CSci 2041: Advanced Programming Principles
Reasoning About Program Behaviour

Gopalan Nadathur

Department of Computer Science and Engineering
University of Minnesota

Lectures in Spring 2015

What and Why?

We want to understand two related aspects about our programs

- How do we show that our functions do what they are supposed to do?
- What can we say about their interactions with other functions?

Some reasons why we should be interested in this topic

- such analysis helps us understand our programs better
- thinking this way makes us design better programs
- being able to say something formally about our programs should give our users more confidence in them

The Difficulty in Reasoning About Programs

Functions are typically written to work over an *infinite* collection of inputs

For example, consider the function to append two lists

```
let rec append l1 l2 =
   match l1 with
   | [] -> l2
   | (h::t) -> h :: (append t l2)
```

Showing that `append` does the right thing for particular inputs is easy; you just "run it"

What we want to show, though, is that `append` will work for *any* given inputs

The standard way to provide a *finite* proof for all inputs for a recursive program is to use *induction*

The Induction Principle for Natural Numbers

Let $P(n)$ be a property over natural numbers that you want to show holds of every natural number

The induction principle says that if you can show

- $P(0)$ holds, and
- for every n if $P(n)$ holds then $P(n+1)$ holds

then it must be the case that $P(n)$ holds for *every* n

As an example, consider showing the following

$$0 + 1 + \ldots + n = (n * (n + 1))/2$$

The key idea: to show a property for an infinite collection you provide a *method* rather than showing each case separately

Partial versus Total Correctness

Consider the factorial program we saw earlier

```
(* fact : int -> int
   precondition : input n >= 0
   invariant : fact n evaluates to n! *)
let rec fact n =
   if (n = 0) then 1 else n * fact (n - 1)
```

Given the specification, there are two different properties that we might want to prove

- for all $n \geq 0$ (*fact n*) evaluates to $n!$
- for all $n \geq 0$, if (*fact n*) evaluates to m, then $m = n!$

The first kind of property is called *total correctness* and the second is called *partial correctness*

We will typically focus on total correctness

Reasoning Inductively About Factorial

First observe that the precondition ensures that the input to `fact` is a natural number

Given this, what we really want to prove is the following property for every natural number n:

($fact\ n$) *evaluates to* $n!$

Thus, using the induction principle, here is what we need to show

- ($fact\ 0$) evaluates to $0!$
- If $fact\ n$ evaluates to $n!$ then ($fact\ (n+1)$) evaluates to $(n+1)!$

It is easy to prove these properties based on the definition of factorial

Reasoning About the Tail-Recursive Factorial

Let us now consider the more efficient definition of factorial

```
(* tr_fact : int -> int
   precondition : input n >= 0
   invariant : tr_fact n evaluates to n! *)
let tr_fact n =
   let rec fact_helper n a =
     if (n = 0) then a
     else fact_helper (n-1) (n * a)
   in fact_helper n 1
```

The property we want to prove for this function is unchanged

for all natural numbers n, (*tr_fact n*) *evaluates to n*!

However, to prove this property, we will need a *lemma* about *fact_helper* and it is that that will be proved inductively

Proving the Tail-Recursive Factorial Correct (Contd)

The key is to articulate an invariant about *fact_helper* such that

- we can prove what we want about *tr_fact* based on it, and
- we can also prove the property about *fact_helper* using induction

A property that works from both perspectives

> *for all natural numbers n and a,* (*fact_helper n a*) *evaluates to* $n! * a$

The property of a natural number *n* we need to prove here:

> *for all a,* (*fact_helper n a*) *evaluates to* $n! * a$

Note that this property itself has a universal quantifier over *a*

Lets check out both that we can prove this property using induction and that it yields a correctness proof for *tr_fact*

Proving the Tail-Recursive Factorial Correct (Contd)

We use natural number induction on n to show for each n that

for all a, (*fact_helper n a*) *evaluates to* $n! * a$

Base Case: (Show property when n = 0)

Looking at the code, we see that, for any a, (*fact_helper* 0 a) evaluates to a, i.e. to $0! * a$

Induction Step: (Assume property for n, show for $n + 1$)

