S4: Reasoning About Correctness

CSci 2041:

Advanced Programming Principles

University of Minnesota,
Prof. Van Wyk,
Spring 2022

Reasoning about correctness

Software correctness is a goal we discussed at the very beginning of class.

We have certain principles support that this goal:

- strong static type systems
- testing
- formal reasoning for proving correctness properties, the topic of these notes

From the S3: Testing slides:

> "Thus we might write correctness specifications as properties that, if they do not specify everything we mean by "correct", at least give us something concrete to reason about."

Here, we pick up on this phrase "reason about."

We ask: How can we formally reason about programs?

(Hint: it is all about induction and is the reason we called some data types "inductive" types.)

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 induction.
- For example, you may have proven that $0 + 1 + 2... + n = (n(n + 1))/2$.

  That is:

$$P(n) : \sum_{i=0}^{n} i = (n(n + 1))/2$$

- The principle of induction for natural numbers is:

$$\forall n \in \mathbb{N}.P(n) \text{ if } P(0) \text{ and } \forall m \in \mathbb{N}.P(m) \implies P(m + 1)$$

  This leads to proofs where
  1. we prove the base case $P(0)$.
  2. then prove the inductive case $P(n + 1)$, assuming that $P(n)$ holds.

  Recall how you would prove the example above.

  Notes with proofs are in the `Notes` directory of the public repo.

Reasoning about functions over natural numbers

Consider:

```
let rec sumTo (x: int) : int = match x with
  | 0 -> 0
  | x -> x + sumTo (x-1)
```

- We would like to show that $\forall n.$ `sumTo` $n = \sum_{i=0}^{n} i$.
- That is, our property P is defined as $P(n)$ is `sumTo` $n = \sum_{i=0}^{n} i$
- Our inductive proof would have two cases:
  - $P(0)$: show that `sumTo` $0 = \sum_{i=0}^{0} i$
  - $P(n)$: show:

    `sumTo` $(n + 1) = \sum_{i=0}^{n+1} i$

    given: `sumTo` `n` $= \sum_{i=0}^{n} i$

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:

  $\forall n \in \texttt{nat}\ .\ P(n)$ if $P(\texttt{Zero})$ and $P(n) \Longrightarrow P(\texttt{Succ}\ n)$

  This is our recipe for writing inductive proofs of $\forall n \in \texttt{nat}\ .\ P(n)$.
- These inductive types all have a `reduce` function that can be generated from the type.

  They also all have a principle of induction that can be generated from the type.

Some functions over `nat`

- conversion to Ocaml `int` values:

```
let rec 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 n', n -> Succ (add n' n)
```


We would like to show that ...

$\forall n \in \texttt{nat}\ .\forall n' \in \texttt{nat}\ .\ \texttt{toInt (add}\ n\ n'\texttt{) = toInt}\ n\ \texttt{+ toInt}\ n'.$

- Thus, our property $P(n)$ is
  $\forall n' \in \texttt{nat}\ .\ \texttt{toInt (add}\ n\ n'\texttt{)} = \texttt{toInt}\ n + \texttt{toInt}\ n'$
- We will often drop the $\forall\ n$ and just assume any (free) variables are universally quantified.

We need to show:

- $P(\texttt{Zero})$: $\forall\ n' \in \texttt{nat}$.
  $\texttt{toInt (add Zero}\ n'\texttt{)} = \texttt{toInt Zero} + \texttt{toInt}\ n'$
- $P(\texttt{Succ n})$: $\forall\ n' \in \texttt{nat}$.
  $\texttt{toInt (add (Succ n)}\ n'\texttt{)} = \texttt{toInt (Succ n)} + \texttt{toInt}\ n'$
  given: $\forall\ n' \in \texttt{nat}$.
  $\texttt{toInt (add n}\ n'\texttt{)} = \texttt{toInt n} + \texttt{toInt}\ n'$

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`:
$\forall n \in \texttt{nat}\ .\ P(n)$ if $P(\texttt{Zero})$ and $P(n) \implies P(\texttt{Succ}\ n)$

Induction over natural numbers works because of the "less than" ordering over them.

We know the only natural numbers are the ones constructed from 0 and adding 1 to another natural number.

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)))`.

We know the only values of type `nat` are those created by the constructors of this type.

- `Zero`
- `Succ`

Strong and Weak Induction

Weak Induction:

- To show $\forall n \in \mathbb{N}.P(n)$ we show
  - $P(0)$ and
  - $P(n) \implies P(n+1)$

Strong Induction:

- To show $\forall n \in \mathbb{N}.P(n)$ we show
  - $(\forall m < n.P(m)) \implies P(n)$

Principles of induction

Every "inductive type" has a principle of induction.
For a type, say,

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

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

$\forall t \in$ `ty`, $P(t)$ if
- $P($`C1` $v)$ holds,
- $P(t') \implies P($`C2` $(v, t'))$ holds, and
- $P(t')$ and $P(t'') \implies P($`C3` $(t', t''))$ holds.

