whereas the latter is semantic, and involves situations
- If assuming A leads to $\bot$, we get $\neg A$ ($\neg$I)
- If assuming $\neg A$ leads to $\bot$, we get A ($\neg$E)
- If we have $\neg\neg A$, we get A ($\neg\neg$)
- If we have both A, and $\neg A$, we get $\bot$ ($\bot$I)
- If we have $\bot$, we get A ($\bot$E)

  A can be anything, as we can prove anything from $\bot$
- If we prove $A \to B$, and $B \to A$, we get $A \leftrightarrow B$ ($\leftrightarrow$I)
- If we prove $A \leftrightarrow B$, and A (or B), we get B (or A), respectively ($\leftrightarrow$E)
- PC is a derived rule from $\neg$I, and $\neg\neg$

Definitions

- If a formula can be proved by a given proof system, it's a **theorem** (hence a **theorem** is any formula A where $\vdash A$)
- A system is **sound** if every theorem is valid, and **complete** if every valid formula is a theorem
- A formula is **consistent** if $\nvdash \neg A$
- A formula is consistent if, and only if it is satisfiable

First-order Predicate Logic

While we can use direct argument, equivalences, and natural deduction, truth tables can no longer be used, since we're working on an infinite set of possible situations.

Limitations of Propositional Logic

- the list is ordered
- every worker has a boss
- there is someone worse off than you
- it also can't express some of de Morgan's arguments, for example;

i. a horse is an animal

ii. therefore, the head of a horse is the head of an animal

Splitting the Atom

Previously, we considered phrases such as `the left lift is falling`, and `Adam gets in the left lift`, as atomic, without any internal structure. However, we can then regard `falling` as a **property**, or an **attribute**.

- a **unary** relation symbol takes one argument, hence it has an **arity** of 1.

  e.g. `falling(LeftLift)`
- a **binary** relation symbol takes two arguments, hence it has an arity of 2.

  e.g. `gets_in(Adam, LeftLift)`
- **constants**, which can name individual objects

  e.g. `LeftLift`, or `Adam`

While `gets_in(Adam, LeftLift)` doesn't seem that different from the original propositional atom `Adam gets in the left lift`, predicate logic is able to vary the arguments passed into the relation `gets_in`. The machinery used in predicate logic is **quantifiers**. In first-order logic, we have two quantifiers;

- $\forall$ - 'for all' e.g $\forall x(A)$, means that the predicate A applies to all x.
- $\exists$ - 'exists' (or 'some') e.g. $\exists x(A)$, means that the predicate A applies to at least one x.

Expressions like `LeftLift`, or `Adam` are constants, but to express more complex statements we need stuff like;

- $\exists x(\texttt{falling}(x) \ \wedge \ \texttt{gets\_in}(\texttt{Adam}, x))$ `Adam` gets into a falling lift

  Some x is a falling lift, and `Adam` gets in x
- $\exists x(\texttt{falling}(x))$ There exists an x that is a falling lift
- $\forall x(\texttt{falling}(x))$ Everything is a falling lift

Signatures

A **signature** is a set of constants, and relation symbols with specific arities. Also known as **similarity type**, **vocabulary**, or **language** (loosely)

This replaces the collection of propositional atoms we previously used in propositional logic. Usually L denotes a signature, $c, d, ...$ for constants, and $P, Q, R, S, ...$ for relation symbols.

Let us define an example signature L, consisting of the following;

- constants
  - `Adam`
  - `Ben`
  - `Charlie`
  - `Apple`
  - `Orange`
  - `Kale`
  - `Phone`
- unary relations (arity 1)

  - `fruit`
  - `human`
  - `student`

- binary relations (arity 2)
  - `ate`

Everything listed in L are just symbols, hence they don't come with any meaning. We will need to add a **situation**.

Terms

In order to write formulas, we need **terms** to name objects, they are not true or false, since they themselves are not formulas.

With a fixed signature L, any constant in L is an L-term, as well as any variable. Nothing else is an L-term.

A **closed** (or **ground**) term doesn't involve a variable. Hence constants are **ground** terms, and variables are not.

Formulas

Once again, with a fixed signature L, we can say the following are formulas, and nothing else is;

- given an n-ary relation R in L, and a set of L-terms $(t_1, t_2, ..., t_n)$, then $R(t_1, t_2, ..., t_n)$ is an atomic L-formula
- if t, and t' are L-terms, then $t = t'$ is an atomic L-formula (equality)
- $\top$, and $\bot$, are atomic L-formulas
- if A, and B are L-formulas, then so are $(\neg A)$, $(A \wedge B)$, $(A \vee B)$, $(A \to B)$, and $(A \leftrightarrow B)$
- if A is an L-formula, then $\forall x(A)$, and $\exists x(A)$ are L-formulas

Note that the binding conventions are the same as propositional logic, with the additional fact that $\forall x$, and $\exists x$ have the same binding strength as $\neg$

Now that we have these definitions, we can begin to construct examples of first-order logical formulas;

- $\texttt{ate}(\texttt{Adam}, x)$ — Adam ate x
- $\exists x(\texttt{ate}(\texttt{Adam}, x))$ — Adam ate something
- $\forall x(\texttt{student}(x) \to \texttt{human}(x))$ — all students are human (important)
- $\forall x(\texttt{ate}(\texttt{Adam}, x) \to \texttt{fruit}(x))$ — Adam only ate fruits / Everything Adam ate is a fruit
- $\forall x \exists y(\texttt{ate}(x, y))$ — eveyone ate something
- $\exists y \forall x(\texttt{ate}(x, y))$ — there is something that everyone ate
- $\exists x \forall y(\texttt{ate}(x, y))$ — someone ate everything

Note the subtle differences in the latter three examples, and how they have a rather drastic impact on the meaning of the formula.

