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 $\text{dom}(M)$ 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 $\text{dom}(M)$ 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 $\text{dom}(M)$ 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: $\texttt{human}_{\textbf{lecturer}}$, $\texttt{human}_{\textbf{PC}}$, $\texttt{human}_{\textbf{rest}}$.
   But if (e.g.) you don't want to talk about human objects of sort **PC**, you can omit $\texttt{human}_{\textbf{PC}}$.
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, $\forall x : \textbf{lecturer}\ \exists y : \textbf{PC}(\texttt{bought}_{\texttt{lecturer,PC}}(x, y))$ is true in a structure if every object of sort **lecturer** bought an object of sort **PC**.

It is not the same as $\forall x\, \exists y\ \texttt{bought}(x, y)$.
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.

Application of logic: specifications

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).

3.1 Logic for specifying Haskell programs

Introduction

A very precise way to specify properties of Haskell programs is to use first-order logic.

(Logic can also be used for Java, etc.)

Next term: gory details.

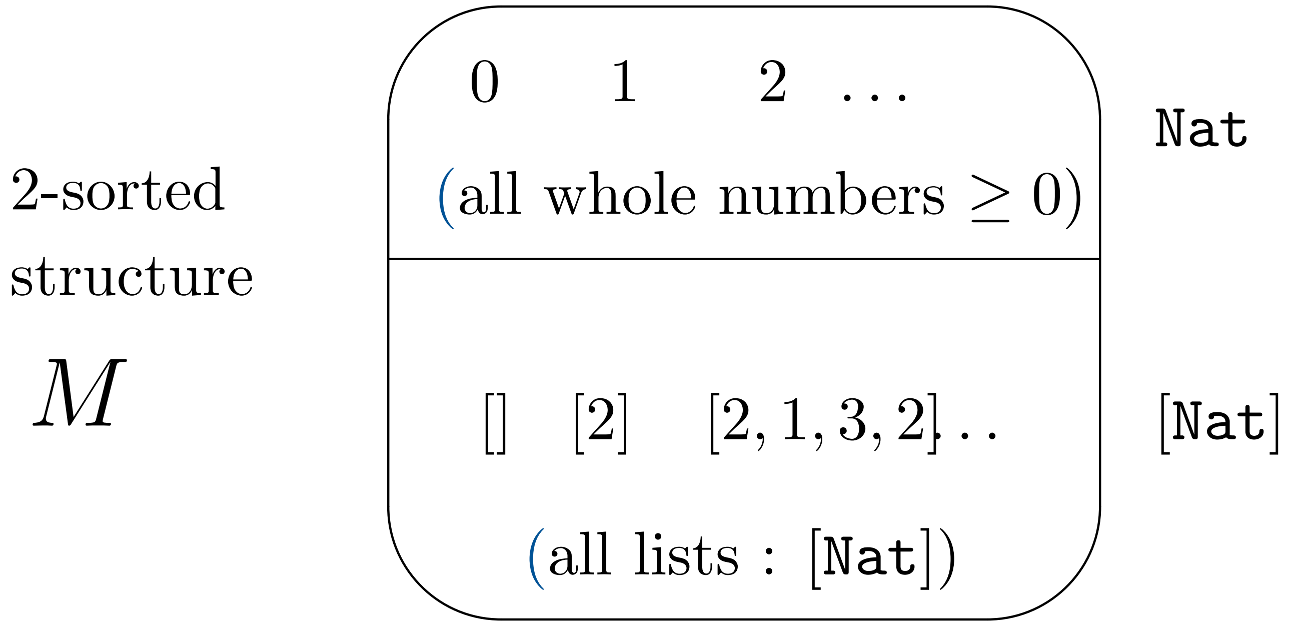
This term: a gentle taster (but still very powerful).

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 actual Haskell `Int` is more longwinded: must keep saying $n \geq 0$ etc.)

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



3.2 Signature for lists

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

We want $[]$, : (cons), $++$, head, tail, length (which we write as $\sharp$), !!.

And $+, -$, etc., for arithmetic.

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

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).

But what is the head or tail of the empty list? What is $xs\,!!\,\sharp(xs)$? What is $34 - 61$?

Two solutions (for `tail`; the others are similar):

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

We'll take the function symbol option (1), as it leads to shorter formulas. But always beware:

Warning: values of functions on 'invalid' arguments are 'unpredictable'.

