Verification using testing

- Code verification gives extra confidence when testing is not enough
  - Maybe not possible to test adequately
  - Or code needs high assurance
  - Proves presence of bugs, not their absence
Verification using proofs

• Goal: prove program works
--- Strategy: prove that each implementation satisfies its specification

--- Consider each module separately
• Assume other modules satisfy their specifications
• Works if no cycles in module dependency; otherwise may have to consider multiple modules at once

--- Key technique: induction
• Necessary for reason about recursive computations

lmax example

• Does the following implementation satisfy its specification?

(* lmax(lst) is the largest element
 * in lst. Requires: lst is non-nil. *)

fun lmax(lst: int list):int =
  case lst of
    [] => raise Fail "?"
  | [x] => x
  | h::t => Int.max(h, lmax(t))

Problem: Recursion leads to circular reasoning!
Proof by induction

Goal: prove some proposition is true for an infinite collection
- E.g., \( \text{lmax(1st)} \) is the max element for all non-empty lists \( \text{lst} \)

1. State the proposition as a condition \( P(n) \) that must be true for all \( n \geq n_o \) (usually \( n_o \) is 0 or 1)
- \( n \) is the length of the list \( \text{lst} \) \( (n \geq 1) \)
- \( P(n) \) is: \( \text{lmax(1st)} \) is the max elem for all lists \( \text{lst} \) of length \( n \)

2. Base case: show \( P(n_o) \)
- E.g., \( \text{lmax(1st)} \) is the max elem for all 1-elm lists \( \text{lst} \)

3. State induction hypothesis \( P(n) \)
- Assume \( \text{lmax(1st)} \) is the max elem for all lists \( \text{lst} \) of length \( n \)

4. Induction step: show \( P(n+1) \) assuming induction hypothesis
- Show: \( \text{lmax(1st)} \) is the max elem for all \((n+1)\)-elem lists \( \text{lst} \)

5. State conclusion: \( P(n) \) is true for all \( n \geq n_o \)

\[ P(1) \implies P(2) \implies P(3) \implies \ldots \implies P(n) \implies \ldots \]
“falling dominos”

\[ \begin{align*}
\text{lmax} & \text{ is the largest element in lst.} \\
& \text{* Requires: \( \text{lst} \) is non-nil. *}
\end{align*} \]

fun \( \text{lmax(lst: int lst): int =} \)
\[
\text{case lst of}
\]
\[
\begin{array}{lll}
| [] & => & \text{raise Fail “?”} \\
| [x] & => & x \\
| \_ : \_ & => & \text{Int.max(h, lmax(t))}
\end{array}
\]

1. State the proposition: for all \( n \geq 1 \), \( \text{lmax(1st)} \) is the max elem for all lists \( \text{lst} \) of length \( n \)

2. Base case: is \( \text{lmax(1st)} \) give max elem for all 1-elm lists \( \text{lst} \)?
- \( \text{lmax([v])} \) case \( [v] \) of ...

3. Induction hypothesis: \( \text{lmax(1st)} \) works for all \( \text{lst} \) of length \( n \)

4. Induction step: consider \( \text{lmax(1st)} \) where \( \text{lst} \) has length \( n+1 \)
- \( \text{lst} = [v_1, v_2, \ldots, v_{n+1}] \)
- \( \text{lmax([v_1, v_2, \ldots, v_{n+1}])} \) case \( [v_1, v_2, \ldots, v_{n+1}] \) of ...
- \( \text{Int.max(v_j, lmax([v_2, \ldots, v_{n+1}]))} \)
- \( \text{IH: lmax([v_2, \ldots, v_{n+1}])} \) evaluates to maximum of \( v_2, \ldots, v_{n+1} \)
- If \( v_j \geq \text{lmax([v_2, \ldots, v_{n+1}])} \), \( v_j \) is max of \( v_1, \ldots, v_{n+1} \)

5. Conclusion: \( \text{lmax} \) finds the max elem for all non-nil lists
Data abstraction

\textbf{Type} \texttt{set} = int list

(* AF: \([x_1, \ldots, x_n]\) represents \([x_1, \ldots, x_n]\) *)

(* RI: no duplicates or negative elements *)

\textbf{Fun} \texttt{union}(s1: set, s2: set)=

\texttt{foldl}(fn(x,s) \Rightarrow \text{if contains}(s,x) \text{ then } s \text{ else } x::s) \ s1 \ s2

**union** is correct if:

If: RI(s1) and RI(s2) hold,

Then: RI(union(s1, s2)) holds and

\AF(\text{union}(s1, s2)) = \AF(s1) \cup \AF(s2)

