Recursion and Induction

15-150

Lecture 3: September 3, 2024

Stephanie Balzer
Carnegie Mellon University

Recap of week 1

Recap of week 1

Functional programming

Recap of week 1

Functional programming

evaluation of expressions (no mutation!)

Recap of week 1

Functional programming

- evaluation of expressions (no mutation!)
- facilitates specification and reasoning about program

Recap of week 1

Functional programming

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

Recap of week 1

Functional programming

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

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

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

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

Extensional equivalence ($\cong$)

Recap of week 1

Extensional equivalence (≅)

→ "Two things are equal if the behave the same"

Recap of week 1

we'll revisit
exact definition

Extensional equivalence (≅)

→ "Two things are equal if the behave the same"

Recap of week 1

Extensional equivalence (≅)

→ "Two things are equal if the behave the same"

Recap of week 1

Extensional equivalence ($\cong$)

- "Two things are equal if the behave the same"
- facilitates compositional (aka modular) reasoning

Recap of week 1

Extensional equivalence (≅)

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

Recap of week 1

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

Recap of week 1

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

Recap of week 1

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

Recap of week 1

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

Recap of week 1

Pattern matching

→ patterns are used at binding sites of values

Recap of week 1

Pattern matching

- patterns are used at binding sites of values
  - eg, val bindings, function arguments, case expression

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

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

Recap of week 1

5-step methodology of function declaration

1 function name and type

Recap of week 1

5-step methodology of function declaration

1. function name and type
2. REQUIRES: precondition

Recap of week 1

5-step methodology of function declaration

1. function name and type
2. REQUIRES: precondition
3. ENSURES: postcondition

Recap of week 1

5-step methodology of function declaration

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

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

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:

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

Today’s topic: functional correctness

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

Today’s topic: functional correctness

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

→ we will use three kinds of induction:

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

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

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

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

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)
```

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)
```

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)
```

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)
```

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)
```

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:

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

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)

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.
*)

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)
```

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)
```

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)
```

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)
```

exponent k is even

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

Let's verify our naive version of power

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)
```

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 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

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).

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).

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).

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?**

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.

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).

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).

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

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)
```

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`.

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.

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!

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!

need 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 more efficient version of power, 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)
```

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`.

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 ???

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)}.

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)}.

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)}.

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!

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

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.

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.

Notice, the code tells us what induction principle to use!

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 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!

Taking stock

Taking stock

We proved two functions correct, using weak and strong induction on natural numbers.

Taking stock

- We proved two functions correct, using weak and strong induction on natural numbers.
- Sometimes we need to rephrase the theorem to facilitate proof.

Taking stock

- We proved two functions correct, using weak and strong induction on natural numbers.
- Sometimes we need to rephrase the theorem to facilitate proof.
  - aka generalize IH (see lecture notes for an example)

Taking stock

- We proved two functions correct, using weak and strong induction on natural numbers.
- Sometimes we need to rephrase the theorem to facilitate proof.
  - aka generalize IH (see lecture notes for an example)
- Sometimes, the function doesn't facilitate a proof by induction on a **natural number**.

Taking stock

- We proved two functions correct, using weak and strong induction on natural numbers.
- Sometimes we need to rephrase the theorem to facilitate proof.
  - aka generalize IH (see lecture notes for an example)
- Sometimes, the function doesn't facilitate a proof by induction on a **natural number**.
- Instead, induction over the **structure** of values defined by a datatype declaration is more suited.

Taking stock

- We proved two functions correct, using weak and strong induction on natural numbers.
- Sometimes we need to rephrase the theorem to facilitate proof.
  - aka generalize IH (see lecture notes for an example)
- Sometimes, the function doesn't facilitate a proof by induction on a **natural number**.
- Instead, induction over the **structure** of values defined by a datatype declaration is more suited.

more on datatypes in
later lectures!

Taking stock

- We proved two functions correct, using weak and strong induction on natural numbers.
- Sometimes we need to rephrase the theorem to facilitate proof.
  - aka generalize IH (see lecture notes for an example)
- Sometimes, the function doesn't facilitate a proof by induction on a **natural number**.
- Instead, induction over the **structure** of values defined by a datatype declaration is more suited.

