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 $P(\ell)$ for all $\ell$ of type $\tau$ list
by structural induction, it suffices to prove.

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

Example: No list equals its own tail.
For all $h$ of type $\tau$ and all $t$ of type $\tau$ list, $h::t \neq t$.

Applications

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

fun len 1
  = let
    fun addlen( n , [] ) = n
    | addlen( n , h::t ) = addlen( n+1 , t )
    in
      addlen( 0 , 1 )
    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 $\ell$, $\ell_1$, and $\ell_2$,
1. $\text{nlen}(\ell_1@\ell_2) = \text{nlen}(\ell_1) + \text{nlen}(\ell_2)$.
2. $\text{revApp}(\ell_1,\ell_2) = \text{nrev}(\ell_1)@\ell_2$.
3. $\text{nrev}(\ell_1@\ell_2) = \text{nrev}(\ell_2)@\text{nrev}(\ell_1)$.
4. $\ell@[] = \ell$.
5. $\ell@(\ell_1@\ell_2) = (\ell@\ell_1)@\ell_2$.
6. $\text{nrev}(\text{nrev}(\ell)) = \ell$.
7. $\text{nlen}(\ell) = \text{len}(\ell)$. 
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 $h : \gamma \to \delta$,
  $h \circ (g \circ f) = (h \circ g) \circ f : \alpha \to \delta$
  ```

- `fun id x = x ;
  For all $f : \alpha \to \beta$, $foid = f = idof`

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 $m : S \to \mathbb{N}$.

Applications

| fun map f [] = [] |
| map $f(h::t) = (f h) :: map f t$ ; |
| 1. Functoriality$^a$ of map. |
| `map id = id` |
| For all $f : \alpha \to \beta$ and $g : \beta \to \gamma$, |
| `map(gof) = map(g) o map(f) : list -> list` |
| 2. For all $f : \alpha \to \beta$, and $\ell_1, \ell_2 : \alpha \text{ list}$, |
| `map f(\ell_1 @ \ell_2) = (map f \ell_1) @ (map f \ell_2) : \beta \text{ list}` |
| 3. For all $f : \alpha \to \beta$, |
| `(map f) o nrev = nrev o (map f) : \beta \text{ list}` |

$^a$This is a technical term from Category Theory.

- Some ways of forming multisets:

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

     $\langle \rangle : x \mapsto 0$

  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

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

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

```haskell
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

```haskell
mset( [] ) = ∅
mset( h::t ) = {h} ∪ mset(t)
```

Then, for all ℓ: α list and n : int,

```
\text{mset}(\text{take}(\ell,n)) \cup \text{mset}(\text{drop}(\ell,n)) = \text{mset}(\ell)
```

### Structural induction on trees

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

\[ P(t) \text{ for all } t \text{ of type } \tau \text{ tree} \]

by \textit{structural induction}, it suffices to prove.

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

\[ P(t_1) \text{ and } P(t_2) \text{ imply } P(\text{node}(n,t_1,t_2)) \]

\textbf{Example}: No tree equals its own left subtree.

For all \( n \) of type \( \tau \) and all \( t_1, t_2 \) of type \( \tau \text{ list} \),

\[ \text{node}(n,t_1,t_2) \neq t_1. \]

### An application

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

Functoriality of \textit{treemap}.

For all \( f : \alpha \rightarrow \beta \) and \( g : \beta \rightarrow \gamma \),

```
\text{treemap}(g \circ f) = \text{treemap}(g) \circ \text{treemap}(f) : \alpha \text{ tree} \rightarrow \gamma \text{ 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
\[
P(t) \text{ for all } t \text{ of type } \tau \text{ FBtree}
\]
and
\[
Q(F) \text{ for all } F \text{ of type } \tau \text{ FBforest}
\]
by structural induction, it suffices to prove.

1. The base case: \( Q(\text{empty}) \).
2. The inductive step: For all \( n \) of type \( \tau \), \( t \) of type \( \tau \text{ FBtree} \), and \( F \) of type \( \tau \text{ FBforest} \),
   \[
   Q(F) \text{ implies } P(\text{node}(n, F))
   \]
   and
   \[
   P(t) \text{ and } Q(F) \text{ imply } Q(\text{seq}(t, F))
   \]

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.

\[
\begin{align*}
\text{FBtreemap id} &= \text{id} \\
\text{FBforestmap id} &= \text{id}
\end{align*}
\]

For all \( f : \alpha \rightarrow \beta \) and \( g : \beta \rightarrow \gamma \),

\[
\begin{align*}
\text{FBtreemap}(g \circ f) &= \text{FBtreemap}(g) \circ \text{FBtreemap}(f) \\
\text{FBforestmap}(g \circ f) &= \text{FBforestmap}(g) \circ \text{FBforestmap}(f)
\end{align*}
\]