Lists in first-order logic: summary

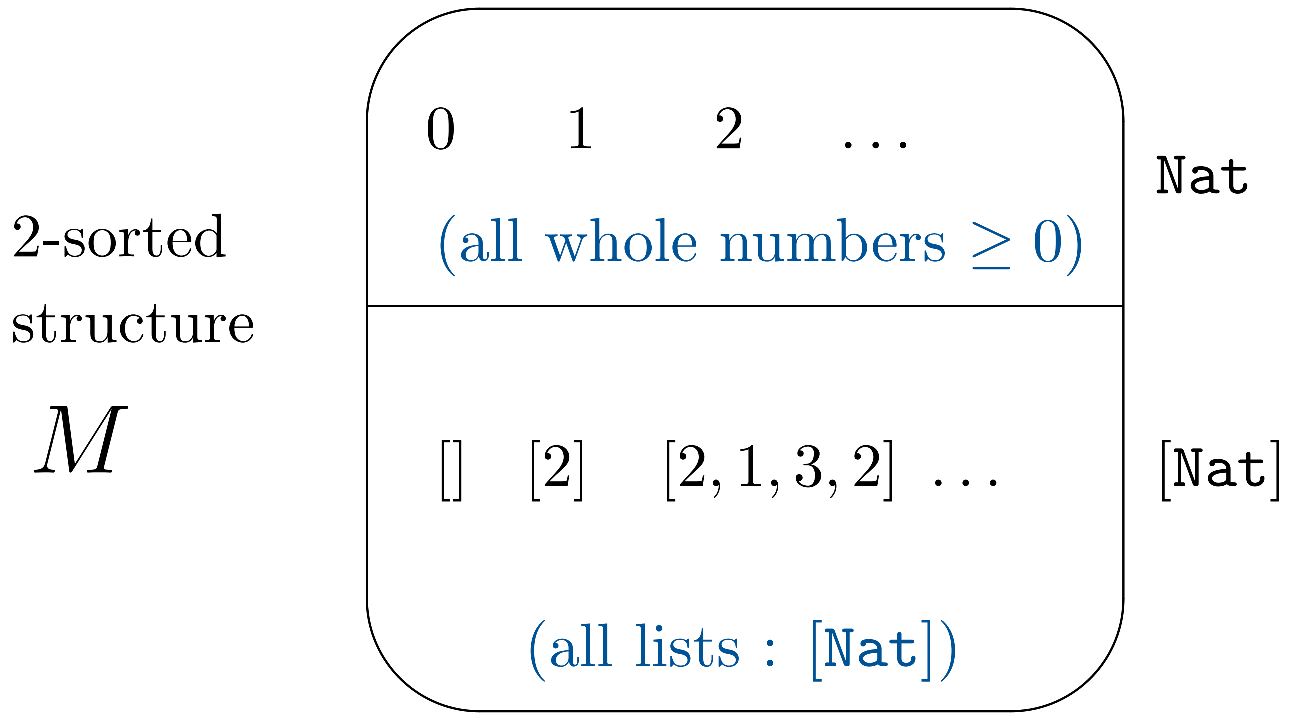
Now we can define a signature L suitable for lists of type $[\mathtt{Nat}]$.

- L has constants $\underline{\mathtt{0}}, \underline{\mathtt{1}}, \ldots : \mathtt{Nat}$, relation symbols $<, \leq, >, \geq$ of sort ($\mathtt{Nat}$,$\mathtt{Nat}$), and function symbols
  - $+, -, \times : (\mathtt{Nat}, \mathtt{Nat}) \to \mathtt{Nat}$
  - $[] : [\mathtt{Nat}]$ (a constant to name the empty list)
  - $\mathtt{cons}(:)\ : (\mathtt{Nat}, [\mathtt{Nat}]) \to [\mathtt{Nat}]$
  - $++ : ([\mathtt{Nat}], [\mathtt{Nat}]) \to [\mathtt{Nat}]$
  - $\mathtt{head} : [\mathtt{Nat}] \to \mathtt{Nat}$
  - $\mathtt{tail} : [\mathtt{Nat}] \to [\mathtt{Nat}]$
  - $\sharp : [\mathtt{Nat}] \to \mathtt{Nat}$
  - $!! : ([\mathtt{Nat}], \mathtt{Nat}) \to \mathtt{Nat}$

  We write the constants as $\underline{\mathtt{0}}, \underline{\mathtt{1}}, \ldots$ to avoid confusion with actual numbers $0, 1, \ldots$
- Let $x, y, z, k, n, m \ldots$ be variables of sort $\mathtt{Nat}$.
  Let $xs, ys, zs, \ldots$ be variables of sort $[\mathtt{Nat}]$.

Semantics

Let M be the L-structure



The L-symbols are interpreted in the natural way: ++ as concatenation of lists, etc.

We define $34 - 61$, `tail`([]), etc. arbitrarily. So don't assume they have the values you might expect.

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:

$\sharp([\,]) = \underline{0}$ Indeed, $\forall xs(\sharp(xs) = \underline{0} \leftrightarrow xs = [\,])$ is also true.
$\forall x \forall xs((\sharp(x : xs) = \sharp(xs) + \underline{1}) \wedge ((x : xs)!!\underline{0} = x))$
$\forall x \forall xs \forall n(n < \sharp(xs) \to (x : xs)!!(n + \underline{1}) = xs!!n)$
The '$n < \sharp(xs)$' is necessary. $xs!!n$ could be anything if $n \geq \sharp(xs)$.

$\forall xs(xs \neq [\,] \to \texttt{head}(xs) = xs!!\underline{0})$, and $\forall x \forall xs(\texttt{head}(x : xs) = x)$

$\forall x \forall xs(\texttt{tail}(x : xs) = xs)$

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