Proving that a function meets its specification

We can show that a program satisfies its specification by proving that it satisfies all of the axioms in the specification. This is the main point of making the specification precise.

For example, here is a program for `maxelem : int list -> int`:

```
fun maxelem nil = 0
  | maxelem([a]) = a
  | maxelem(a::b::l) = if a>maxelem(b::l) then a else maxelem(b::l)
```

Since the definition is recursive, we can use structural induction to prove that it satisfies its specification:

```
Forall l => l<>nil implies member(maxelem l, l)
Forall (a,l) => member(a,l) implies (maxelem l) >= a
```

The specification and the program make use of other functions (e.g. `member`, `>=`). A really formal proof would have to refer to the definitions of these or to previously-established properties of them. We will be a little less formal and just use properties as required without proof.

Theorem: `Forall l => l<>nil implies member(maxelem l, l)`
Proof: By induction on `l`.
Base cases:

`l=nil`: the precondition is false
`l=[a]`: `member(maxelem [a],[a]) = member(a,[a]) = true`

Step case: `l=a::b::l'`

Inductive hypothesis: `member(maxelem b::l',b::l')`
To prove: `member(maxelem a::b::l',a::b::l')`.

Case 1: `a > maxelem(b::l')`
`member(maxelem a::b::l',a::b::l') = member(a,a::b::l') = true`

Case 2: `maxelem(b::l') >= a`
`member(maxelem a::b::l',a::b::l') = member(maxelem(b::l'),a::b::l') = true` (by IH and properties of `member`).

Theorem: `Forall (a,l) => member(a,l) implies (maxelem l) >= a`
Proof: By induction on `l`.
Base cases:

`l=nil`: there is no `a` such that `member(a,l)`
`l=[b]`: `member(a,[b])` implies `a=b` and so `maxelem [b] = b >= a`

Step case: `l=a::b::l'`

Inductive hypothesis: `member(c,b::l') implies (maxelem(b::l')) >= c`
To prove: `member(c,a::b::l') implies (maxelem(a::b::l')) >= c`

Case 1: `a > maxelem(b::l')`
Then `maxelem(a::b::l') = a`. For every `c` such that `member(c,a::b::l')`, either `c=a` and so `maxelem(a::b::l') = a >= c`, or `member(c,b::l')` and so `maxelem(a::b::l') = a > maxelem(b::l')` (by case assumption) `>= c` (by IH).

Case 2: `maxelem(b::l') >= a`
Then `maxelem(a::b::l') = maxelem(b::l')`. For every `c` such that `member(c,a::b::l')`, either `c=a` and so `maxelem(a::b::l') = maxelem(b::l') >= a` (by case assumption) `= c` or `member(c,b::l')` and so `maxelem(a::b::l') = maxelem(b::l') >= c` (by IH).

Structural induction. Recall that for natural numbers we had the following induction principle:

To prove $\forall n.P(n)$:
1. Base case: Prove $P(0)$
2. Step case: Assume $P(n)$ (the inductive hypothesis) and prove $P(n+1)$

For lists, we have structural induction:

To prove $\forall l.P(l)$:
1. Base case: Prove $P(\texttt{nil})$
2. Step case: Assume $P(l)$ (the inductive hypothesis) and prove $P(a::l)$

Compare this with the definition of the list datatype:

```
datatype 'a list = nil | :: of 'a * 'a list
```

The base case of structural induction corresponds to the constructor that takes no arguments of type `'a list` (i.e. `nil`) and the step case corresponds to the constructor that does takes an argument of type `'a list`.

(For the proofs above, we used the following derived induction principle because of the form of the `maxelem` function:

To prove $\forall l.P(l)$:
1. Base case 1: Prove $P(\texttt{nil})$
2. Base case 2: Prove $P(\texttt{[}a\texttt{]})$
3. Step case: Assume $P(b::l)$ (the inductive hypothesis) and prove $P(a::b::l)$

This can be derived from usual structural induction for lists: the second base case and the step case in the derived induction principle come from a case analysis in the step case of the usual induction principle.)

Structural induction principles can be derived from the definition of other user-defined datatypes. For instance, consider the tree datatype defined earlier:

```
datatype 'a tree = Empty | Node of 'a tree * 'a * 'a tree
```

Here's the corresponding structural induction principle:

To prove $\forall t.P(t)$:
1. Base case: Prove $P(\texttt{Empty})$
2. Step case: Assume $P(t)$ *and* $P(t')$ (the inductive hypotheses), prove $P(\texttt{Node}(t,x,t'))$

We get to make *two* inductive hypotheses because `Node` takes two tree arguments.

Recall the functions:

```
fun height Empty = 0
  | height(Node(t1,_,t2)) = 1 + Int.max(height t1,height t2)

fun size Empty = 0
  | size(Node(t1,_,t2)) = 1 + size t1 + size t2
```

Theorem: `Forall t => size t >= height t`
Proof: By induction on `t`.
Base case: `t=Empty`

`size Empty = 0 = height Empty`

Step case: `t=Node(t1,x,t2)`

Inductive hypothesis 1: `size t1 >= height t1`
Inductive hypothesis 2: `size t2 >= height t2`
To prove: `size(Node(t1,x,t2)) >= height(Node(t1,x,t2))`

`size(Node(t1,x,t2)) = 1 + size t1 + size t2 >= 1 + height t1 + size t2`
(by IH1) `>= 1 + height t1 + height t2` (by IH2) `>= 1 + Int.max(height t1, height t2)`
(property of `Int.max`) `= height(Node(t1,x,t2))`

We can read off a structural induction principle from the definition of most user-defined ML datatypes. Another example:

```
datatype 'a tree = Empty
                 | Leaf of 'a
                 | Node1 of 'a * 'a tree
                 | Node2 of 'a tree * 'a * 'a tree
```

Here is the corresponding induction principle:

To prove $\forall t.P(t)$:
1. Base case for `Empty`: Prove $P(\texttt{Empty})$
2. Base case for `Leaf`: Prove $P(\texttt{Leaf}(x))$
3. Step case for `Node1`: Assume $P(t)$ and prove $P(\texttt{Node1}(x,t))$
4. Step case for `Node2`: Assume $P(t)$ and $P(t')$ and prove $P(\texttt{Node2}(t,x,t'))$

`Leaf` is a base case even though it has an argument, because the argument is not a tree.

Things get a little more complicated with mutually recursive datatypes, or when one recursive datatype is used in the definition of another. Then to prove a property of values of one type you need to come up with an appropriate property of values of the other type(s), and do induction to prove both at once.

The details are left as an exercise; consider for instance

```
datatype 'a tree = Empty | Node of 'a * 'a tree list

datatype expr = Numeral of int
          | Var of string
          | Plus of expr * expr
          | Times of expr * expr
          | Comresult of com * expr
and com = Assign of string * expr
        | While of expr * com
```

Some user-defined types do not give rise to an induction principle. Consider for instance:

```
datatype D = Int of int | Fun of D->D
```

The problem here is that a value of the form $\texttt{Fun}(f)$ is not constructed directly from another value of type `D`.

This kind of problem arises whenever the arguments of constructors have recursive occurrences of the type being defined on the left-hand side of `->` (in general, in a "negative position"). Such datatypes do not arise very often.