Recursion and Induction

15-150

Lecture 3: September 2, 2025

Stephanie Balzer
Carnegie Mellon University

Recap of week 1

Functional programming

- evaluation of expressions (no mutation!)
- facilitates specification and reasoning about program
  - correctness proof (today's topic!)
- facilitates parallelism

Types, expressions, values

- types as specifications
- observation: once your program type checks, it works!

Recap of week 1

we’ll revisit exact definition

Extensional equivalence (≅)

- “Two things are equal if the behave the same”
- facilitates compositional (aka modular) reasoning
  - replace equals by equals in any sub-expression

Declarations, binding and scope

- shadowing of bindings
- function declarations bind a closure to the function identifier
  - closure comprises lambda expression and environment with bindings existing at declaration time

Recap of week 1

Pattern matching

- patterns are used at binding sites of values
  - eg, val bindings, function arguments, case expression
- allow us to match against an expected value
- allow us to decompose a value in its constituent parts, introducing appropriate bindings for parts

Recap of week 1

5-step methodology of function declaration

1. function name and type
2. REQUIRES: precondition
3. ENSURES: postcondition
4. function body
5. tests

Today, we add a 6th step:

6. correctness proof

Today's topic: functional correctness

Let's prove our programs correct, one function at a time!

- we will use three kinds of induction:
  - mathematical induction
  - strong induction
  - structural induction
- we consider how expressions are evaluated
- we may appeal to mathematical properties and assume that SML implements them correctly

An example: compute n^k

```
(* power : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: power(n,k) ==> n^k, with 0^0 = 1.
*)
```

An example: compute n^k

```
(* power : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: power(n,k) ==> n^k, with 0^0 = 1.
*)

fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

pattern matching

An example: compute n^k

```
(* power : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: power(n,k) ==> n^k, with 0^0 = 1.
*)

fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

recursive call

An example: compute n^k

```
(* power : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: power(n,k) ==> n^k, with 0^0 = 1.
*)

fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

this function is not very efficient:

eg, $3^7 = 3 * 3 * 3 * 3 * 3 * 3 * 3$

Number of recursive calls: O(k)

Can we do better than that?

Idea for making power more efficient

- this function is not very efficient:

$$3^7 = 3 * 3 * 3 * 3 * 3 * 3 * 3$$

eg, $3^7 = 3 * 3 * 3 * 3 * 3 * 3 * 3$

- Number of recursive calls: O(k)
- Can we do better than that?

Assume we have functions `even` and `square`. Now we can get a more efficient implementation:

eg, $3^7 = 3 * (3^{3})^{2}$

$= 3 * (3 * (3^{1})^{2})^{2}$

$= 3 * (3 * (3 * 1)^{2})^{2}$

- Number of recursive calls: O(log(k))

A more efficient version of power

A more efficient version of power

```
(* even : int -> bool
   REQUIRES: true
   ENSURES: even(k) evaluates to true if k is even
                    evaluates to false if k is odd.
*)
```

A more efficient version of power

```
(* even : int -> bool
   REQUIRES: true
   ENSURES: even(k) evaluates to true if k is even
                    evaluates to false if k is odd.
*)

fun even (k:int) : bool = ((k mod 2) = 0)
```

A more efficient version of power

```
(* even : int -> bool
   REQUIRES: true
   ENSURES: even(k) evaluates to true if k is even
                    evaluates to false if k is odd.
*)

fun even (k:int) : bool = ((k mod 2) = 0)

(* square : int -> int
   REQUIRES: true
   ENSURES: square(n) ==> n^2
*)
```

A more efficient version of power

```
(* even : int -> bool
   REQUIRES: true
   ENSURES: even(k) evaluates to true if k is even
                    evaluates to false if k is odd.
*)

fun even (k:int) : bool = ((k mod 2) = 0)

(* square : int -> int
   REQUIRES: true
   ENSURES: square(n) ==> n^2
*)

fun square (n:int) : int = n * n
```

A more efficient version of power

```
(* powere : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: powere(n,k) ==> n^k, with 0^0 = 1.

