CSci 2041 Advanced Programming Principles L5: Reasoning About Correctness

Prof. Eric Van Wyk

Fall 2014

Reasoning about program correctness

How do we know our programs are correct?

- We can reason formally - proving the correctness of a function.
- After understanding this formal approach, we can see how to reason informally, but rigorously, about correctness.
- We may not choose to formally prove program correctness, but we can at least reason clearly about them.
- This may also lead to a way of thinking about programming so that the specification of correctness is used to design correct programs from the beginning.

Induction on natural numbers

- You should be familiar with proving properties of natural numbers using inductions.
- For example, you may have proven that $0 + 1 + 2 ... + n = (n(n + 1))/2$.
- The principle of induction for natural numbers is:

$$\forall n, P(n) \text{ if } P(0) \text{ and } P(n-1) \implies P(n)$$

  This leads to proofs where

  1. we prove the base case $P(0)$.
  2. then prove the inductive case $P(n)$, assuming that $P(n-1)$ holds.

  Recall how you would prove the example above.

A comment on notation here: our notation $0 + 1 + 2 + ... + n$ means add all numbers from 0 to n. Thus,

- $0 + 1 + 2 + ... + 5 = 15$

- $0 + 1 + 2 + ... + 1 = 0 + 1$, not, say 4

Were are plugging in a value to this odd looking expression:

- $0 + 1 + 2 + ... + n$

You can think of the first part

$$0 + 1 + 2 + ... +$$

as the (odd looking) name of the operation.

Reasoning about functional programs over natural numbers

Consider the following function:

```
let rec sumTo n = match n with
  | 0 -> 0
  | n -> n + sumTo (n-1)
```

- We would like to show that $\forall n : \texttt{sumTo}\ n = 0 + 1 + 2 + ... + n$.
- That is, our property P is defined as $P(n)$ is $\texttt{sumTo}\ n = 0 + 1 + 2 + ... + n$
- Our inductive proof would have two cases:
  - P(0): show that $\texttt{sumTo}\ 0 = 0$
  - P(n): show that $\texttt{sumTo}\ n = 0 + 1 + 2 + ... + n$ using the fact that $\texttt{sumTo (n-1)} = 0 + 1 + ... + (n - 1)$

Generalizing induction to structured types

- Induction over natural numbers is just a special case of a more general form of induction over structured values.
- We can define a structured type for natural numbers as follows:

  ```
  type nat = Zero | Succ of nat
  ```

- The principle of induction for type `nat` is:

  $\forall n,\ P(n)$ if $P(\texttt{Zero})$ and $P(\texttt{Succ n})$ if $P(n)$

Some functions over nat

- conversion to Ocaml int values:

```
let toInt = function
  Zero -> 0
  Succ n -> toInt n + 1
```

- addition over nat types:

```
let rec add n1 n2 = match (n1, n2) with
  | Zero, n -> n
  | Succ n1', n -> Succ (add n1' n)
```

We would like to show that

- $\forall$ `n1, n2` $\in$ `nat`,
  `toInt (add n1 n2) = toInt n1 + toInt n2`.
- Thus, our property P`(n1,n2)` is
  `toInt (add n1 n2) = toInt n1 + toInt n2`
- (We could define P as a property over just one `nat` as follows:

  ```
  P(n1) is
     ∀ n2 ∈ nat:
        toInt (add n1 n2) = toInt n1 + toInt n2
  ```

  But this isn't necessary for our needs.
  It is equivalent to P`(n1,n2)`.)

P`(n1,n2)` is `toInt (add n1 n2) = toInt n1 + toInt n2`

- Our proof proceeds by induction over first argument to of *P* (which is the first argument to `add`).
- We need to show
  - P(Zero,n2), that is:
    `toInt (add Zero n2) = toInt Zero + toInt n2`
  - P(Succ n, n2), that is:
    ```
    toInt (add (Succ n) n2) =
     toInt (Succ n) + toInt n2
    ```
    given: `toInt (add n n2) = toInt n + toInt n2`

Follow the proof on the whiteboard.

- Have we covered all of the cases? How do we know?
  - Yes we have.
  - The only ways to create values of type `nat` is by the two constructor cases. The base case and the inductive case.
- Recall the principle of induction for type `nat`:
  $\forall n, P(n)$ if $P(\texttt{Zero})$ and $P(\texttt{Succ n})$ if $P(n)$
- Induction over natural numbers works because of the "less than" ordering over them.
- A similar "less than" ordering exists for our type `nat`.
  It is "component of".
  Instead of $2 < 3$ for natural numbers, we have
  `Succ (Succ Zero))` is a component of
  `Succ (Succ (Succ Zero)))`.

