~ Lecture X ~

Keywords:

testing and verification; rigorous and formal proofs; structural induction on lists; law of extensionality; multisets; structural induction on trees.

References:

- [MLWP, Chapter 6]

Testing and verification

Functional programs are easier to reason about

- We wish to establish that a program is correct, in that it meets its specification.
- **Testing.**

  Try a selection of inputs and check against expected results.

  There is no guarantee that all bugs will be found.
- **Verification.**

  Prove that the program is correct within a mathematical model.

  Proofs can be long, tedious, complicated, hard, *etc.*

Rigorous vs. formal proof

A rigorous proof is a convincing mathematical argument

- **Rigorous proof.**
  - What mathematicians and some computer scientists do.
  - Done in the mathematical vernacular.
  - Needs clear foundations.
- **Formal proof.**
  - What logicians and some computer scientists study.
  - Done within a formal proof system.
  - Needs machine support.

Modelling assumptions

- Proofs treat *programs as mathematical objects*, subject to mathematical laws.
- Only *purely functional programs* will be allowed.
- *Types* will be interpreted *as sets*, which restricts the form of `datatype` declarations.
- We shall allow only *well-defined expressions*. They must be *legally typed*, and must denote *terminating computations*. By insisting upon termination, we can work within elementary set theory.

Structural induction on lists

Let P be a property on lists that we would like to prove.

To establish

$$P(\ell) \text{ for all } \ell \text{ of type } \tau\ \texttt{list}$$

by *structural induction*, it suffices to prove.

1. The *base case*: $P([\,])$.
2. The *inductive step*: For all h of type τ and t of type $\tau\ \texttt{list}$,

$$P(t) \text{ implies } P(h::t)$$

Example: No list equals its own tail.

For all h of type τ and all t of type $\tau\ \texttt{list}$, $h::t \neq t$.

Applications

```
fun nlen [] = 0
  | nlen (h::t) = 1 + nlen(t) ;

fun len l
  = let
      fun addlen( n , [] ) = n
        | addlen( n , h::t ) = addlen( n+1 , t )
    in
      addlen( 0 , l )
    end ;
```



```
infix @ ;
fun [] @ l = l
  | (h::t) @ l = h :: ( t@l ) ;

fun nrev [] = []
  | nrev (h::t) = (nrev t) @ [h] ;

fun revApp( [] , l ) = l
  | revApp( h::t , l ) = revApp( t , h::l ) ;
```


- For all lists ℓ, ℓ_1, and ℓ_2,

1. $\texttt{nlen}(\ell_1 @ \ell_2) = \texttt{nlen}(\ell_1) + \texttt{nlen}(\ell_2)$.
2. $\texttt{revApp}(\ell_1, \ell_2) = \texttt{nrev}(\ell_1) @ \ell_2$.
3. $\texttt{nrev}(\ell_1 @ \ell_2) = \texttt{nrev}(\ell_2) @ \texttt{nrev}(\ell_1)$.
4. $\ell @ [\,] = \ell$.
5. $\ell @ (\ell_1 @ \ell_2) = (\ell @ \ell_1) @ \ell_2$.
6. $\texttt{nrev}(\texttt{nrev}(\ell)) = \ell$.
7. $\texttt{nlen}(\ell) = \texttt{len}(\ell)$.

Equality of functions

The *law of extensionality* states that functions $f, g : \alpha \to \beta$ are equal iff $f(x) = g(x)$ for all $x \in \alpha$.

Example:

- Associativity of composition.

  ```
  infix o;
  fun (f o g) x = f( g x ) ;
  ```

  For all $f : \alpha \to \beta$, $g : \beta \to \gamma$, and $h : \gamma \to \delta$,

  $$\boxed{h \circ (g \circ f) = (h \circ g) \circ f \quad : \alpha \to \delta}$$

- `fun id x = x ;`

  For all $f : \alpha \to \beta$, $\boxed{f \circ id = f = id \circ f}$

Applications

```
fun map f [] = []
  | map f (h::t) = (f h) :: map f t ;
```

1. Functoriality[a] of `map`.

   $$\boxed{\mathtt{map}\ \mathtt{id} = \mathtt{id}}$$

   For all $f : \alpha \to \beta$ and $g : \beta \to \gamma$,

   $$\boxed{\mathtt{map}(g \circ f) = \mathtt{map}(g) \circ \mathtt{map}(f) \quad : \alpha\ \mathtt{list} \to \gamma\ \mathtt{list}}$$

2. For all $f : \alpha \to \beta$, and $\ell_1, \ell_2 : \alpha\ \mathtt{list}$,

   $$\boxed{\mathtt{map}\ f\ (\ell_1 \mathbin{@} \ell_2) = (\mathtt{map}\ f\ \ell_1) \mathbin{@} (\mathtt{map}\ f\ \ell_2) \quad : \beta\ \mathtt{list}}$$

3. For all $f : \alpha \to \beta$,

   $$\boxed{(\mathtt{map}\ f) \circ \mathtt{nrev} = \mathtt{nrev} \circ (\mathtt{map}\ f) \quad : \beta\ \mathtt{list}}$$

[a] This is a technical term from *Category Theory*.