Semantics

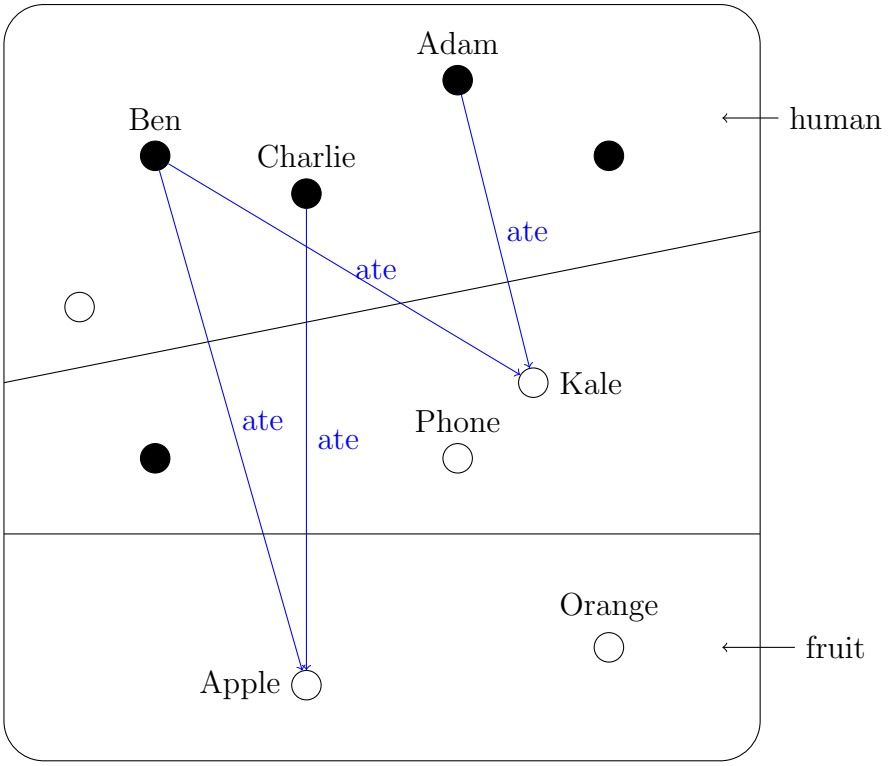
Like in propositional logic, we have to specify a **situation** for predicate logic, and how to evaluate predicate logic formulas in a specific situation.

With a given signature L, we have an L-structure (or **model**) M, which identifies an non-empty set of objects that M covers. It is defined as the **domain**, or **universe** of M, also known as $\text{dom}(M)$. M also specifies the meanings of the symbols in L, in terms of the objects in $\text{dom}(M)$.

An **object** in $\text{dom}(M)$ is the interpretation in M of a constant, and **relation** on $\text{dom}(M)$ is the interpretation in M of a relation symbol.

Using the our previously defined symbols, we can define a model M on L, which must state which objects are in $\text{dom}(M)$, which objects are the constants (`Adam`, `Ben`, ...), which objects are `human`, `student`, `fruit`, and which objects `ate` which.

- the labelled nodes represent the constants in L
- the interpretations / meanings of `fruit`, and `human` are drawn as regions (the arrows)
- the interpretation of `student` are the black nodes
- the interpretation of the binary relation `ate` is shown by the arrow between two nodes



NOTATION: given an L-structure M, and a constant c in L, we use the notation c^M to denote the interpretation of c in M. c is an object in $\text{dom}(M)$ that c names in M. Therefore the black node in the model, is $\texttt{Adam}^M$, which is not to be confused with `Adam`. The meaning of a constant c is the object c^M, which is assigned by the L-structure M, hence a constant can have multiple meanings since each L-structure assigns c a new meaning.

While our model is quite simple, it illustrates the basic requirements for an L-structure; it has the collection of objects ($\text{dom}(M)$), marks the constants, marks which objects satisfy the unary relations, as well as directed arrows showing which pairs of objects satisfy the binary relations. Generally, there isn't any easy way to represent n-ary relations (where $n \geq 3$). 0-ary (nullary) relations are propositional atoms.

The structure M tells us that `student(Ben)` is true, as the node representing $\texttt{Ben}^M$ is coloured black; therefore this can be written as $M \vDash$ `student(Ben)` - or M says `student(Ben)`. M also tells us that

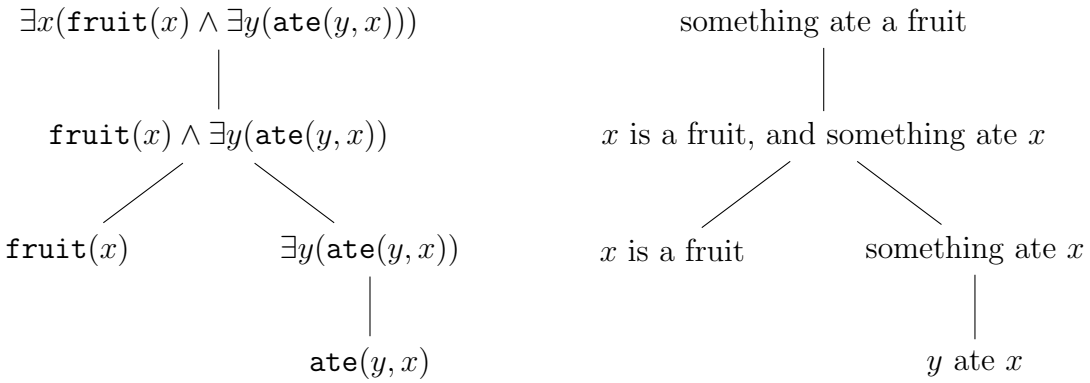
$\texttt{ate}(\texttt{Ben}, \texttt{Orange})$ is false, hence we can write $M \not\models \texttt{ate}(\texttt{Ben}, \texttt{Orange})$. M also states $\texttt{Kale}^M$ is not $\texttt{human}$, therefore we are able to write $M \models \neg\texttt{human}(\texttt{Kale})$.

This is a different use of $\models$ from the start of the module.