Every "inductive type" comes with its own principle of induction.
For a type, say,

```
type t = C1 of base
       | C2 of base * t
       | C3 of t * t
```

where `base` is a base type like `int`, the principle of induction is

$\forall t, P(t)$ if $P(\texttt{C1 v})$ and
$P(\texttt{C2 (v, t2)})$ if $P(\texttt{t1})$ and
$P(\texttt{C3 (t1, t2)})$ if $P(\texttt{t1})$, $P(\texttt{t2})$

(for any `v`, `t1`, and `t2`)

Disjoint unions, when they are recursive, are also commonly called inductive types.

This stems from their natural use in inductive proofs.

Another function over `nat` numbers

```
let rec sumTo n = match n with
  | Zero -> Zero
  | Succ n' -> add n (sumTo n')
```

- `P(n):`
  `toInt(sumTo n) = 0 + 1 + 2 + ...  + (toInt n)`
- How can we prove this?

Proving sumTo

```
P(n):
toInt(sumTo n) = 0 + 1 + 2 + ...  + (toInt n)
```

Our two cases:

`P(Zero)`:

show: `toInt (sumTo Zero) =`
```
      0 + 1 + 2 ...  + (toInt Zero)
```

`P(Succ n’)`:

show: `toInt (sumTo (Succ n’) =`
```
      0 + 1 + 2 + ...  + (toInt (Succ n’))
```

given: `toInt (sumTo n’) = 0 + 1 + ...  + (toInt n’)`

Referential Transparency

- In our reasoning process we replace (sub) expressions with other expressions that have the same value.
- This is correct because expression evaluation does not change the value of variables.
- With side effects (assignment statements) this is not true. *e.g.* , the evaluation of `add Zero n` produces `n` without changing values of any variables.
- This is called referential transparency.
- It is fundamental to reasoning about expressions in functional programs.

Lists and trees

- Admittedly, dealing with natural numbers in this way is not very convenient.
- It is done to relate inductive proofs you may have done before to those over structured types.
- Let's move on to lists and trees.

Lists

The natural questions to access are

- What is the type?
- What is its principle of induction?

```
type 'a list = [] | :: of 'a * 'a list
```

$\forall \ell, P(l)$ if $P(\texttt{[ ]})$ and
$P(\texttt{v ::}\ \ \ell'))$, if $P(\ell')$

Formal and informal arguments

- Our previous proofs rather formal, we had very little, if any, informal reasoning.
- In some proofs about programs over lists it is convenient to not prove formal properties about what "reverse of a list" means or what "sum up values in a list" means.
- We will give some justifications for steps in our proofs as informal, but hopefully rather obvious, properties of our data.
- We could prove these formally, but that goes beyound our intentions here.
- Our informal parts will be placed in quotations to make them visually distinct from our formal arguments.

Sum over a list

Our intuitive understanding of adding up the elements in a list can be expressesd as follows:

- "sum of an empty list is 0"
- "sum of a list x::xs is x + the sum of the list xs"

This is naturally seen in an implementation of `sum`.

```
let rec sum l = match l with \\
  | [] -> 0  \\
  | x:xs -> x + sum xs
```

And even more directly here:

```
let rec sum = function \\
  | [] -> 0  \\
  | x:xs -> x + sum xs
```

If we reason informally, we see that the understanding of sum lines up directly with the implementation.

Let's walk through a more formal reasoning to understand the process:

```
let rec sum l = match l with \\
  | [] -> 0  \\
  | x:xs -> x + sum xs\\
```

Our property uses some informal reasoning:

- $P(\ell)$: `sum` $\ell =$ "sum of elements in ℓ"

Before beginning any proof, we need to answer the following questions: What are the cases we need to prove?
What is the induction hypothesis?

- P([]): `sum []` = "sum of elements in [ ]"
- P(x::xs): `sum (x::xs)` = "sum of elements in x::xs"
  Given:
  - `sum xs` = "sum of elements in xs"

(Follow proof on the whiteboard.)

How about a property about `sum` and list append?
P(`l1, l2`): `sum (l1 @ l2) = sum l1 + sum l2`

What do we need to specify before we begin the proof?

What is the proof?

Follow this on the whiteboard.

List reverse

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

P(lst): `reverse lst` = "reverse of the list lst"

We won't bother to formalize what reverse means, but will simply put in quotations the informal parts of our arguments to make them distinct, visually, from the formal bits.

How would this proof proceed?

List reverse with accumulators

```
let rec rev l r' = match l with
  | [] r' -> r'
  | (x::xs) -> rev xs (x::r')

let reverse l = rev l []
```

P1(lst): `rev lst acc` = "reverse of the list lst" `@ acc`
P2(lst): `reverse lst` = "reverse of the list lst"

How would this proof proceed?

The important point here is the use of two properties, one for each function.