CS312

Program Correctness Proofs

Lecture 12

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

3

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!

4

Proof by induction

Goal: prove some proposition is true for an infinite collection

- E.g., `lmax(lst)` is the max element for *all* non-empty lists `lst`

1. State the proposition as a condition $P(n)$ that must be true for all $n \geq n_0$ (usually n_0 is 0 or 1)
   - n is the length of the list `lst` ($n \geq 1$)
   - $P(n)$ is: `lmax(lst)` is the max elem for all lists `lst` of length n
2. Base case: show $P(n_0)$
   - E.g., `lmax(lst)` is the max elem for all 1-elem lists `lst`
3. State *induction hypothesis* $P(n)$
   - Assume `lmax(lst)` is the max elem for all lists `lst` of length n
4. Induction step: show $P(n+1)$ assuming induction hypothesis
   - Show: `lmax(lst)` is the max elem for all $(n+1)$-elem lists `lst`
5. State conclusion: $P(n)$ is true for all $n \geq n_0$

$$P(1) \Rightarrow P(2) \Rightarrow P(3) \Rightarrow \ldots \Rightarrow P(n) \Rightarrow \ldots$$

"falling dominoes"

5

lmax

```
(* lmax(lst) is the largest element in lst.
 * Requires: lst is non-nil. *)
fun lmax(lst: int lst):int =
  case lst of
    [] => raise Fail "?"
  | [x] => x
  | h::t => Int.max(h, lmax(t))
```

1. State the proposition: for all $n \geq 1$, `lmax(lst)` is the max elem for all lists `lst` of length n
2. Base case: is `lmax(lst)` give max elem for all 1-elem lists `lst?`
   - `lmax([`v`])` à `case [`v`] of …` à v
3. Induction hypothesis: `lmax(lst)` works for all `lst` of length n
4. Induction step: consider `lmax(lst)` where `lst` has length $n+1$
   - `lst` = `[`$v_1,v_2,\ldots v_{n+1}$`]`
   - `lmax([`$v_1,v_2,\ldots v_{n+1}$`])`à `case ([`$v_1,v_2,\ldots v_{n+1}$`] of …`
     à `Int.max(`v_1`, lmax([`$v_2,\ldots v_{n+1}$`]))`
   - IH: `lmax([`$v_2,\ldots v_{n+1}$`]))` evaluates to maximum of $v_2,\ldots,v_{n+1}$
   - If $v_1 \geq$ `lmax([`$v_2,\ldots v_{n+1}$`])`, v_1 is max of $v_1,\ldots,v_{n+1}$
5. Conclusion: `lmax` finds the max elem for all non-nil lists

6

Data abstraction

```
type set = int list
(* AF: [x1,…,xn] represents {x1,…,xn} *)
(* RI: no duplicates or negative elements *)
fun union(s1: set, s2: set)=
 foldl(fn(x,s) => if contains(s,x) then s else x::s) s1 s2
```

union is correct if:

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

Then: RI(**union(s1,s2)**) holds and
AF(**union(s1,s2)**) = AF(**s1**) ∪ AF(**s2**)

$$\begin{tikzcd} \bullet \arrow[r, "\cup"] & \bullet \;\; \mathrm{AF}(\mathtt{s1}) \cup \mathrm{AF}(\mathtt{s2}) \\ \mathtt{s1,s2} \; \bullet \arrow[u, "\mathrm{AF}"] \arrow[r, "\mathtt{union}"'] & \bullet \arrow[u, "\mathrm{AF}"'] \end{tikzcd}$$

Correctness

- Given: $\mathbf{s_1}$ and $\mathbf{s_2}$ contain no negative elements or duplicates
- Show: RI(**union**($\mathbf{s_1}$,$\mathbf{s_2}$)) and
  AF(**union**($\mathbf{s_1}$,$\mathbf{s_2}$)) = AF($\mathbf{s_1}$) ∪ AF($\mathbf{s_2}$)
- union(s_1,s_2) à

```
foldl (fn(x,s) =>
        if contains(s,x) then s else x::s)
        s1 s2
```

- Now we need to use induction!

Proof by induction

- State proposition in terms of P(n): for all $n \geq 0$, if RI(**s1**) and RI(**s2**) and **s2** has length n, `foldl(…) s1 s2` evaluates to a list **l** such that
  RI(**l**) is true & AF(**l**) = AF(**s1**) ∪ AF(**s2**)

- Base case: `foldl(…) s1 []` evaluates to `l=s1`
  RI(**s1**) so RI(**l**) , AF(**s1**) ∪ AF(**[]**) = AF(**s1**) ∪ ∅ = AF(**s1**) = AF(**l**)

- Induction hypothesis: assume P(n)

- Induction step: assume RI(**s1**) & RI(**s2**) and **s2=**[$v_1, \ldots, v_{n+1}$]
  - Recall: `foldl f b (h::t) à foldl f (f(h,b)) t`
  - `foldl (…) s1` [$v_1, \ldots, v_{n+1}$]
    `à foldl (…) ((…)(v1,s1))` [$v_1, \ldots, v_{n+1}$]

9

Inductive step

```
fun union(s1: set, s2: set)=
  foldl (fn(x,s) => if contains(s,x) then s else x::s) s1 s2
```

- Given: $\mathbf{s_1}$ and $\mathbf{s_2}$ contain no negative elements or duplicates
- Show: RI(**union**($\mathbf{s_1}$,$\mathbf{s_2}$)) & AF(**union**($\mathbf{s_1}$,$\mathbf{s_2}$)) = AF($\mathbf{s_1}$) ∪ AF($\mathbf{s_2}$)

- Induction hypothesis P(n): if RI(**s1**) & RI(**s2**) and **s2** has length n, `foldl(…) s1 s2` evaluates to a list **l** such that:

  RI(**l**) is true & AF(**l**) = AF(**s1**) ∪ AF(**s2**)

- Induction step, show P(n+1)

10

Inductive step

- Induction step, show $P(n+1)$:

  assume RI(`s1`) & RI(`s2`) and `s2`=[v_1, …, v_{n+1}]
  - `foldl (…) s1` [v_1,…,v_{n+1}]
    à `foldl (…) ((…)(`v_1`, s1))` [v_2,…,v_{n+1}]
    à `foldl (…) (if contains(s1,`v_1`) then s1 else `v_1`::s1))` [v_2,…,v_{n+1}]
  - Have RI(`s1`) , so we can assume **`contains`** works
    `if contains(s1,`v_1`) then s1 else `v_1`::s1` à `s1'` where RI(`s1'`) and AF(`s1'`) = AF(`s1`) $\cup$ {v_1}
  - Now, can use induction hypothesis on `foldl (…) s1'`[v_2,…,v_{n+1}] – it evaluates to a list `l` such that
    RI(`l`) & AF(`l`) = AF(`s1'`) $\cup$ AF([v_2,…,v_{n+1}])
    = AF(`s1`) $\cup$ {v_1} $\cup$ { v_2,…,v_{n+1}}
    = AF(`s1`) $\cup$ AF(`s2`)
  - This `l` is the result of **`union(s1,s2)`** – we're done!

11

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

12