CSci 2041
Advanced Programming Principles
L6: Reasoning About Correctness, Part 2

Prof. Eric Van Wyk

Fall 2014

Instead of those informal properties (from L5 slides), consider this property:

```
reverse (l1 @ l2) = reverse l2 @ reverse l1
```

for our original definition of `reverse`

```
let rec reverse l = match l with
  | [ ] -> [ ]
  | x::xs -> reverse xs @ [x]
```

Reasoning with cases

Consider ordered lists:

```
let rec place e l = match l with
  | [ ] -> [e]
  | x::xs -> if e < x then e::x::xs
             else x :: (place e xs)

let rec is_elem e l = match l with
  | [ ] -> false
  | x::xs -> e = x || (e > x && is_elem e xs)

let rec sorted l = match l with
  | [ ] -> true
  | x::[] -> true
  | x1::x2::xs -> x1 <= x2 && sorted (x2::xs)
```

(in ordered_list.ml)

Can we show that:

- `is_elem e (place e l)`
- `sorted l` $\Rightarrow$ `sorted (place e l)`

Trees

Similar approaches to reasoning can be applied to many inductive types and functions over them.
Consider our `btree`.

```
type 'a btree = Empty
              | Node of ('a * 'a btree * 'a btree)
```

What is the principle of induction:

$\forall t, P(t)$ *holds if* $P($`Empty`$)$
$P($`Node (v,t1,t2)`$)$ holds when $P($`t1`$)$ and $P($`t2`$)$

Consider our `sum` function

```
let rec sum_btree t =
  match t with
  | Empty -> 0
  | Node (a,l,r) -> a + sum_btree l + sum_btree r
```

What can we prove about this?

Ordered trees

Consider a function to add elements to a ordered tree:

```
let rec insert t i = match t with
  | Empty -> Node (i, Empty, Empty)
  | Node (v, t1, t2)
      -> if v <= i
         then Node (v, t1, insert t2 i)
         else Node (v, insert t1 i, t2)
```

What might we prove here?

```
let rec flatten t = match t with
  | Empty -> [ ]
  | Tip(v) -> [v]
  | Node(v,t1,t2) -> flatten t1 @ flatten t2

let sort l = flatten (tree_from_list l)

let test = list_ordered (sort l)
```

We could prove `test` for any list.
(See `ordered_btree.ml`)

Other proofs

```
let rec euclid m n =
  if m = n then m
  else
    if m < n
    then euclid m (n-m)
    else euclid (m-n) n
```

Our specifications

$$
\begin{array}{lcll}
gcd\ m\ n & = & gcd\ m\ (n-m) & \text{if } n > m \\
gcd\ m\ n & = & gcd\ (m-n)\ n & \text{if } m > n \\
gcd\ m\ n & = & m & \text{if } m = n
\end{array}
$$

Does induction over natural numbers work here?

Induction over the recursive computation

Consider some invariant P for a function.

For example `euclid m n`$=$ *gcd m n*.

Induction over the computation:

Base case: show the property holds for non-recursive calls.

Inductive case: show it holds for other calls, assuming that the invariant holds on the recursive call.

Reasoning about imperative programs

Recall our `euclid` function.

```
let rec euclid m n =
  if m = n then m
  else
    if m < n
    then euclid m (n-m)
    else euclid (m-n) n
```

How might we have written this in an imperative language?

```
x = m
y = n


while x <> y {
  if x > y
    x = x - y
  else
    y = y - x
}
(* Answer stored in x *)
```

How can we reason about this program?
How can we prove it is correct?

- Must we consider state? We no longer have referential transparency.
- Or just consider loop invariants.
- These invariants must hold
  - before the loop,
  - after each time through it,
  - and the loop

```
(* Pre condidtion: m > 0, n > 0 *)
x = m
y = n

(* Loop invariant: gcd m n = gcd x y *)
while x <> y {
  if x > y
    x = x - y
  else
    y = y - x
}
(* Answer stored in x *)
(* Post condidtion: gcd m n = x *)
```

How can we reason about this program?
How can we prove it is correct?

We use invariants similar to those from reasoning about functional programs.

For `euclid` we had: `euclid m n` $= gcd\ m\ n$

In the imperative code we have `x` $= gcd\ m\ n$

Show:

1. The precondition implies the loop invariant holds before entering the loop.
2. If it holds before the loop, it holds after an iteration of the loop.
3. After the loop the negation of the condition and the loop invariant (which we know will be true) implies the post condition.

Argue, informally, each case:

1. ... pretty easy ...
2. ... two cases to consider ...
3. ... now x = y, so ...

Our proofs of properties or invariants of functions, for example

- `is_elem e (place e l)` or
- `euclid m n` $= gcd\ m\ n$

are formal. Every step is justified.

But these ideas carry over to imperative programming as well.

Granted, we do these less formally, but they are still rigorous.

Designing correct functional programs

A past homework problem: `approx_squareroot`
Write the approximate square root function so that it maintains the invariant that

```
l * l <= x * x <= u *u
```

$\Rightarrow$

```
let (l',u') = approx_squareroot x l u
in l' * l' <= x * x <= 'u * u'
```

This says nothing about how progress is made, or when the function terminates.
Assume, as before, that it returns the inputs `l` and `u` when their difference is less that some value `accuracy`.

Designing correct imperative programs

Consider Jon Bentley's Programming Pearls paper "Writing Correct Programs"

It describes using invariants to develop correct imperative programs.