`There should be a section here about another model on the same signature, but that takes too much effort to draw.`

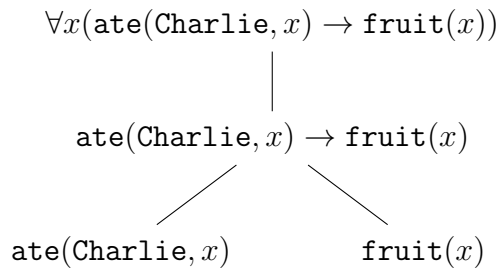
TIP: if you have something asking $M \models \forall x(R(x, ...) \rightarrow B)$?, the idea is to restrict the $\forall x$. We know that falsity implies anything from propositional logic, and therefore we only need to consider the cases in which $R(x, ...)$ is true. For example, on M, if we were asked $M \models \forall x(\texttt{ate}(\texttt{Adam}, x) \rightarrow \texttt{ate}(\texttt{Ben}, x))$?, instead of evaluating all the objects in $\text{dom}(M)$, we should only consider the ones in which $\texttt{ate}(\texttt{Adam}, x)$ evaluates to true, which would be just the object $\texttt{Kale}^M$. And as $\texttt{ate}(\texttt{Ben}, \texttt{Kale}^M)$ is true, the statement is valid.

TIP: for a fairly complex formula (we'll work with $\exists x(\texttt{fruit}(x) \wedge \exists y(\texttt{ate}(y, x)))$), a simple method is to work out what each subformula says in English (working upwards from the atomic subformulas). For example;



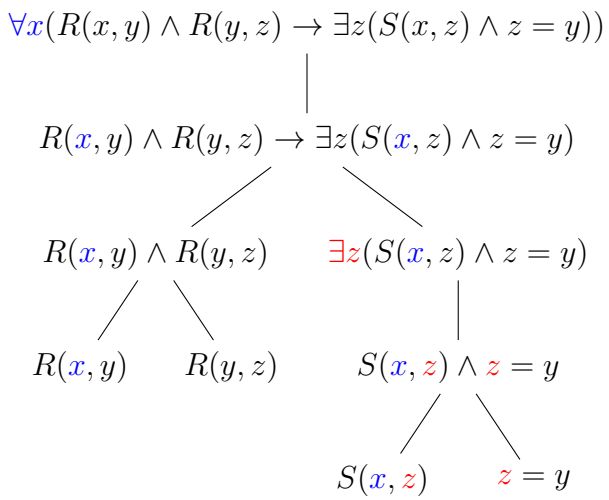
Truth in a Structure (Formal)

Once again, natural language is only a rough guide, and as engineers we need a more rigorous system. While we are able to work out the truth value of a complex formula by evaluating propositional atoms from the root of a formation tree in propositional logic, it's not as simple in predicate logic. For example, if we tried to evaluate $\forall x(\texttt{ate}(\texttt{Charlie}, x) \rightarrow \texttt{fruit}(x))$ with a formation tree, we'd quickly run into trouble;



What are the truth values for the leaf nodes? We don't know, since formulas of predicate logic doesn't have to be true or false in a given structure.

With a given formula A, a variable x in an atomic subformula of A is **bound** if it's under a quantifiers in the formation tree. Otherwise, the variable is **free**. For example (copied directly from *First-order logic.pdf*);

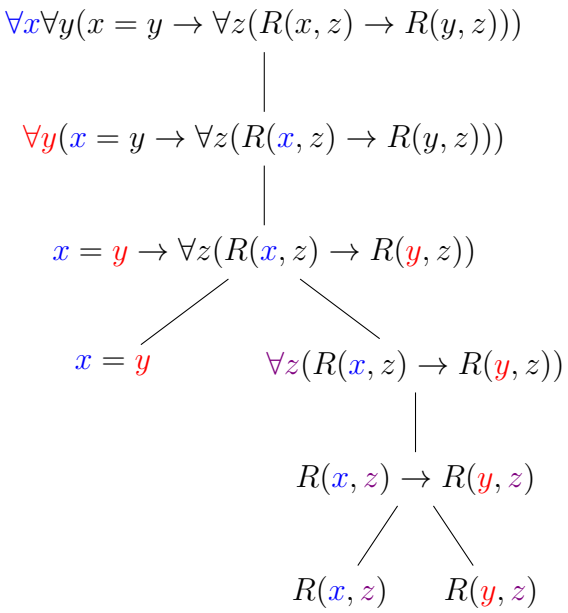


The coloured variables are bound, and the uncoloured ones are free. Notice that z occurs as both a free, and as an unbound variable. The two instances of z are different, and have nothing to do with each other.

A **sentence** is defined as a formula without any free variables (hence all variables are bound).

For example, $\forall x(\texttt{ate}(\texttt{Charlie}, x) \to \texttt{fruit}(x))$ is a valid sentence, however the subformulas aren't $(\texttt{ate}(\texttt{Charlie}, x) \to \texttt{fruit}(x))$ isn't a sentence, since the x is free.

For example, take the slightly more complex formula $\forall x \forall y(x = y \to \forall z(R(x,z) \to R(y,z)))$; we can say it's a sentence. This can be proven by the following formation tree;



Evidently, there are no free variables, as all the atomic L-formula consist of bound variables.