Multisets

Multisets are a useful abstraction to specify properties of functions operating on lists.

- A *multiset*, also referred to as a *bag*, is a collection of elements that takes account of their number but not their order.

  Formally, a multiset m on a set S is represented as a function $m : S \to \mathbb{N}$.

- Some ways of forming multisets:

  1. the *empty multiset* contains no elements and corresponds to the constantly 0 function

     $$\emptyset : x \mapsto 0$$

  2. the *singleton s multiset* contains one occurrence of s, and corresponds to the function

     $$\langle s \rangle : x \mapsto \begin{cases} 1 & \text{, if } x = s \\ 0 & \text{, otherwise} \end{cases}$$

  3. the *multiset sum* m_1 and m_2 contains all elements in the multisets m_1 and m_2 (accumulating repetitions of elements), and corresponds to the function

     $$m_1 \uplus m_2 : x \mapsto m_1(x) + m_2(x)$$

An application

Consider

```
fun take( [] , _ ) = []
  | take( h::t , i )
      = if i > 0
        then h :: take( t , i-1 )
        else [] ;

fun drop( [], _ ) = []
  | drop( l as h::t , i )
      = if i > 0 then drop( t , i-1 )
        else l ;
```


and let

$$\begin{aligned} \underline{\mathrm{mset}}(\,[\,]\,) &= \emptyset \\ \underline{\mathrm{mset}}(\,h::t\,) &= \langle h\rangle \uplus \underline{\mathrm{mset}}(t) \end{aligned}$$

Then, for all $\ell : \alpha\ \mathtt{list}$ and $n : \mathtt{int}$,

$$\boxed{\underline{\mathrm{mset}}(\mathtt{take}(\ell, n)) \uplus \underline{\mathrm{mset}}(\mathtt{drop}(\ell, n)) = \underline{\mathrm{mset}}(\ell)}$$

Structural induction on trees

Let P be a property on binary trees that we would like to prove. To establish

$$P(t) \text{ for all } t \text{ of type } \tau\ \mathtt{tree}$$

by *structural induction*, it suffices to prove.

1. The *base case*: $P(\mathtt{empty})$.
2. The *inductive step*: For all n of type τ and t_1, t_2 of type $\tau\ \mathtt{tree}$,

$$P(t_1) \text{ and } P(t_2) \text{ imply } P(\mathtt{node}(n, t_1, t_2))$$

Example: No tree equals its own left subtree.

For all n of type τ and all t_1, t_2 of type $\tau\ \mathtt{list}$,

$$\mathtt{node}(n, t_1, t_2) \neq t_1.$$

An application

```
fun treemap f empty = empty
  | treemap f ( node(n,l,r) )
      = node( f n , treemap f l , treemap f r ) ;
```

Functoriality of `treemap`.

$$\boxed{\mathtt{treemap\ id} = \mathtt{id}}$$

For all $f : \alpha \to \beta$ and $g : \beta \to \gamma$,

$$\boxed{\mathtt{treemap}(g \circ f) = \mathtt{treemap}(g) \circ \mathtt{treemap}(f) \quad : \alpha\ \mathtt{tree} \to \gamma\ \mathtt{tree}}$$

Structural induction on finitely-branching trees

```
datatype
  'a FBtree = node of 'a * 'a FBforest
and
  'a FBforest = empty | seq of 'a FBtree * 'a FBforest ;
```


Let P and Q be properties on finitely-branching trees and forests, respectively, that we would like to prove.

To establish

$$\begin{array}{ll} & P(t) \text{ for all } t \text{ of type } \tau\ \texttt{FBtree} \\ \text{and} & \\ & Q(F) \text{ for all } F \text{ of type } \tau\ \texttt{FBforest} \end{array}$$

by *structural induction*, it suffices to prove.

1. The *base case*: $Q(\texttt{empty})$.
2. The *inductive step*: For all n of type τ, t of type $\tau\ \texttt{FBtree}$, and F of type $\tau\ \texttt{FBforest}$,

$$\begin{array}{ll} & Q(F) \text{ implies } P(\texttt{node}(n,F)) \\ \text{and} & \\ & P(t) \text{ and } Q(F) \text{ imply } Q(\texttt{seq}(t,F)) \end{array}$$

An application

```
fun FBtreemap f ( node(n,F) )
      = node( f n , FBforestmap f F )
and FBforestmap f empty = empty
  | FBforestmap f ( seq(t,F) )
      = seq( FBtreemap f t , FBforestmap f F ) ;
```

Functoriality of `FBtreemap` and `FBforestmap`.

$$\boxed{\texttt{FBtreemap id} = \texttt{id} \qquad \texttt{FBforestmap id} = \texttt{id}}$$

For all $f : \alpha \to \beta$ and $g : \beta \to \gamma$,

$$\boxed{\begin{array}{rcl} \texttt{FBtreemap}(g \circ f) & = & \texttt{FBtreemap}(g) \circ \texttt{FBtreemap}(f) \\ \texttt{FBforestmap}(g \circ f) & = & \texttt{FBforestmap}(g) \circ \texttt{FBforestmap}(f) \end{array}}$$