Correctness

- Given: \(s_1\) and \(s_2\) contain no negative elements or duplicates

- Show: RI(union(s1, s2)) and

  \AF(\text{union}(s_1, s_2)) = \AF(s_1) \cup \AF(s_2)

- \texttt{union}(s_1, s_2)

  \texttt{foldl} (fn(x,s) \Rightarrow \text{if contains}(s,x) \text{ then } s \text{ else } x::s) \ s_1 \ s_2

- Now we need to use induction!
Proof by induction

- **State proposition** in terms of \( P(n) \): for all \( n \geq 0 \), if \( \text{RI}(s_1) \) and \( \text{RI}(s_2) \) and \( s_2 \) has length \( n \), \( \text{foldl}(\ldots) \) \( s_1 \ s_2 \) evaluates to a list \( l \) such that
  \( \text{RI}(l) \) is true & \( \text{AF}(l) = \text{AF}(s_1) \cup \text{AF}(s_2) \)

- **Base case**: \( \text{foldl}(\ldots) \) \( s_1 \ [\] \) evaluates to \( l = s_1 \)
  \( \text{RI}(s_1) \) so \( \text{RI}(1) \), \( \text{AF}(s_1) \cup \text{AF}(\{\}) = \text{AF}(s_1) \cup \emptyset = \text{AF}(s_1) = \text{AF}(1) \)

- **Induction hypothesis**: assume \( P(n) \)

- **Induction step**: assume \( \text{RI}(s_1) \& \text{RI}(s_2) \) and \( s_2 = \{v_1, \ldots, v_{n+1}\} \)
  - Recall: \( \text{foldl} f \ b \ (h:\{t\}) = \text{foldl} f (f(h,b)) t \)
  - \( \text{foldl} (\ldots) \) \( s_1 \ [v_1, \ldots, v_{n+1}] \)
  - \( \text{foldl} (\ldots) (\ldots)(\ldots, s_1) \) \([v_1, \ldots, v_{n+1}] \)

Inductive step

\[
\text{fun union}(s_1: \text{set}, s_2: \text{set}) = \\
\text{foldl} (\text{fn}(x,s) \Rightarrow \text{if contains}(s,x) \text{ then } s \text{ else } x:s) s_1 s_2 \\
\text{Given: } s_1 \text{ and } s_2 \text{ contain no negative elements or duplicates} \\
\text{Show: } \text{RI}(\text{union}(s_1, s_2)) \& \text{AF}(\text{union}(s_1, s_2)) = \text{AF}(s_1) \cup \text{AF}(s_2)
\]

- **Induction hypothesis \( P(n) \):** if \( \text{RI}(s_1) \& \text{RI}(s_2) \) and \( s_2 \) has length \( n \),
  \( \text{foldl}(\ldots) \ s_1 \ s_2 \) evaluates to a list \( l \) such that:
  \( \text{RI}(l) \) is true & \( \text{AF}(l) = \text{AF}(s_1) \cup \text{AF}(s_2) \)

- **Induction step, show \( P(n+1) \)**
Inductive step

- Induction step, show P(n+1):
  assume $\text{RI}(s_1)$ & $\text{RI}(s_2)$ and $s_2 = [v_1, \ldots, v_n, \ldots]$.

  - foldl $\ldots$ $s_1$ $[u_1, \ldots, u_n, 1]$
  - foldl $\ldots$ $s_2$ $[(\ldots)(v, s_1)]$ $[v_2, \ldots, v_n, 1]$

  - foldl $\ldots$ (if contains(s_1, v) then $s_1$ else $v$) $\ldots$ $s_1'$

- Have $\text{RI}(s_1)$, so we can assume contains works

  - if contains(s_1, v) then $s_1$ else $v$ : : $s_1'$ where $\text{RI}(s_1')$ and $\text{AF}(s_1') = \text{AF}(s_1) \cup \{v\}$

- Now, can use induction hypothesis on foldl $\ldots$ $s_1'$ $[v_2, \ldots, v_n, 1]$

  - it evaluates to a list 1 such that $\text{RI}(1) & \text{AF}(1) = \text{AF}(s_1') \cup \text{AF}([v_2, \ldots, v_n, 1])$
  
  - $= \text{AF}(s_1) \cup \{v\} \cup [v_2, \ldots, v_n, 1]$
  
  - $= \text{AF}(s_1) \cup \text{AF}(s_2)$

- This 1 is the result of union(s1, s2) – we’re done!

Some thoughts

- We can really prove code works!
- Convincing proof requires knowing evaluation rules for language
- Almost any interesting code requires proof by induction
- Using recursive functions, loops correctly requires inductive reasoning – you have already (partly) internalized this process
- Reasoning explicitly avoids errors