Problems with Free Variables

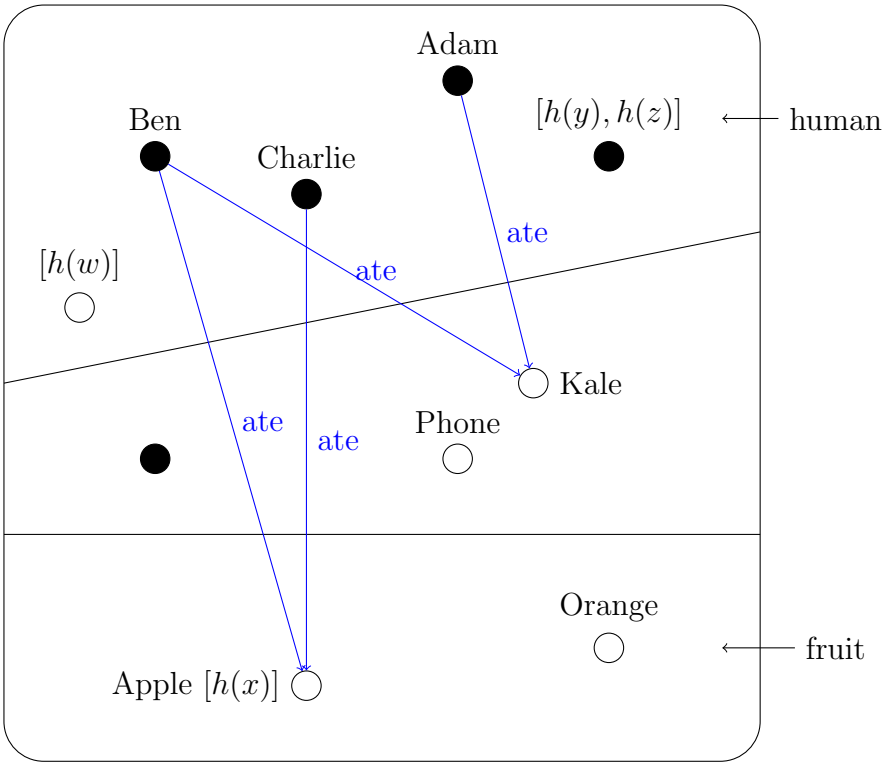
While a sentence can be evaluated to true or false in a structure, the same cannot be said for non-sentences. A formula with free variables doesn't evaluate to a truth value, seeing as they have no meaning in a given L-structure M. For example, $x = 7$ might be true, but we have no way of saying whether it is, since we don't know the value of x in M. Therefore the structure is an **incomplete** situation, since it doesn't fix the meanings of free variables. Note that we need to specify values for fre variables, even if they don't change the answer e.g. $x = x$.

We solve this with **assignments**; suppling a missing value to a free variable. **An assignment does for a variable the same as what a structure / model does for a constant**.

NOTATION: let there be a signature L, have an L-structure M, and let h be an assignment into M. Then for any given L-term t, the value of t in M under h is allocated by;

- t is constant — M: t^M
- t is variable — h: $h(t)$

Reusing the previously drawn model (M, h), because doing diagrams with TikZ is painful;



- the value of the term `Charlie` in M under h is the black node marked 'Charlie' — $\texttt{Charlie}^M$
- the value of the term x in M under h is the white node marked 'Apple' — $h(x)$

This now allows us to evaluate anything without quantifiers; with a fixed L-structure M, and an assignment h, we can write $M, h \vDash A$, or $M, h \nvDash A$ depending on whether A is true in M under h (or not). Therefore the semantics of quantifier-free formulas are as follows;

1. let R represent an n-ary relation in L, $t_1, t_2, ..., t_n$ be L-terms, and t_i has the value a_i in M under h, $\forall i \in [1..n]$. Then, $M, h \vDash R(t_1, t_2, ..., t_n)$ if, and only if the sequence $(a_1, a_2, ..., a_n)$ is in the relation R, otherwise $M, h \nvDash R(t_1, t_2, ..., t_n)$.
2. let t, t' be L-terms, then $M, h \vDash t = t'$ if they both have the same value in M under h, and $M, h \nvDash t = t'$ otherwise.

   Hence in our example, we have $M, h \vDash \texttt{Apple} = x$
3. $M, h \vDash \top$, and $M, h \nvDash \bot$
4. $M, h \vDash A \wedge B$ if $M, h \vDash A$, as well as $M, h \vDash B$, and $M, h \nvDash A \wedge B$ otherwise.
5. $\neg A, A \vee B, A \to B, A \leftrightarrow B$, same as propositional logic.

Problems with Bound Variables

While knowing how to specify values for free variables is useful, the majority of formulas we'll be dealing with involve quantifiers, and hence involve bound variables. Values are not given by the situation, as they are controlled by quantifiers. In general, if we encounter $\exists$, we want to find **some** assignment to make the formula true, and with $\forall$, it requiresthat **all** assignments keep it true.

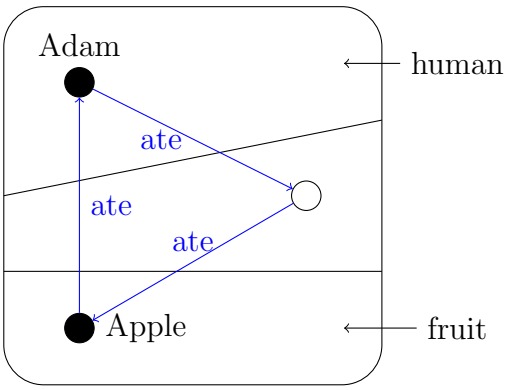
The list below is a continuation of the semantics mentioned in the previous section;

6. $g =_x h$ means $g(y) = h(y)$ for all y, other than x (however $g(x) = h(x)$ is still possible). $g =_x h$ does not imply $g = h$, as we might have $g(x) \neq h(x)$.
7. $M, h \vDash \exists x(A)$ if $M, g \vDash A$ for **some** assignment g, with $g =_x h$, else $M, h \nvDash \exists x(A)$

   (NOT SURE) $M, h \vDash \exists x(A) \triangleq \exists g(g =_x h \wedge [M, g \vDash A])$ - I probably shouldn't use $\vDash$ to express a truth value.
8. $M, h \vDash \forall x(A)$ if $M, g \vDash A$ for **every** assignment g, with $g =_x h$, else $M, h \nvDash \forall x(A)$

   (NOT SURE) $M, h \vDash \forall x(A) \triangleq \forall g(g =_x h \wedge [M, g \vDash A])$