Taking stock

- We proved two functions correct, using weak and strong induction on natural numbers.
- Sometimes we need to rephrase the theorem to facilitate proof.
  - aka generalize IH (see lecture notes for an example)
- Sometimes, the function doesn’t facilitate a proof by induction on a **natural number**.
- Instead, induction over the **structure** of values defined by a datatype declaration is more suited.
  - today: example using **structural induction** on lists

Lists

Lists

Type: `t list` for any type `t`

Lists

Type:	`t list`	for any type `t`
Values:	$[v_1, \ldots, v_n]$	where, v_i is a value of type `t`, $n \geq 0$
	`[]` or `nil`	empty list

Lists

Type:	`t list`	for any type `t`
Values:	$[v_1, \ldots, v_n]$	where, v_i is a value of type `t`, $n \geq 0$
	`[]` or `nil`	empty list
Expressions:	`v`	all the values
	`e::es`	where `e: t` and `es: t list`

Lists

Type:	`t list`	for any type `t`
Values:	`[v1,…,vn]`	where, v_i is a value of type `t`, $n \geq 0$
	`[]` or `nil`	empty list
Expressions:	`v`	all the values
	`e::es`	where `e: t` and `es: t list`

“cons”

Lists

Type:	`t list`	for any type `t`
Values:	`[`v_1`,…,`v_n`]`	where, v_i is a value of type `t`, $n \geq 0$
	`[]` or `nil`	empty list
Expressions:	`v`	all the values
	`e::es`	where `e: t` and `es: t list`

Lists

Type:	`t list`	for any type `t`
Values:	`[v1,…,vn]`	where, v_i is a value of type `t`, $n \geq 0$
	`[]` or `nil`	empty list
Expressions:	`v`	all the values
	`e::es`	where `e: t` and `es: t list`
		eg, `1::[2, 3]` yields `[1, 2, 3]`

Lists

Type:	`t list`	for any type `t`
Values:	`[v1,…,vn]`	where, v_i is a value of type `t`, $n \geq 0$
	`[]` or `nil`	empty list
Expressions:	`v`	all the values
	`e::es`	where `e: t` and `es: t list`
		eg, `1::[2, 3]` yields `[1, 2, 3]`

cons is right-associative

Lists

Type: `t list` for any type `t`

Values: `[v1,…,vn]` where, v_i is a value of type `t`, $n \geq 0$

`[]` or `nil` empty list

Expressions: `v` all the values

`e::es` where `e: t` and `es: t list`

eg, `1::[2, 3]` yields `[1, 2, 3]`

cons is right-associative

ie, `1::2::3::nil` means `1::(2::(3::nil))`

Lists

Type:	`t list`	for any type `t`
Values:	`[v1,…,vn]`	where, v_i is a value of type `t`, $n \geq 0$
	`[]` or `nil`	empty list
Expressions:	`v`	all the values
	`e::es`	where `e: t` and `es: t list`
		eg, `1::[2, 3]` yields `[1, 2, 3]`

cons is right-associative

ie, `1::2::3::nil` means `1::(2::(3::nil))`

cons evaluates left to right (for sequential evaluation)

Length function for an int list

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)
```

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length (                ) : int =
  | length (        ) =
```

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int =
  | length (     ) =
```

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int =
  | length (x::xs) =
```

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int =
  | length (x::xs) =
```

patterns!

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int =
  | length (x::xs) =
```

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) =
```

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Let’s verify that length is total (“always yields a value”)!

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

- Let’s verify that length is total (“always yields a value”)!
- Unlike in math, functions in SML are not total

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

- Let’s verify that length is total (“always yields a value”)!
- Unlike in math, functions in SML are not total
  - eg, recursive functions can loop!

Totality

Totality

Definition:

A function `f: X -> Y` is total, if `f` reduces to a value and `f(x)` reduces to a value for all values `x` in `X`.

Totality

Definition:

A function `f: X -> Y` is total, if `f` reduces to a value and `f(x)` reduces to a value for all values `x` in `X`.

→ Partiality complicates extensional equivalence (next time!)

Length function for an int list

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

➡ Let's verify that length is total ("always yields a value")!

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Let’s verify that length is total (“always yields a value”)!

We proceed by structural induction on argument list.

Structural induction for lists

Structural induction for lists

To prove a property P(L) for every value L of type int list:

- show that P([ ]) holds
- show that, if P(L’) for some value L’ of type int list,
  then it also holds for v :: L’ (for any value v of type int).

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

➡ Let's verify that length is total ("always yields a value")!

➡ We proceed by structural induction on argument list.

Length function for an int list

```
(* length : int list -> int
   REQUIRES: true
   ENSURES: length(L) returns the number of
            integers in L.
*)

fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

- Let’s verify that length is total (“always yields a value”)!
- We proceed by structural induction on argument list.
  - Let’s do the proof together!

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Proof: By structural induction on `L`.

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Proof: By structural induction on `L`.

Base case: L = `[]`.

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Proof: By structural induction on `L`.

Base case: L = `[]`.

Need to show: `length([])` evaluates to some value `v:int`.

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Proof: By structural induction on `L`.

Base case: L = `[]`.