We need to show for an arbitrary a that

(*fact_helper* $(n + 1)$ a) *evaluates to* $(n + 1)! * a$

By specializing the property assumed for n, we have that

(*fact_helper n* $((n + 1) * a)$ *evaluates to* $n! * (n + 1) * a$

But $n! * (n + 1) * a = (n + 1)! * a$ and (*fact_helper* $(n + 1)$ a) evaluates to whatever (*fact_helper n* $((n + 1) * a)$) evaluates to

In other words, we have the desired conclusion

Strong Induction for Natural Numbers

Sometimes it is more convenient to work with an induction principle that on the surface looks stronger

This principle says that to show $P(n)$ holds for every natural number n it suffices to show that

- $P(0)$ holds and
- If $P(m)$ holds for every $m < n$ then $P(n)$ holds

Note that the second condition in fact includes the first so we don't need to state it explicitly

This principle is often referred to as *strong induction*

In fact, it is derivable from the ordinary induction principle and is only a rephrasing of it that is more convenient in some cases

Expressing Correctness Properties

In proving correctness of function definitions, we need to articulate a standard that we can call the "mathematical notion"

For example, for the factorial function, we proved correctness relative to a mathematical definition of factorial

Similarly, we will consider a fibonnaci function in OCaml that will be expected to implement the following mathematical definition

$$math\_fib(n) = \begin{cases} 1 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ math\_fib(n-1) + math\_fib(n-2) & \text{otherwise} \end{cases}$$

Sometimes, the OCaml code will look like the math definition

However, this is not necessarily the case; e.g. consider the iterative versions of factorial and fibonnaci

Later, we will even not use a math definition directly but express correctness in terms of other function definitions

Proving the Simple Fibonnaci Function Correct

Consider proving the following function definition correct

```
(* fib : int -> int
   precondition: input n > 0
   invariant : fib n evaluates to the
               fibonnaci of n *)
let rec fib n =
   if (n = 1) then 1
   else if (n = 2) then 1
        else fib (n-1) + fib (n-2)
```

The property we want to show for *fib*

for all n, fib(n) evaluates to math_fib(n)

We will work out this proof on the white board; it needs us to use the strong induction principle

Homework problem: Prove the correctness of the iterative version of fibonnaci we developed in homework 2

Induction over Structured Data

The starting point for many of our programs is the description of a type whose values are given as follows

- Via constructors, possibly over other types, that encode a few "base" cases

  E.g. `[]` for lists `Empty` for trees

- Via constructors that use objects of the same type to encode an inductive construction

  E.g. `::` for lists and `Node` for trees

- With an implicit proviso that these are the *only* ways to construct objects of that type

To prove properties over such data, we can use an approach like that for natural numbers with a related induction principle

The key: objects of this type are constructed possibly using simpler objects of the same type

The Generalized Form of the Induction Principle

Suppose we have an inductive type defined as follows

```
type t = C1 of int | C2 of bool * t | C3 of t * t
```

Then, the induction principle associated with t says that

- If $P(C1\ n)$ holds for every integer n,
- For every boolean b, if $P(o')$ holds for any o' of type t, then $P(C2\ (b, o'))$ also holds, and
- If $P(o')$ and $P(o'')$ hold for any o' and o'' of type t, then $P(C3\ (o', o''))$ also holds

then $P(o)$ holds for *every* object o of type t

As with natural numbers, we provide a *method* to prove the property for every object of type t, rather than an explicit proof

Induction for Explicitly Defined Natural Numbers

As a first example of the "structural" induction principle consider the following type declaration

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

To show that $P(n)$ holds for every n of type `nat`, we do the following

- Show that $P(Zero)$ holds
- Show that if $P(n')$ holds for any n' of type `nat`, then $P(Succ\ n')$ also holds

This principle looks a lot like what we had with the machine supported view of natural numbers

Using the Explicit Definition of Natural Numbers

We can define all the usual functions on natural numbers over the explicitly encoded form

```
let rec plusNat x y =
 match x with
 | Zero -> y
 | (Succ x') -> Succ (plusNat x' y)

let rec multNat x y =
 match x with
 | Zero -> Zero
 | (Succ x') -> plusNat y (multNat x' y)
```

Exercise: Try defining the exponentiation function on natural numbers this way

Question: Are there reasons for sometimes preferring the explicit representation to the machine supported form?