Let us create a simple situation Q;



y \ x	Adam	∘	Apple	
Adam	h_1	h_2	h_3	$=_x$
∘	h_4	h_5	h_6	$=_x$
Apple	h_7	h_8	h_9	$=_x$
	$=_y$	$=_y$	$=_y$	

e.g. $h_2(x) = \circ$, and $h_2(y) = \text{Adam}$

Using this example, we can begin to answer questions about Q, with quantifiers;

- $Q, h_2 \nvDash \texttt{human}(x)$, as $h_2(x)$ refers to $\circ$, which isn't in the section of objects that satisfy the `human` relation.
- $Q, h_2 \vDash \exists x(\texttt{human}(x))$, as there exists an assignment h_1 ($h_1 =_x h_2$, and $M, h_1 \vDash \texttt{human}(x)$).
- $Q, h_7 \nvDash \forall x(\texttt{human}(x))$, as we only need one example to disprove it, either h_8, or h_9 can be used to disprove it, as $M, h_8 \nvDash \texttt{human}(x)$.

For a complex example; $Q, h_4 \vDash \forall x \exists y(\texttt{ate}(x, y))$, we need to establish that $Q, g \vDash \exists y(\texttt{ate}(x, y))$ for $g = h_4, h_5, h_6$;

- $Q, h_4 \vDash \exists y(\texttt{ate}(x, y))$

  h_4 is valid, as $h_4(x) = \text{Adam}$, $h_4(y) = \circ$, and $\texttt{ate}(\text{Adam}, \circ)$ exists in M
- $Q, h_5 \vDash \exists y(\texttt{ate}(x, y))$

  h_8 is valid, as $h_8(x) = \circ$, $h_8(y) = \text{Apple}$, and $\texttt{ate}(\circ, \text{Apple})$ exists in M
- $Q, h_6 \vDash \exists y(\texttt{ate}(x, y))$

  h_3 is valid, as $h_3(x) = \text{Apple}$, $h_3(y) = \text{Adam}$, and $\texttt{ate}(\text{Apple}, \text{Adam})$ exists in M

Therefore, we've proven $Q, h_4 \vDash \forall x \exists y(\texttt{ate}(x, y))$.

Notation

If we had a formula $A(x_1, x_2, ..., x_n)$, it indicates that the free variables of A are in the set $(x_1, x_2, ..., x_n)$, but not all the variables need to occur free.

For example, if we were to have a formula C representing $\forall x(R(x, y) \rightarrow \exists y(S(y, z)))$, we can write $C(y, z)$, $C(a, b, c, d, e, f, x, y, z)$, or even just C. But we cannot write it as $C(x)$, as it doesn't include the free variables.

For an example formula $C(x_1, x_2, ..., x_n)$, and x_i has the value a_i in M under h, $\forall i \in [1..n]$, we can write $M \vDash C(a_1, a_2, ..., a_n)$ instead of $M, h \vDash A$.

Using the same C mentioned above, and we have $h(y) = a$, and $h(z) = b$, instead of writing $M, h \vDash C$, we can write $M \vDash C(a, b)$, or $M \vDash \forall x(R(x, a) \rightarrow \exists y(S(y, b)))$

Let there be an L-structure, M, with L-formula $A(x, y_1, y_2, ..., y_n)$, and $\text{dom}(M) = (a_1, a_2, ..., a_n)$, proving $M \vDash \forall x(A(x, a_1, a_2, ..., a_n))$, with all objects x in $\text{dom}(M)$, and similar for $\exists$.

Evaluation

In practice, there are multiple methods for working out $M \vDash A$;

- working out the natural language meaning of A, and checking it against M
- checking all assignments (using the definitions for semantics of quantifier, and quantifier-free formulas)
- rewriting the formula with equivalences
- using a combination of the three methods

However, generally evaluation is difficult. In most practical cases, with a easily understood formula, evaluating mentally is possible.

Translation

While translating isn't much harder than in propositional logic, it's important to use standard natural language constructions when translating some logical patterns (e.g. $\forall x(A \rightarrow B)$ roughly translates to 'every A is a B'). All variables **must** be eliminated, as it isn't used in natural language. These are some examples;

- $\forall x(\texttt{student}(x) \land \neg(x = \texttt{Adam}) \rightarrow \texttt{ate}(x, \texttt{Apple}))$

  For all x, if x is a student who isn't Adam, then x ate an Apple

  Every student apart from Adam ate an apple
- $\exists x \exists y \exists z(\texttt{ate}(x, y) \land \texttt{ate}(x, z) \land \neg(y = z))$

  There are x, y, and z, such that x ate y, and z, but y is not z

  Something ate at least two different things
- $\forall x(\exists y \exists z((\texttt{ate}(x, y) \land \texttt{ate}(x, z) \land \neg(y = z)) \rightarrow x = \texttt{Adam}))$

  For all x that ate two different things (see the translation from above), then x is Adam

  Anything thate ate two different things is Adam

  Remember that falsity implies anything, hence if no-one ate two things, then the sentence still holds
- $\exists x(\texttt{student}(x) \rightarrow \texttt{ate}(x, \texttt{Apple}))$

  There is an x, such that if x is a student, then x ate an apple

  If there are any students, then at least one student ate an apple.

