1.3 Translation into and out of logic

Introduction

Translating predicate logic sentences from logic to English is not much harder than in propositional logic.

But you need to use standard English constructions when translating certain logical patterns.

Example 1.14

$\forall x(A \to B)$. Rough translates as every A is a B.

Also, you can end up with a mess that needs careful simplifying. You'll need common sense!

Variables must be eliminated: English doesn't use them.

Examples

$\forall x(\texttt{lecturer}(x) \land \neg(x = \texttt{Frank}) \to \texttt{bought}(x, \texttt{Texel}))$

'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.)

$\exists x \exists y \exists z(\texttt{bought}(x, y) \land \texttt{bought}(x, z) \land \neg(y = z))$

'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.'

$\forall x(\exists y \exists z(\texttt{bought}(x, y) \land \texttt{bought}(x, z) \land \neg(y = z)) \to x = \texttt{Tony})$

'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 say Tony did buy 2 things, just that noone else did.

Over to you...

1. $\forall x(\texttt{lecturer}(x) \rightarrow \texttt{bought}(x, \texttt{Clyde}))$

2. $\forall x(\texttt{lecturer}(x) \wedge \texttt{bought}(x, \texttt{Clyde}))$

3. $\exists x(\texttt{lecturer}(x) \wedge \texttt{bought}(x, \texttt{Clyde}))$

4. $\exists x(\texttt{lecturer}(x) \rightarrow \texttt{bought}(x, \texttt{Clyde}))$

English to logic translation: advice I

Express the sub-concepts in logic. Then build these pieces into a whole logical sentence.

- Sub-concept 'x is bought'/'x has a buyer': $\exists y\ \texttt{bought}(y, x)$.
- Any bought thing isn't human:
  $\forall x(\exists y\ \texttt{bought}(y, x) \to \neg\, \texttt{human}(x))$.
  Important: $\forall x \exists y(\texttt{bought}(y, x) \to \neg\, \texttt{human}(x))$ would not do.
- Every PC is bought: $\forall x(\texttt{PC}(x) \to \exists y\ \texttt{bought}(y, x))$.
- Some PC has a buyer: $\exists x(\texttt{PC}(x) \land \exists y\ \texttt{bought}(y, x))$.
- No lecturer bought a PC:
  $\neg\exists x(\texttt{lecturer}(x) \land \underbrace{\exists y(\texttt{bought}(x, y) \land \texttt{PC}(y))}_{x \text{ bought a PC}})$.

English-to-logic translation: advice II

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 $\forall x\, \texttt{lecturer}(x) \to \forall x\, \texttt{human}(x)$.
- 'Some lecturer is human': $\exists x(\texttt{lecturer}(x) \wedge \texttt{human}(x))$.
  NOT $\exists x(\texttt{lecturer}(x) \to \texttt{human}(x))$.
- Frank bought a PC: $\exists x(\texttt{PC}(x) \wedge \texttt{bought}(\texttt{Frank}, x))$

The patterns $\forall x(A \to B)$ and $\exists x(A \wedge B)$, are therefore very common.

$\forall x(A \wedge B)$, $\forall x(A \vee B)$, $\exists x(A \vee B)$ also crop up: they say everything/something is A and/or B.

But $\exists x(A \to B)$, especially if x occurs free in A, is extremely rare. If you write it, check to see if you've made a mistake.

Alessandra Russo

English-to-logic translation: advice III (counting)

- **There is at least one PC**:
  $\exists x\ \texttt{PC}(x)$.

- **There are at least two PCs**:
  $\exists x \exists y(\texttt{PC}(x) \wedge \texttt{PC}(y) \wedge x \neq y)$,
  or (more deviously) $\forall x \exists y(\texttt{PC}(y) \wedge y \neq x)$.

- **There are at least three PCs**:
  $\exists x \exists y \exists z(\texttt{PC}(x) \wedge \texttt{PC}(y) \wedge \texttt{PC}(z) \wedge x \neq y \wedge y \neq z \wedge x \neq z)$,
  or $\forall x \forall y \exists z(\texttt{PC}(z) \wedge z \neq x \wedge z \neq y)$.