Relating the Different Representations

A function to translate explicit natural numbers to the machine supported form

```
let rec toInt n =
  match n with
  | Zero -> 0
  | (Succ n') -> toInt n' + 1
```

Using this translation, we can state correctness properties for functions on the explicit form, e.g.

for all n1, n2,
toInt (plusNat n1 n2) = (toNat n1) + (toNat n2)

Note that in writing equations of this kind we are also assuming similar termination properties on the left and right

I.e., we must also show that both *(toNat n1)* and *(toNat n2)* evaluate to something if and only if *(toInt (plusNat n1 n2))* does

Proving the Correctness of Plus

We use (structural) induction to show for each *n1* that

forall n2, toInt (plusNat n1 n2) = (toNat n1) + (toNat n2)

Base Case: *n1* **is** *Zero*

For any *n2*, both sides of the equation evaluate to whatever *(toInt n2)* evaluates to

Induction Step: *n1* **is** *(Succ n1')*

For any arbitrary *n2*, the induction hypothesis gives us

toInt (plusNat n1' n2) = (toNat n1') + (toNat n2)

The lefthand side of the equation we have to show evaluates to *toInt (plusNat n1' n2) + 1*

Similarly, the righthand side evaluates to *((toNat n1') + 1) + (toNat n2)*

The induction hypothesis now ensures that the equation holds

Referential Transparency

In proofs of properties of the kind considered, we will often replace subexpressions using the function definitions

Such replacements are justified by the principle of *referential transparency*

Referential transparency depends on two properties

Preservation of Termination Behaviour

In particular, equations signify similar termination properties and replacements must not change such behaviour

Preservation of the Result

Replacing subparts must not change the value of the overall expression

The second property is easy when we do not have side effects

For the first, we typically rely on termination

The Induction Principle for Lists

The list type can be viewed as if it is defined as follows

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

The induction principle for lists then says that if you can show

- $P([])$ holds, and
- for every x of type `'a` and ℓ' of type `('a list)` for which $P(\ell')$ holds it is the case that $P(x :: \ell')$ holds

then it must be the case that $P(\ell)$ holds for every ℓ of type `('a list)`

Proving Append Correct Using Induction

We can finally take a look proving `append` correct

```
let rec append l1 l2 =
   match l1 with
   | [] -> l2
   | (h::t) -> h :: (append t l2)
```

Consider first the property $P(\ell_1)$ described below

> *for every* ℓ_2, (*append* ℓ_1 ℓ_2) *evaluates to the result of appending* ℓ_1 *and* ℓ_2

We then want to show: "for every ℓ_1, $P(\ell_1)$ holds"

Easy to do using the induction principle and the following facts:

- the result of appending [] with ℓ must be ℓ
- if the append of ℓ_1 and ℓ_2 is ℓ then the append of $(x :: \ell_1)$ and ℓ_2 must be $(x :: \ell)$

Termination and Coverage Checking

We can prove *append* correct in another (more intuitive) way

We show that it works for *all* inputs by showing two things:

- Every possible case for the (first) input list is covered and works correctly assuming recursive calls are "good"

  In effect, we are showing that the property claimed for *append* is *invariant* over recursive calls
- For each case of the first input, we must also check that *append* terminates

  This is important: the "invariant" argument uses natural number induction over the recursive calls

Note that OCaml helps you directly with the coverage aspect, but it gives you flexibility with analyzing termination

Note also that the induction principle for lists implicitly builds both aspects into the reasoning process

Recap: Induction and an Informal Style of Reasoning

We can prove functions like *append* correct in another, possibly more more intuitive style

We show that they work for all relevant inputs by showing

- Every input case is covered and the function works correctly assuming recursive calls are “good”

  In effect, we are showing that the claimed property to be an *invariant*

- For each case of the first input, we check also that the function terminates

  This is important: the “invariant” argument uses natural number induction over the recursive calls

The induction principle for a type builds both aspects implicitly into the reasoning process

A Broken Definition of Append

To understand the connection between the intuitive reasoning style and structural induction consider the following code

```
let rec append l1 l2 =
   match l1 with
   | [] -> l2
   | (h::t) -> (append l1 l2)
```

We will not be able to show now that *append* terminates for all inputs

Correspondingly, we cannot complete an inductive proof: the induction hypothesis tells us something only about *(append t l2)*

To see a similar connection with coverage checking, consider the case when the second clause is left out in the definition

Relating Functions on Lists

Consider the following function for summing up list elements

```
let rec sumlist lst =
  match lst with
  | [] -> 0
  | (h::t) -> h + (sumlist t)
```