Common English-to-logic translations exist; for example

- $\forall x(\texttt{student}(x) \to \texttt{human}(x))$ all students are human
- $\exists x(\texttt{student}(x) \wedge \texttt{human}(x))$ some student is human

  $\exists x(\texttt{student}(x) \to \texttt{human}(x))$ is different - it's also true when there are no students. It's a rare case, so if it occurs, it should be checked, especially if x is free in A.
- $\exists x(\texttt{fruit}(x) \wedge \texttt{ate}(\texttt{Adam}, x))$ Adam ate a fruit
- The common patterns you'll likely see are;
  - $\forall x(A \to B)$
  - $\exists x(A \wedge B)$
  - $\forall x(A \wedge B)$
  - $\forall x(A \vee B)$
  - $\exists x(A \vee B)$

We can also use propositional logic to count (the ones in red are less straightforward);

- $\exists x(\texttt{fruit}(x))$ there is at least one fruit
- $\exists x \exists y(\texttt{fruit}(x) \wedge \texttt{fruit}(y) \wedge x \neq y)$ there are at least two fruits
- $\forall x \exists y(\texttt{fruit}(y) \wedge x \neq y)$ there are at least two fruits

  We know at least 1 fruit exists, since there exists a fruit, let it be a. Now, when $x = a$, there has to be at least another fruit b, which isn't the same as a, hence there are at least two fruits.
- $\exists x \exists y \exists z(\texttt{fruit}(x) \wedge \texttt{fruit}(y) \wedge \texttt{fruit}(z) \wedge x \neq y \wedge y \neq z \wedge z \neq x)$ there are at least three fruits
- $\forall x \forall y \exists z(\texttt{fruit}(z) \wedge z \neq x \wedge z \neq y)$ there are at least three fruits
- $\neg \exists x(\texttt{fruit}(x))$ there are no fruits
- $\forall x(\neg \texttt{fruit}(x))$ there are no fruits
- $\neg(\texttt{(there are at least two fruits)})$ there is at most one fruit
- $\forall x \forall y(\texttt{fruit}(x) \wedge \texttt{fruit}(y) \to x = y)$ there is at most one fruit
- $\exists x \forall y(\texttt{fruit}(y) \to y = x)$ there is at most one fruit
- $\texttt{(there is at least one fruit)} \wedge \texttt{(there is at most one fruit)}$ there is exactly one fruit
- $\exists x(\texttt{fruit}(x) \wedge \forall y(\text{fruit}(y) \to x = y))$ there is exactly one fruit
- $\exists x \forall y(\texttt{fruit}(y) \leftrightarrow x = y)$ there is exactly one fruit

Function Symbols

A **function symbol** is like a relation symbol, or constant, but is interpreted in a given structure as a function. Functions have fixed arities (number of arguments), and function symbols are often written as f, or g.

With functions, we can now amend a few definitions; a **signature** is a set of constants, as well as relation and function symbols with specific arities. With a fixed signature L, any constant in L is an L-term, as well as any variable. If f is an n-ary function symbol in L, and the collection of terms $t_1, t_2, ..., t_n$ are also L-terms, then $f(t_1, t_2, ..., t_n)$ is also an L-term. Nothing else is an L-term. Additionally, an L-structure must also define the meaning of any existing function symbols.

Given a unary function symbol f, a binary function symbol g, a constant c, and a variable x, the following are all L-terms;

- c

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

Any of the L-terms not containing an x (variable) are closed (ground) terms.

Let f be an arbitrary function symbol f in L, an L-structure M must define which object from $\text{dom}(M)$ is associated with each sequence of arguments $(a_1, a_2, ..., a_n)$, note that $a_1, a_2, ..., a_n \in \text{dom}(M)$. Formally, we write $f^M : \text{dom}(M)^n \mapsto \text{dom}(M)$

AMENDED NOTATION: let there be a signature L, have an L-structure M, let h be an assignment into M, and let f be a function symbol defined in M. Then for any given L-term t, the value of t in M under h is allocated by;

- t is constant — M: t^M
- t is variable — h: $h(t)$
- t is $f(t_1, t_2, ..., t_n)$, and t_i is a_i in M under h — f: $f^M(a_1, a_2, ..., a_n)$

Therefore the value of a term in M under h is an object in $\text{dom}(M)$, and not a truth value.

Sorts

In logic, the types we are used to in conventional (typed) programming languages, are referred to as **sorts**. Once again, dealing with our previously defined L-structure M, objects might be students, fruits, etc. With a given collection of sorts $\mathbf{s}, \mathbf{s}', \mathbf{s}'', ...$ determined, and named, by the application, it **cannot** generate new sorts such as $(\mathbf{s}, \mathbf{s}')$ (a tuple), and therefore the extra sorts must be explicitly added to the original collection of sorts.

Once again, we will need to adjust some definitions. In order to give each term a sort, each variable, and constant, comes with a specific sort $\mathbf{s}$, for example $c : \mathbf{s}$, $x : \mathbf{s}$. Each n-ary function f, has a template $f : (\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n) \mapsto \mathbf{s}$, where $\mathbf{s}, \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n$ are all sorts. If the all the terms match with the sorts (such that t_i has sort $\mathbf{s}_i$), then $f(t_1, t_2, ..., t_n)$ evaluates to a term of sort $\mathbf{s}$. Otherwise, it doesn't have any meaning. Again; we can extend the requirements for formulas in **many-sorted** logic;