Need to show: `length([])` evaluates to some value `v:int`.

Showing:

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Proof: By structural induction on `L`.

Base case: L = `[]`.

Need to show: `length([])` evaluates to some value `v:int`.

Showing:

```
    length []
```

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Proof: By structural induction on `L`.

Base case: L = `[]`.

Need to show: `length([])` evaluates to some value `v:int`.

Showing:

```
    length []
⟹  0
```

Length is total: proof

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Theorem: For al values `L:int list`, `length(L)` evaluates to an integer value `v`.

Proof: By structural induction on `L`.

Base case: L = `[]`.

Need to show: `length([])` evaluates to some value `v:int`.

Showing:

```
    length []
⟹  0
```
(step, 1st clause of `length`)

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

```
    length(x::xs)
```

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

```
    length(x::xs)
⟹  1 + length(xs)
```

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

`length(x::xs)`

$\Longrightarrow$ `1 + length(xs)` (step, 2nd clause of `length`)

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

`length(x::xs)`

$\Longrightarrow$ `1 + length(xs)` (step, 2nd clause of `length`)

$\Longrightarrow$ `1 + v`

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

`length(x::xs)`

$\Longrightarrow$ `1 + length(xs)` (step, 2nd clause of `length`)

$\Longrightarrow$ `1 + v` (IH)

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

`length(x::xs)`

$\Longrightarrow$ `1 + length(xs)` (step, 2nd clause of `length`)

$\Longrightarrow$ `1 + v` (IH)

$\Longrightarrow$ `v'`

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

	`length(x::xs)`	
$\Longrightarrow$	`1 + length(xs)`	(step, 2nd clause of `length`)
$\Longrightarrow$	`1 + v`	(IH)
$\Longrightarrow$	`v'`	(some `v'`, assume + total)

Length is total: proof continued

```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

Inductive case: L = `x::xs`, for some values `x:int` and `xs:int list`.

IH: `length(xs)` evaluates to some value `v:int`.

Need to show: `length(x::xs)` evaluates to some value `v':int`.

Showing:

	`length(x::xs)`	
$\Longrightarrow$	`1 + length(xs)`	(step, 2nd clause of `length`)
$\Longrightarrow$	`1 + v`	(IH)
$\Longrightarrow$	`v'`	(some `v'`, assume + total)

SML addition must be total!

Correspondence

Correspondence

Data structure	Code	Proof

Correspondence

Data structure **Code** **Proof**

base case(s):

inductive/recursive case(s):

Correspondence

Data structure	Code	Proof
base case(s):		
`0`	`fun power(_, 0) = 1`	`power(_, 0)` ↪ 1
inductive/recursive case(s):		

Correspondence

Data structure	**Code**	**Proof**

base case(s):

`0`	`fun power(_, 0) = 1`	`power(_, 0)` ↪ 1

inductive/recursive case(s):

`(k-1)+1`	`power(n,k) = n*power(n, k-1)`	IH: `power(n,k)` ↪ n^k NTS: `power(n,k+1)` ↪ n^{k+1}

Correspondence

Data structure	**Code**	**Proof**
base case(s):		
`0`	`fun power(_, 0) = 1`	`power(_, 0)` ↪ 1
`[ ]`	`0`	`length []` ↪ 0
inductive/recursive case(s):		
`(k-1)+1`	`power(n,k) = n*power(n, k-1)`	IH: `power(n,k)` ↪ n^k NTS: `power(n,k+1)` ↪ n^{k+1}

Correspondence

Data structure	**Code**	**Proof**
base case(s):		
`0`	`fun power(_, 0) = 1`	`power(_, 0)` ↪ 1
`[ ]`	`0`	`length []` ↪ 0
inductive/recursive case(s):		
`(k-1)+1`	`power(n,k) = n*power(n, k-1)`	IH: `power(n,k)` ↪ n^k NTS: `power(n,k+1)` ↪ n^{k+1}
`x :: xs`	`length(x::xs) = 1+length(xs)`	IH: `length(xs)` ↪ v NTS: `length(x::xs)` ↪ v’

Correspondence

Data structure	**Code**	**Proof**
base case(s):		
`0`	`fun power(_, 0) = 1`	`power(_, 0)` ↪ 1
`[ ]`	`0`	`length []` ↪ 0
inductive/recursive case(s):		
`(k-1)+1`	`power(n,k) = n*power(n, k-1)`	IH: `power(n,k)` ↪ n^k NTS: `power(n,k+1)` ↪ n^{k+1}
`x :: xs`	`length(x::xs) = 1+length(xs)`	IH: `length(xs)` ↪ v NTS: `length(x::xs)` ↪ v'

possibly several bases and inductive cases!