We can then prove that *append* preserves sums

for all integer lists l_1 and l_2
sumlist (append l_1 l_2) = (sumlist l_1) + (sumlist l_2)

Specifically, we show by induction for every integer list l_1 that

for all integer lists l_2
sumlist (append l_1 l_2) = (sumlist l_1) + (sumlist l_2)

Pay special attention to referential transparency in the proof that we do on the white board in class

Treating Program Expressions as an Algebra

In the most recent reasoning examples, we have treated program expressions in an algebraic way

In particular, we have used function definitions as “known” identities to transform expressions and equalities

Moreover, after we derive new equalities, we can use them also in the reasoning process

Of course, we have to interpret the equalities carefully and we have to ensure referential transparency for the transformations

In general, such reasoning works out and is a special attraction of side-effect free functional programming

Reasoning About the Reverse Function

Consider now the following definition of *reverse*

```
let rec reverse l =
   match l with
   | [] -> []
   | (h::t) -> append (reverse t) [h]
```

Once we have a mathematical definition of reverse we can prove correctness easily

Some further properties that we will prove on the white board using induction on lists

forall lists l, forall list elements x
(reverse (append l [x])) = x::(reverse l)

forall l, reverse (reverse l) = l

We get to use the first property in proving the second

Binary Trees and their Induction Principle

As another example, consider the binary tree type

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

The induction principle associated with this type says that if you can show

- $P(Empty)$ holds, and
- if $P(t_1)$ and $P(t_2)$ hold for t_1 and t_2 of type *('a btree)*, then $P(Node(x, t_1, t_2))$ holds for any x of type *'a*

then $P(t)$ must hold for any t of type *('a btree)*

Lets look at some uses of this induction principle to get familiar with it

Summing Up Tree Elements

Consider the following function definition

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

To determine its correctness, we have to define the mathematical notion of summing up a tree

$$math\_ST(t) = \begin{cases} 0 & \text{if } t = Empty \\ i + math\_ST(l) + math\_ST(r) & t = Node(i, l, r) \end{cases}$$

Once we have done this it is easy to show

$\forall t \in (int\ btree)$, *(sumTree t) evaluates to math_sT(t)*

We will do this on the white board using induction on trees

Inserting in Binary Search Trees

For simplicity, we will work with the “”slightly broken” version of search tree functions that leave the ordering implicit

```
let rec insert t i =
   match t with
   | Empty -> Node (i,Empty,Empty)
   | Node (i',l,r) ->
        if (i < i') then Node (i',insert l i,r)
        else Node (i',l,insert r i)
```

Lets try a simple proof first that shows that insertion has the kind of effect we want on the sum

$\forall t \in (\textit{int btree}), \forall i \in \textit{int}$

sumTree (insert t i) = (sumTree t) + i

We will prove this on the white board using induction on the tree type

Correctness for Binary Search Tree Functions

We would want these functions to preserve invariants and also achieve the objectives set out for them

For example, the *insert* function should satisfy the following properties

- If the input tree is “ordered” then the output tree should also be ordered
- After the insert, anything that was in the old tree should be in the new one too
- After the insert, the only things in the new tree should be things in the old tree or the item inserted

We could try to state and prove these properties informally

The approach we will follow: define additional functions and express the properties through relationships with them

Insert and Minimum and Maximum in Trees

Consider the following functions for finding the smallest and largest elements in trees

```
let rec minTree =
 function
 | Empty -> None
 | Node (i,l,r) ->
     match (minTree l) with
      | None -> (Some i)
      | (Some i') as m -> m
```

```
let rec maxTree =
 function
 | Empty -> None
 | Node (i,l,r) ->
     match (maxTree r) with
      | None -> (Some i)
      | (Some i') as m -> m
```

How does *insert* impact the results of these functions?