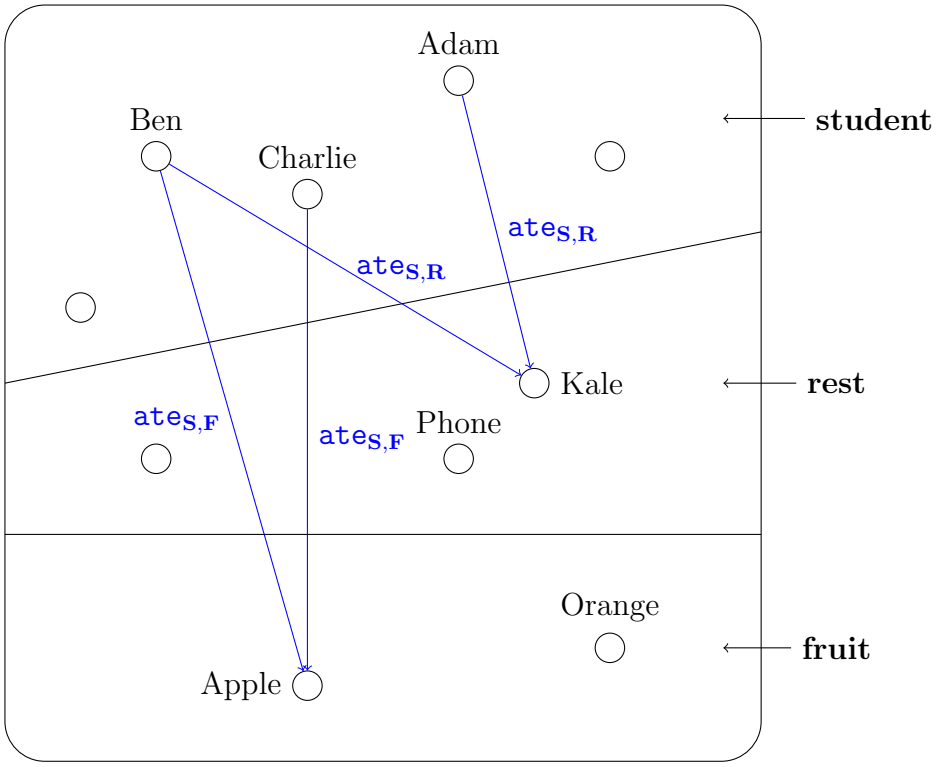
- an n-ary relation has a template $R(\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n)$, where $\mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_n$ are all sorts; if the all the terms match with the sorts (such that t_i has sort $\mathbf{s}_i$), then $R(t_1, t_2, ..., t_n)$ is a formula
- $t = t'$ is only a formula if t and t' have the same sort
- other operations don't change, however we can indicate the sort of bound variables by writing $\forall x : \mathbf{s}(A)$, and $\exists x : \mathbf{s}'(B)$

  This can also be used in order to simplify some formulas; instead of writing $\forall x(\texttt{student}(x) \to \exists y(\texttt{fruit}(y) \wedge \texttt{ate}(x, y)))$, we can write $\forall x : \textbf{student}\ \exists y : \textbf{fruit}\ (\texttt{ate}(x, y))$

Once again, we will be using the structure M, note that $\text{ate}_{\mathbf{S},\mathbf{F}}$ is shorthand for $\text{ate}_{\textbf{student},\textbf{fruit}}$, and $\text{ate}_{\mathbf{S},\mathbf{R}}$ is shorthand for $\text{ate}_{\textbf{student},\textbf{rest}}$. They are **not** two different sorts, an object can only have one sort.

- sorts have the ability to replace some, or even all, unary relation symbols
- as objects can only have 1 sort, the previous definition of `human` would have to be replaced with $\texttt{human}_{\textbf{student}}$, $\texttt{human}_{\textbf{rest}}$, and $\texttt{human}_{\textbf{fruit}}$, but if we don't want to discuss humans of sort **fruit**, we can omit it

- binary relations have to be defined for each pair of sorts, which is why $\texttt{ate}_{\textbf{student,fruit}}$, and $\texttt{ate}_{\textbf{student,rest}}$ have to be written differently, even though they previously referred to the same relation
- an example of how quantifiers can be used with sorts is shown above



Specifications

A program **specification** describes what a program is expected to do. It states the inputs, outputs, and their corresponding types. The **pre-conditions** are conditions in which the program is guaranteed to operate under, and the **post-condition** states the outcome in all cases. This will be explored in much greater detail during **CO141 - Reasoning about Programs**.

A programmer wants the post-condition to be as close to the pre-condition as possible, as that leads to less work. If the pre-condition is weaker (and therefore more general, as there are less assumptions), or if the post-condition is stronger (more results to produce), it means there would be more work for the programmer. On the other hand, the customer wants the opposite to the programmer, since a weaker pre-condition means that there is less work to do prior to the execution of the program, and more is gained after.

The compromise comes when the customer promises pre-conditions for which the program will operate, and the programmer then guarantees the post-conditions, where the program will produce the expected outputs, given the pre-conditions are met.

Specifying Haskell Programs

Let 0, 1, 2, ... be represented by the sort `Nat` (natural numbers), and a sort `[Nat]` for lists of said numbers. This is easier than using Haskell's `Int` sort, which would require one to say $\forall x : \texttt{Int}\ (x \geq 0 \rightarrow A)$, instead of the simpler $\forall x : \texttt{Nat}\ (A)$

We also need to define stuff such as [], :, ++, head, tail, length (#), !!, +, -, ×, etc. for lists and arithmetic.

Here we run into a problem, as a structure must provide meaning for a function symbol on all possible arguments (given the sorts match the template), but operations such as `tail`, `-`, `!!` are partial functions. For example, with `tail`, we can either use a function $\texttt{tail} : \texttt{[Nat]} \mapsto \texttt{[Nat]}$, and give an

arbitrary value of sort [Nat] for tail([]), or we could use a relation Rtail([Nat], [Nat]), such that Rtail(xs, ys) has a truth value of $\top$ when ys is the tail of xs, and $\bot$ otherwise. Normally, we'll use the first option, but note that values of functions on **invalid** arguments are **unpredictable**.

Let us declare the signature L, which does work on lists of sort [Nat]. We need the constants of Nat; 0, 1, 2, ..., as well as the relation symbols $<$, $\leq$, $>$, and $\geq$, which have sort (Nat, Nat), and a set of function symbols (note that I will use $\texttt{Nat}^2$ to mean (Nat, Nat) just to remain consistent with the notation in **CO145 - Mathematical Methods**);

- $+, -, \times : \texttt{Nat}^2 \mapsto \texttt{Nat}$
- [] : [Nat] — a nullary function (constant) to represent the empty list
- cons(:) : (Nat, [Nat]) $\mapsto$ [Nat]
- ++ : $[\texttt{Nat}]^2 \mapsto$ [Nat]
- head : [Nat] $\mapsto$ Nat
- tail : [Nat] $\mapsto$ [Nat]
- # : [Nat] $\mapsto$ Nat
- !! : ([Nat], Nat) $\mapsto$ Nat

We will also use $x, y, z, k, n, m, ...$ to represent variables of Nat, and $xs, ys, zs, ...$ to represent variables of [Nat].