- **There are no PCs**:
  $\neg \exists x\ \texttt{PC}(x)$

English-to-logic translation: advice III (counting)

- **There is at most one PC**: 3 ways:
  1. $\neg\exists x\exists y(\mathtt{PC}(x) \wedge \mathtt{PC}(y) \wedge x \neq y)$
     This says 'not(there are at least two PCs)' — see above.
  2. $\forall x\forall y(\mathtt{PC}(x) \wedge \mathtt{PC}(y) \to x = y)$
  3. $\exists x\forall y(\mathtt{PC}(y) \to y = x)$

- **There's exactly one PC**: 3 ways:
  1. 'There's at least one PC' $\wedge$ 'there's at most one PC'
  2. $\exists x(\mathtt{PC}(x) \wedge \forall y(\mathtt{PC}(y) \to y = x))$
  3. $\exists x\forall y(\mathtt{PC}(y) \leftrightarrow y = x)$

1.4 Function symbols and sorts

— the icing on the cake ('syntactic sugar')

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, as relation symbols or constants, is interpreted in a structure, but as a function.

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 1.2:

Definition 1.15 (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 1.3:

Definition 1.16 (term)

Fix a signature L.

1. Any constant in L is an L-term.
2. Any variable is an L-term.
3. If f is an n-ary function symbol in 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 1.17

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

c	$g(x, x)$	(x is a variable, as usual)
$f(c)$	$g(f(c), g(x, x))$	

Terms on the left are closed, or ground, terms. Those on the right are not.

Semantics of function symbols

We need to extend Definition 1.5 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 $\mathrm{dom}(M)$) f associates with each sequence $(a_1, \ldots, a_n)$ of objects in $\mathrm{dom}(M)$.
We write this object as $f^M(a_1, \ldots, a_n)$. There must be such a value.
[Formally, f^M is a function $f^M : \mathrm{dom}(M)^n \to \mathrm{dom}(M)$.]
A 0-ary function symbol is like a constant.

Example 1.18

In arithmetic, M might say $+, \times$ are addition and multiplication of numbers: it associates 5 with $2 + 3$, 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 nobody would want such an M.

Evaluating terms with function symbols

We can now extend Definition 1.11:

Definition 1.19 (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 t^M 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 $t_1, \ldots, t_n$ 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* $\mathrm{dom}(M)$, *rather than true or false!*

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

Example: arithmetic terms

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

- constants $\underline{\mathtt{0}}, \underline{\mathtt{1}}, \underline{\mathtt{2}}, \ldots$ (I use underlined typewriter font to avoid confusion with actual numbers $0, 1, \ldots$)
- 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 1.16 and 1.4.
Some terms: $x + \underline{\mathtt{1}}$, $\underline{\mathtt{2}} + (x + \underline{\mathtt{5}})$, $(\underline{\mathtt{3}} \times \underline{\mathtt{7}}) + x$. Not $x + y + z$.
Formulas: $\underline{\mathtt{3}} \times x > \underline{\mathtt{0}}$, $\forall x(x > \underline{\mathtt{0}} \to x \times x > x)$.

Many-sorted logic

Introduction

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

E.g., some objects in a structure M may be lecturers, others may be PCs, numbers, etc.

We can handle this with unary relation symbols, or with 'many-sorted first-order logic'. We'll use many-sorted logic mainly to specify programs.

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

These sorts do not generate extra sorts, like $\mathbf{s} \to \mathbf{s}'$ or $(\mathbf{s}, \mathbf{s}')$.

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 1.16), to give each term a sort:

- each variable and constant comes with a sort $\mathbf{s}$. To indicate which sort it is, we write $x : \mathbf{s}$ and $c : \mathbf{s}$. There are infinitely many variables of each sort.
- each n-ary function symbol f comes with a template

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

  where $\mathbf{s}_1, \ldots, \mathbf{s}_n$, and $\mathbf{s}$ are sorts.
  Note: $(\mathbf{s}_1, \ldots, \mathbf{s}_n) \to \mathbf{s}$ is not itself a sort.
- For such an f and terms $t_1, \ldots, t_n$, if t_i has sort $\mathbf{s}_i$ (for each i) then $f(t_1, \ldots, t_n)$ is a term of sort $\mathbf{s}$.
  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 $)\forall)\to$.