   powere computes n^k using O(log(k)) multiplies.
*)
```

A more efficient version of power

```
(* powere : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: powere(n,k) ==> n^k, with 0^0 = 1.

   powere computes n^k using O(log(k)) multiplies.
*)

fun powere (_:int, 0:int) : int =
  | powere (n:int, k:int) : int =
```

A more efficient version of power

```
(* powere : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: powere(n,k) ==> n^k, with 0^0 = 1.

   powere computes n^k using O(log(k)) multiplies.
*)

fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
```

A more efficient version of power

```
(* powere : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: powere(n,k) ==> n^k, with 0^0 = 1.

   powere computes n^k using O(log(k)) multiplies.
*)

fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
```

exponent k is even

A more efficient version of power

```
(* powere : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: powere(n,k) ==> n^k, with 0^0 = 1.

   powere computes n^k using O(log(k)) multiplies.
*)

fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Number of recursive calls: O(log(k))

Let’s verify our naive version of power

```
(* power : (int * int) -> int
   REQUIRES: k >= 0
   ENSURES: power(n,k) ==> n^k, with 0^0 = 1.
*)

fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

- How shall we proceed?
- Let’s use mathematical induction!

Mathematical (simple, weak) induction

To prove a property P(n) for every natural number n:

- show that P(0) holds
- then, show that for all $k \geq 0$, P(k+1) follows logically from P(k).

base case

Mathematical (simple, weak) induction

To prove a property P(n) for every natural number n:

- show that P(0) holds
- then, show that for all $k \geq 0$, P(k+1) follows logically from P(k).

inductive step

Mathematical (simple, weak) induction

To prove a property P(n) for every natural number n:

- show that P(0) holds
- then, show that for all k $\geq$ 0,
  P(k+1) follows logically from P(k).

Why does it work?

- P(0) is proved directly.
- P(1) follows from P(0).
- P(2) follows from P(1).
- etc…

Let's verify our naive version of power

```
fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

Theorem: `power(n, k)` evaluates to n^k, for all integer values $k \geq 0$ and all integer values `n`.

Proof by mathematical induction on ???

Let’s verify our naive version of power

```
fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

Theorem: `power(n, k)` evaluates to n^k, for all integer values $k \geq 0$ and all integer values `n`.

→ Proof by mathematical induction on k.

k is the integer that gets smaller!

needed for applying IH!

Let’s verify our naive version of power

```
fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

Theorem: `power(n, k)` evaluates to n^k, for all integer values $k \geq 0$ and all integer values `n`.

- Proof by mathematical induction on k.
- Let’s do the proof together!

Let's verify our naive version of power

```
fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

Theorem: `power(n, k)` evaluates to n^k, for all integer values `k` ≥ `0` and all integer values `n`.

Proof: By mathematical induction on k.

Base case: k = 0.

<u>Need to show</u>: `power(n,0)` evaluates to n^0, for all n. Note: $n^0 = 1$.

<u>Showing</u>:

`power(n,0)`

⟹ `1` (step, 1st clause of `power`)

Let's verify our naive version of power