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.

Rewrite `sumTo` to take `nat` numbers

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

- P(n): `toInt(sumTo'` n`)` = $\sum_{i=0}^{\texttt{toInt}\ n} i$
- How can we prove this?

Exercise S4: #1:

P(n):
toInt(sumTo' n) = $\sum_{i=0}^{\texttt{toInt } n} i$

1. What is the base case that we need to prove?
   That is, what do we need to show?
2. What is the inductive case that we need to prove?
   That is, what do we need to show? what is given to us?

```
let rec toInt n = match n with
  | Zero -> 0
  | Succ n -> toInt n + 1
let rec sumTo' match n with
  | Zero -> Zero
  | Succ n' -> add n (sumTo' n')
let rec add n1 n2 = match n1, n2 with
  | Zero, n -> n
  | (Succ n', n -> Succ (add n' n)
```


Proving this property about sumTo

P(n):
toInt(sumTo' n) = $\sum_{i=0}^{\texttt{toInt } n} i$

Our two cases:
1. P(Zero):

show: toInt (sumTo' Zero) = $\sum_{i=0}^{\texttt{toInt Zero}} i$

2. P(Succ n):

show: toInt (sumTo' (Succ n) = $\sum_{i=0}^{\texttt{toInt (Succ n)}} i$

given: toInt (sumTo' n) = $\sum_{i=0}^{\texttt{toInt n}} i$

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 ask are

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

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

$\forall \ell, P(\ell)$ holds if

$P(\texttt{[ ]})$ holds and

$P(\ell') \Longrightarrow \forall v,\ P(v\ \texttt{::}\ \ \ell')$

Sum of a list

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

We may instead consider general properties relating functions and operations.

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 files in `Notes` in the public repository on GitHub.

Exercise S4: #2: List reverse

Consider this property:
`reverse (l1 @ l2) = reverse l2 @ reverse l1`

for our definition of `reverse`

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

Can we prove this?

Look for solution in `Notes`.

Consider ordered lists: (in `ordered_list.ml`)

```
let rec place e l = match l with
  | [ ] -> [e]
  | x::xs when e < x -> e::x::xs
  | x::xs -> 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)
```


Can we show that:

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

Proofs of these and comments about those proofs will be in a "notes" file in the `Notes` directory in the GitHub public repository.

Trees

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

```
type 'a tree = Empty
             | Node of 'a tree * 'a * 'a tree
```

What is the principle of induction:

$\forall t \in$ `'a tree`, $P(t)$ holds if
- $P($`Empty`$)$ and
- $P(t_1)$ and $P(t_2) \Longrightarrow P($`Node (`$t_1$`,`$v$`,`$t_2$`)`$)$

Binary search trees

```
let rec insert (e: 'a) (t: 'a tree) : 'a tree = ...

let rec to_list (t: 'a tree) : 'a list = ...

let rec to_tree (l: 'a list) : 'a tree = ...

let sort l = to_list (to_tree l)
```

We could prove `sorted (sort l)` for any list `l`.

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?

Exercise S4: #3: Write `euclid` as an imperative program.

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

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

```
(* 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 *)
```


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 ...

Exercise S4: #4:

```
(* Pre condition: x >= 0 *)
p = 0;
i = 0;
while i < x {
   p = p + y;
   i = i + 1;
}
(* Post condition: p = x * y *)
```

What is an appropriate loop invariant?

Argue that the post condition holds if the pre condition does.

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.

If you're interested, see "An Axiomatic Basis for Computer Programming" by Tony Hoare in the `Resources/Books-Papers` directory.

Prove statements of this form: $\{\ P\ \}$ `code` $\{\ Q\ \}$

Designing correct functional programs

Consider the following algorithms written in psuedo-code similar to C. Assume that the "input" value `n` is greater than 1.0 and all variables hold real numbers.

```
lower = 1.0;
uppper = n;
accuracy = 0.001;
while ( (upper - lower) > accuracy ) {
  guess = (lower + uppper) / 2.0;
  if ( (guess*guess) > n)
     upper = guess;
  else
     lower = guess;
}
```


Write the approximate square root function so that it maintains the invariant that
`lower * lower <= n <= upper * upper`
⇒
```
let (lower',upper') = sqrt_step n lower upper
in lower' * lower' <= n <= 'upper * upper'
```

This says nothing about how progress is made, or when the function terminates.

Assume that `sqr_step` returns the inputs `lower` and `upper` when their difference is less that some value `accuracy`.

We'll develop this solution in `approx_sqrt.ml` in the `SamplePrograms` directories in the GitHub public repo.

Designing correct imperative programs

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

It describes using invariants to develop correct imperative programs.

This paper is in the `Resources` directory of the GitHub public repo.