Formulas in many-sorted logic

- Each n-ary relation symbol R comes with a template $R(\mathbf{s}_1, \ldots, \mathbf{s}_n)$, where $\mathbf{s}_1, \ldots, \mathbf{s}_n$ are sorts.
  For terms $t_1, \ldots, t_n$, if t_i has sort $\mathbf{s}_i$ (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

$$\begin{array}{lll} \forall x : \mathbf{s}\ A & \text{instead of just} & \forall x A \\ \exists x : \mathbf{s}\ A & \text{instead of just} & \exists x A \end{array}$$

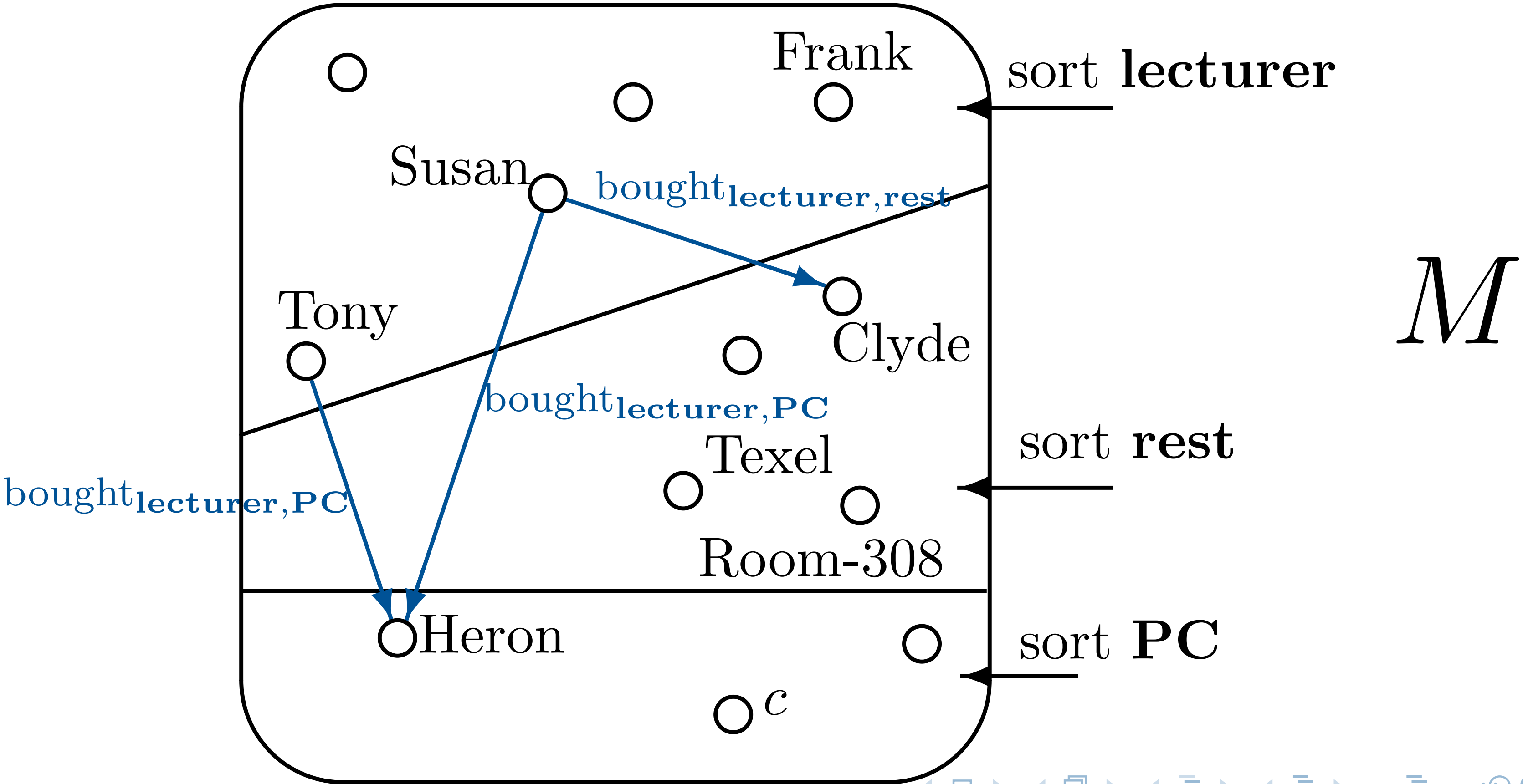
  if x has sort $\mathbf{s}$. Alternatively, declare the variables of each sort.