```
fun power (_:int, 0:int) : int = 1
  | power (n:int, k:int) : int = n * power(n, k-1)
```

Inductive case: Step from k to k+1, with $k \geq 0$.

IH: `power(n,k)` evaluates to n^k, for $k \geq 0$ and all integers n.

Need to show: `power(n,k+1)` evaluates to n^{k+1}.

Showing:

	`power(n,k+1)`	
$\Longrightarrow$	`n * power(n,k+1-1)`	(step, 2nd clause of `power`)
$\Longrightarrow$	`n * power(n,k)`	(math)
$\Longrightarrow$	$n * n^k$	(IH)
$\Longrightarrow$	$n \cdot n^k$	(evaluation rule for $*$)
$\Longrightarrow$	n^{k+1}	(math)

Let's verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Theorem: `power(n, k)` evaluates to n^k, for all integer values $k \geq 0$ and all integer values `n`.

- Proof by ???
  - Note: k does no longer decrease by one!

Let's verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Theorem: `power(n, k)` evaluates to n^k, for all integer values $k \geq 0$ and all integer values `n`.

➡ Proof by strong induction on k.

Strong induction

To prove a property P(n) for every natural number n:

- show that P(0) holds
- then, show that for all k > 0,
  P(k) follows logically from {P(0), …, P(k-1)}.

base case

Strong induction

To prove a property P(n) for every natural number n:

- show that P(0) holds
- then, show that for all $k > 0$,
  P(k) follows logically from {P(0), …, P(k-1)}.

inductive step

Strong induction

To prove a property P(n) for every natural number n:

- show that P(0) holds
- then, show that for all k > 0,
  P(k) follows logically from {P(0), …, P(k-1)}.

Note: allowed to appeal to IH for any k’ < k!

For mathematical induction, IH can only be appealed to for the immediate predecessor!

Let's verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

not immediate predecessor!

Theorem: `power(n, k)` evaluates to n^k, for all integer values $k \geq 0$ and all integer values `n`.

- Proof by strong induction on k.
- Notice, the code tells us what induction principle to use!
- Let's do the proof together!

Let’s verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Theorem: `powere(n, k)` evaluates to n^k, for all integer values `k` $\geq$ `0` and all integer values `n`.

Proof: By strong induction on k.

Base case: k = 0.

Need to show: `powere(n,0)` evaluates to n^0, for all n. Note: $n^0 = 1$.

Showing:

```
    powere(n,0)
⟹  1                    (step, 1st clause of powere)
```

Let's verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Inductive case: $k > 0$.

IH: `powere(n,k')` evaluates to $n^{k'}$, for $0 \leq k' < k$ and all integers n.

Need to show: `powere(n,k)` evaluates to n^k, for all integers n.

Showing:

```
   powere(n,k)
⟹ if even(k)                    (step, 2nd clause of powere)
   then square(powere(n,k div 2))
   else n * powere(n,k-1)
```

Distinguish two subcases, depending on whether k is even or odd.

Let's verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Inductive case: $k > 0$.

Case: $k = 2k'$, for some $k' < k$, assuming correctness of even.

Showing:

`powere(n,k)`

$\Longrightarrow$ `square(powere(n, k div 2))` (by assumption about `even`)

$\Longrightarrow$ `square(powere(n, k'))` (since $k = 2k'$, assuming `div` is correct)

$\Longrightarrow$ `square(`$n^{k'}$`)` (IH)

$\Longrightarrow$ $(n^{k'})^2$ (by Lemma)

Lemma: For every integer value n, square(n) evaluates to n^2.

Let's verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Inductive case: $k > 0$.

Case: $k = 2k'$, for some $k' < k$, assuming correctness of even.

Showing:

	`powere(n,k)`	
$\Longrightarrow$	`square(powere(n, k div 2))`	(by assumption about **`even`**)
$\Longrightarrow$	`square(powere(n, k'))`	(since $k = 2k'$, assuming **`div`** is correct)
$\Longrightarrow$	`square(`$n^{k'}$`)`	(IH)
$\Longrightarrow$	$(n^{k'})^2$	(by Lemma)
=	$n^{2k'} = n^k$	(math)

Let's verify our more efficient version of power, powere

```
fun powere (_:int, 0:int) : int = 1
  | powere (n:int, k:int) : int =
      if even(k)
      then square(powere(n, k div 2))
      else n * powere(n, k-1)
```

Inductive case: k > 0.

Case: k = 2k'+1, for some k' < k, assuming correctness of even.

Showing:

`powere(n,k)`

$\Longrightarrow$ `n * (powere(n, k-1))` (by assumption about **`even`**)

$\Longrightarrow$ `n *` n^{k-1} (IH)

$\Longrightarrow$ n^k (math)

That's all for today. See you on Thursday!