3.3 Specifying Haskell functions

Now we know how to use logic to say things about lists, we can use logic to specify Haskell functions. There are three bits to it.

1. Type Information

- This is stuff like 'the first argument is a `Nat` and the second a [`Nat`]'.
- It is determined by the program header.
- It is not part of the pre-condition (coming next)

2. Pre-conditions in logic

The pre-condition expresses restrictions on the arguments or parameters that can be legally passed to a function.

To do a pre-condition in logic, you write a formula $A(x_1, \ldots, x_n)$ so that any arguments $a_1, \ldots, a_n$ satisfy the intended pre-condition ('are legal') if and only if $A(a_1, \ldots, a_n)$ is true.
E.g., for the function $\log(x)$, you'd want a pre-condition of $x > \underline{0}$.
For $\max xs$ you'd want $xs \neq [\,]$.

Pre-conditions are usually very easy to write:

- xs is not empty: use $xs \neq [\,]$.
- n is positive: use $n > \underline{0}$.

If there are no restrictions on the arguments beyond their type information, you can write 'none', or $\top$, as pre-condition. This is perfectly normal and is no cause for alarm.

3. Post-conditions in logic

The post-condition expresses the required connection between the input and output of a function.

It expresses what the program does, but not how the program works. It can look completely different from the program code.

To do a post-condition 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.

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 satisfying the pre-condition (if any), the function should return some c such that $M \models A(a, b, c)$.

There is no requirement that c be unique! We could well 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 $pre, post$.

Example: specifying the function 'isin'

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

- This is $\texttt{isin}(x, xs) \leftrightarrow \exists k : \texttt{Nat}(k < \sharp(xs) \wedge xs!!k = x)$.
  I used `(E)` and `&`, as I can't type $\exists, \wedge$ in Haskell.
  Similarly, use `\/` or `|` for $\vee$, `(A)` for $\forall$, and `~` or `!` for $\neg$.
- For any number a and list bs in M, we have
  $M \models \exists k : \texttt{Nat}(k < \sharp(bs) \wedge bs!!k = a)$ just when a occurs in bs.
  So we have the intended post-condition for `isin`.
- $\forall x \forall xs(\texttt{isin}(x, xs) \leftrightarrow \exists k : \texttt{Nat}(k < \sharp(xs) \wedge xs!!k = x))$ would be better, but it's traditional to use free variables for the function arguments. Implicitly, though, they are universally quantified.
- We treat functions with boolean values (like `isin`) as relation symbols. Functions that return number or list values (values in $\text{dom}(M)$) are treated as function symbols.

Least entry

Write $in(x, xs)$ for the formula $\exists k : \texttt{Nat}(k < \sharp(xs) \wedge xs!!k = x)$.
Then $in(m, xs) \wedge \forall n(in(n, xs) \rightarrow n \geq m)$
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(in(n,xs) -> n>=m), where m = least xs
```

Ordered (or sorted) lists

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

Use lists of pointers (below, they are ts, us):

$$\begin{aligned}\texttt{merge}(xs, ys, zs) \;\leftrightarrow\quad & \sharp xs + \sharp ys = \sharp zs \\ & \wedge \exists ts \exists us \big(\sharp ts = \sharp xs \wedge \sharp us = \sharp ys \\ & \qquad \wedge \forall n (n < \sharp zs \rightarrow in(n, ts{+}{+}us)) \\ & \qquad \wedge sorted(ts) \wedge sorted(us) \\ & \qquad \wedge \forall n (n < \sharp xs \rightarrow zs!!(ts!!n) = xs!!n) \\ & \qquad \wedge \forall n (n < \sharp ys \rightarrow zs!!(us!!n) = ys!!n) \big)\end{aligned}$$

ts gets the positions in zs that come from xs, and us gets the rest.
For example, merge([7,7], [6,8,5], [6,8,7,5,7]),
and indeed we can take $ts = [\quad 2,\ \ 4]$
and $us = [0,1,\ \ 3\ \ ]$.

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.

Arguments, validity

Valid Arguments

Our experience with propositional logic tells us how to define 'valid argument' etc.

Definition 4.1 (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. Let L be a signature.

Definition 4.2 (valid formula)

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

Definition 4.3 (satisfiable formula)

An L-formula A is satisfiable if for some L-structure M and assignment h into M, we have $M, h \models A$.

Validity, satisfiability, equivalence

Definition 4.4 (equivalent formulas)

L-formulas A, B are logically equivalent if for every L-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 definitions that we have seen in propositional logic, 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: for example,
  $\forall x(\texttt{horse}(x) \to \texttt{animal}(x)) \models \forall x[\exists y(\texttt{headof}(x,y) \wedge \texttt{horse}(y)) \to \exists y(\texttt{headof}(x,y) \wedge \texttt{animal}(y))]$.

  A horse is an animal $\models$ the head of a horse is the head of an animal.

Deciding if an argument $A_1, \ldots, A_n \models B$ is valid is extremely hard in general. We can't just check that all L-structures + assignments that make $A_1, \ldots, A_n$ true also make B true (like truth tables), because there are infinitely many L-structures (some are infinite!)

Theorem 4.5 (Church, 1936)

No computer program can be written to identify precisely the valid arguments of predicate logic.

Useful ways of validating arguments

In spite of Theorem 4.5, we can often verify in practice that a particular argument 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.