E.g., roughly, you can write $\forall x : \textbf{lecturer}\ \exists y : \textbf{PC}(\texttt{bought}(x, y))$ instead of $\forall x(\texttt{lecturer}(x) \to \exists y(\texttt{PC}(y) \wedge \texttt{bought}(x, y)))$.

L-structures for many-sorted L — example

Let L be a many-sorted signature. An L-structure is defined as before (Definition 1.5 + slide 70), but additionally it allocates each object in its domain to a single sort. No sort should be empty.

E.g., if L has sorts **lecturer**, **PC**, **rest**, an L -structure looks like:



Interpretation of L-symbols in L-structures

Let M be a many-sorted L-structure.

- For each constant $c : \mathbf{s}$ in L, M must say which object of sort $\mathbf{s}$ in $\mathrm{dom}(M)$ is 'named' by c.
- For each function symbol $f : (\mathbf{s}_1, \ldots, \mathbf{s}_n) \to \mathbf{s}$ in L and all objects $a_1, \ldots, a_n$ in $\mathrm{dom}(M)$ of sorts $\mathbf{s}_1, \ldots, \mathbf{s}_n$, respectively, M must say which object $f^M(a_1, \ldots, a_n)$ of sort $\mathbf{s}$ is associated with $(a_1, \ldots, a_n)$ by f.
  M doesn't say anything about $f(b_1, \ldots, b_n)$ if $b_1, \ldots, b_n$ don't all have the right sorts.
- For each relation symbol $R(\mathbf{s}_1, \ldots, \mathbf{s}_n)$ in L, and all objects $a_1, \ldots, a_n$ in $\mathrm{dom}(M)$ of sorts $\mathbf{s}_1, \ldots, \mathbf{s}_n$, respectively, M must say whether $R(a_1, \ldots, a_n)$ is true or not.
  M doesn't say anything about $R(b_1, \ldots, b_n)$ if $b_1, \ldots, b_n$ don't all have the right sorts.

Notes

1. Sorts can replace some or all unary relation symbols.
2. As in Haskell, each object has only 1 sort, not 2.
   So for M above, **`human`** would have to be implemented as three unary relation symbols: $\texttt{human}_{\mathbf{lecturer}}$, $\texttt{human}_{\mathbf{PC}}$, $\texttt{human}_{\mathbf{rest}}$.
   But if (e.g.) you don't want to talk about human objects of sort **PC**, you can omit $\texttt{human}_{\mathbf{PC}}$.
3. We need a binary relation symbol $\texttt{bought}_{\mathbf{s},\mathbf{s}'}$ for each pair $(\mathbf{s}, \mathbf{s}')$ of sorts (unless **s**-objects are not expected to buy $\mathbf{s}'$-objects).
4. Messy alternative: use sorts for human lecturer, PC-lecturer, etc — all possible types of object.

Quantifiers in many-sorted logic

Semantics of formulas is defined as before (Definition 1.12), but assignments must respect sorts of variables.

In a nutshell: if variable x has sort **s**, then $\forall x$ and $\exists x$ range over objects of sort **s** only.

For example, $\forall x : \mathbf{lecturer}\ \exists y : \mathbf{PC}(\mathtt{bought}_{\mathtt{lecturer},\mathtt{PC}}(x, y))$ is true in a structure if every object of sort **lecturer** bought an object of sort **PC**.

It is not the same as $\forall x\, \exists y\ \mathtt{bought}(x, y)$.
It does not say that every **PC**-object bought a **PC**-object as well (etc etc).

Do not get worried about many-sorted logic. It looks complicated, but it's easy once you practise. It is there to help you (like types in programming), and it can make life easier.