- New min should be the old min or the added item

  $\forall t \in (\text{'}a\ btree), \forall t', \forall x \quad ((insert\ t\ x) \text{ evaluates to } t') \supset$
  $(minTree\ t' = minTree\ t) \vee (minTree\ t' = Some\ x))$

- New max should be the old max or the added item

  $\forall t \in (\text{'}a\ btree), \forall x \in' a, \quad ((insert\ t\ x) \text{ evaluates to } t') \supset$
  $(maxTree\ t' = maxTree\ t) \vee (maxTree\ t' = Some\ x))$

An inductive proof of these is left as a lab exercise

Insert Preserves Search Trees

We first define a function for identifying search trees

```
let rec isSearchTree t =
 match t with
 | Empty -> true
 | Node (i,l,r) ->
    isSearchTree l && isSearchTree r &&
     (bigger i (maxTree l)) && (smaller i (minTree r))
```

The following property then expresses the fact that *insert* preserves the invariant

$\forall t \in (\text{'}a\ \textit{btree}), \forall x \in \text{'}a$
$((\textit{insert}\ t\ x)$ evaluates to t' and
$(\textit{isSearchTree}\ t)$ evaluates to $\textit{true}) \supset$
$(\textit{isSearchTree}\ t')$ evaluates to $\textit{true}$

We will prove this property on the white board by induction on t, using the earlier properties concerning *minTree* and *maxTree*

Trees Have the Right Elements After Insertion

The remaining properties we want to check about *insert*

- *insert* does not lose any elements, i.e. what was in the tree earlier remains in it
- *insert* adds only the one element given to it

To check the elements in the tree in stating these properties, we can use the *find* function

```
let rec find t i =
   match t with
   | Empty -> false
   | Node (i',l,r) ->
       if (i = i') then true
       else if (i < i') then find l i
            else find r i
```

Trees Have the Right Elements After Insertion (Contd)

Using *find* we can formalize the properties we want to check for *insert* as follows

- *insert* does not lose any elements
  $\forall t \in (\prime a\ btree), \forall x \in \prime a$
  $((insert\ t\ x)$ evaluates to $t' \supset$
  $\forall x' \in' a((find\ t\ x')$ evaluates to $true \supset$
  $(find\ t'\ x')$ evaluates to $true))$
- *insert* adds only the one element given to it
  $\forall t \in (\prime a\ btree), \forall x \in \prime a$
  $((insert\ t\ x)$ evaluates to $t' \supset$
  $\forall x' \in' a((find\ t'\ x')$ evaluates to $true \supset$
  $((x = x') \vee (find\ t\ x')$ evaluates to $true)))$

Both can be proved by induction on the structure of t

One of these will be solved in the handout and the other is a homework problem

Translating the Insights to Imperative Programs

Imperative programs are essentially *state transformers*

Correctness here be expressed in terms of properties we want to hold between program variables after each statement

The key difficulty is in dealing with loops; for straightline programs, the way properties are transformed is obvious

For loops, we can use the following idea

- Visualize the loop as a tail-recursive function
- Think of the precondition of the tail-recursive function as a loop invariant
- Think of the expected output of the function to determine what should be true after the loop

Lets look at some examples of the use of this approach

Reasoning About an Iterative Factorial Program

The iterative program

```
acc = 1; i = n ;
while (i > 0)
  { acc = i * acc;  i = i - 1; }
```

The tail-recursive version of the same program

```
let rec tr_fact i acc =
   if (i > 0) then tr_fact (i-1) (i * acc)
   else acc
```

The precondition for the tail-recursive function:

$$acc = n!/i! \wedge 0 <= i <= n$$

The basis for the correctness of the function:

when $i = 0$, *we have* $acc = n!$

But these are simply the loop invariant and the property true after the loop

Reasoning About an Iterative GCD Program

The iterative program

```
n = n0; m = m0;
while (not (n = m)) {
  if (n > m) then n = n - m,
  else m = m - n; }
```

The tail-recursive rendition of the same program

```
let rec gcd n m =
  if (n = m) then n
  else if (n > m) then gcd (n - m) m
       else gcd n (m - n)
```

The precondition when reasoning about the function

$$gcd(n, m) = gcd(n_0, m_0) \textit{ for chosen } n_0, m_0$$

The basis for correctness of the function

$$gcd(n, m) = gcd(n_0, m_0) = n \textit{ if } n = m$$

Once again these are the loop invariant and end conditions

More on Reasoning About Imperative Programs

Quantifying program behaviour via state transformations is used extensively in software development and testing:

Three references to check out:

- An extremely readable paper by Jon Bentley on programming and checking correctness of programs
- The original paper by Tony Hoare that presented the formal method for reasoning about program behaviour
- A retrospective paper by Tony Hoare on how the method he originated has panned out over time

These papers will be posted on the course web page

If these kinds of topics interest you, check out the Coq system and also think of doing a programming languages course!