Now we need to assign meaning to all the functions; which will be done in the L-structure M;

- $\#([]) = 0$, or $\forall xs(\#(xs) = 0 \leftrightarrow xs = [])$ — length of empty list
- $\forall x \forall xs(\#(x : xs) = \#(xs) + 1)$ — recursive definition of length
- $\forall x \forall xs((x : xs)!!0 = x)$ — index 0 of list
- $\forall x \forall xs \forall n(n < \#(xs) \rightarrow (x : xs)!!(n + 1) = x!!n)$ — recursive definition of index

  Note how we need to specific $n < \#(xs)$, as we want the **consequent** to only apply if the **antecedent** is $\top$.
- $\forall xs(xs \neq [] \rightarrow \texttt{head}(xs) = xs!!0)$ — head of list
- $\forall x \forall xs(\texttt{head}(x : xs) = x)$ — also head of list, similar to the first definition, and index 0
- $\forall xs \forall ys \forall zs(xs = ys + + zs \leftrightarrow$
  $\#(xs) = \#(ys) + \#(zs) \wedge$
  $\forall n(n < \#(ys) \rightarrow xs!!n = ys!!n) \wedge$
  $\forall n(n < \#(zs) \rightarrow xs!!(n + \#(ys)) = zs!!n))$

Specifying Type

The sorts of the arguments of a function is not from the **pre-condition**, but rather determined by the header of the program.

Specifying Pre-conditions

With a given n-ary formula A, any arguments $a_1, a_2, ..., a_n$ satisfy the pre-condition if, and only if $A(a_1, a_2, ..., a_n)$ is true. For example, $\log(x)$ has the pre-condition $x > 0$ (x is positive), and maxxs has the pre-condition $xs \neq []$ (xs isn't empty). If there are no restrictions other than sort, then 'none', or $\top$ is the pre-condition.

Specifying Post-conditions

The post-condition expresses what the program will do, not how it will do it. Generally, it can look drastically different to the actual code. The formula's **free-variables** should be the arguments of a function. The formula should only evaluate to $\top$ if, and only if the output is as expected.

Let there be some formula $B(x_1, x_2, ..., x_n, y)$ representing the post-condition, corresponding to the n-ary function symbol f, where $x_1, x_2, ..., x_n$ are the input arguments of correct sort, and y is the output. Let there also be the pre-condition $A(x_1, x_2, ..., x_n)$, corresponding to the same function. To express that for any $a, b, ...$ that satisfy the pre-condition, it should return some z that satisfies the post-condition (note that z isn't unique, it can be anything that satisifies the condition). In general, we should write $M \vDash \forall x_1, \forall x_2, ..., \forall x_n(A(x_1, x_2, ..., x_n) \rightarrow \exists y(B(x_1, x_2, ..., x_n, y)))$.

A simple example to start with would be $\texttt{contains}(x, xs)$, with Haskell type `Nat -> [Nat] -> Bool`. There are no pre-conditions, hence the pre-condition would be $\top$ (or 'none'). However, the post condition would be as follows; $\texttt{contains}(x, xs) \leftrightarrow \exists k : \texttt{Nat}\ (k < \#(xs) \wedge xs!!k = x)$.

Therefore, we can write $M \vDash \exists k : \texttt{Nat}\ (k < \#(bs) \wedge bs!!k = a)$, if a is in the list bs in M. In general, $M \vDash \forall x \forall xs(\texttt{contains}(x, xs) \leftrightarrow \exists k : \texttt{Nat}\ (k < \#(xs) \wedge xs!!k = x))$, however it's traditional to use free variables for arguments. Functions with boolean return values are treated as relation symbols, and ones that return other values (in $\text{dom}(M)$) are treated as function symbols.

A more complex example would be one that finds the least value of a list, with type signature `[Nat] -> Nat`. This will also use the previously defined **contains** function. Here with have the pre-condition $xs \neq []$, specifying the input isn't an empty list. As we can't treat this as a relation, we need to specify that $m = \texttt{least}(xs)$ in the post-condition. Now we can use this variable in the newly formed relation $\texttt{contains}(m, xs) \wedge \forall n(\texttt{contains}(n, xs) \rightarrow n \geq m)$.

Another useful post condition to remember would be the condition for a sorted list (which is treated as a relation, since it has signature `[Nat] -> Bool`); $\forall n \forall m(n < m \wedge m < \#(xs) \rightarrow xs!!n \leq xs!!m)$

Validity in Predicate Logic

While this carries many similarities with the previous definitions of valid arguments in propositional logic, we should still define some terms (in order to avoid tedious repetition, let there exist an L-structure M, and an assignment h into M);

An **argument** ($A_1, A_2, ..., A_n$, therefore B) is **valid** if $M, h \vDash A_1,\ M, h \vDash A_2,\ ...,\ M, h \vDash A_n$, then $M, h \vDash B$. This is written as $A_1, A_2, ..., A_n \vDash B$. In the special case that $n = 0$, then $\vDash B$, so B is true within any L-structure under any assignment.

A formula in L, A, is **valid** if we have $M, h \vDash A$, for every L-structure M under any assignment h. Also written as $\vDash A$. On the other hand, if there exists some M, h that $M, h \vDash A$, then it is **satisfiable**.

Two formulas in L, A, B, are **logically equivalent** if we have $M, h \vDash A$ if, and only if $M, h \vDash B$ for all assignments h into every L-structure M.

All valid propositional arguments, such as $A \rightarrow B, A \vDash B$ are valid in predicate logic. However, with predicate logic, we can create very powerful arguments such as $\forall(\texttt{horse}(x) \rightarrow \texttt{animal}(x) \vDash \forall[\exists y(\texttt{head-of}(x, y) \wedge \texttt{horse}(y)) \rightarrow \exists y(\texttt{head-of}(x, y) \wedge \texttt{animal}(y))]$, which couldn't be expressed before (we tried to, right at the start of the first-order predicate logic section).