Lists, structural induction, tail recursion, and extensional equivalence

15-150

Lecture 5: September 4, 2025

Stephanie Balzer
Carnegie Mellon University

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:	`[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" constructor

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

"cons" constructor

can be matched against!

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

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

cons is right-associative

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`
		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 and functions

Totality and functions

Definition:

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

Totality and functions

Definition:

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

f may be some expression that reduces to a function

Totality and functions

Definition:

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

f may be some expression that reduces to a function

→ Partiality complicates extensional equivalence (more later!)

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 = `[]`.

<u>Need to show</u>: `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

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

Length is total: proof

```
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

```
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

```
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

```
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`.

<u>IH</u>: `length(xs)` evaluates to some value `v:int`.

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

<u>Showing</u>:

Length is total: proof

```
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

```
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

```
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)
```
(step, 2nd clause of `length`)

Length is total: proof

```
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

```
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

```
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)        (step, 2nd clause of length)
⟹  1 + v                 (IH)
⟹  v'
```

Length is total: proof

```
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

```
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**

base case(s):

inductive/recursive case(s):

Correspondence

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

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!

Let's reconsider our length function

Let’s reconsider our length function

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

Let’s reconsider our length function

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

Is this function efficient?

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

Let’s reconsider our length function

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

Is this function efficient? Well, let’s look at an evaluation trace:

```
length [3,4,5] ==>
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
[…, 3/x, [4,5]/xs] 1 + length(xs)
```

```
length [3,4,5] ==> 1 + length [4,5]
```

Let’s reconsider our length function

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

Is this function efficient? Well, let’s look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
```

Let’s reconsider our length function

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

Is this function efficient? Well, let’s look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

space

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let's reconsider our length function

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

Is this function efficient? Well, let's look at an evaluation trace:

space

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let’s reconsider our length function

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

Is this function efficient? Well, let’s look at an evaluation trace:

space

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let’s reconsider our length function

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

Is this function efficient? Well, let’s look at an evaluation trace:

space

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let’s reconsider our length function

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

Is this function efficient? Well, let’s look at an evaluation trace:

space

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let’s reconsider our length function

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

Is this function efficient? Well, let’s look at an evaluation trace:

space

```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

each recursive call has pending computations

Let’s write a tail recursive version!

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

Let's write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let's write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

accumulator

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let's write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

extensional equivalence

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

extensional equivalence

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

extensional equivalence

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

space-inefficient version

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int =
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) =
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Let's write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Let's write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

tail call

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

tail call

`1+acc` happens before recursive call!

Remember: evaluation order for applications

Remember: evaluation order for applications

Function application: `e1 e2`

Remember: evaluation order for applications

Function application: `e`$_1$ `e`$_2$

1 Reduce `e`$_1$ to a function `value fn (x:t) => e`.

Remember: evaluation order for applications

Function application: `e`$_1$ `e`$_2$

1. Reduce `e`$_1$ to a function `value fn (x:t) => e`.
2. Reduce `e`$_2$ to a value `v`.

Remember: evaluation order for applications

Function application: `e`$_1$ `e`$_2$

1. Reduce `e`$_1$ to a function value `fn (x:t) => e`.
2. Reduce `e`$_2$ to a value `v`.
3. Extend function closure environment with binding for `x` and `v`.

Remember: evaluation order for applications

Function application: $\texttt{e}_1\ \texttt{e}_2$

1. Reduce $\texttt{e}_1$ to a function `value fn (x:t) => e`.
2. Reduce $\texttt{e}_2$ to a value `v`.
3. Extend function closure environment with binding for `x` and `v`.
3. Evaluate function body `e` in resulting environment.

Let's write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Let's write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

fun length' (L: int list):
```

Let’s write a tail recursive version!

Definition: A function is **tail recursive**, if it does not perform any computations after calling itself recursively.

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

fun length' (L: int list): int = tlength (L, 0)
```

Improved space efficiency

```
fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

fun length' (L: int list): int = tlength (L, 0)
```

Improved space efficiency

```
fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

fun length' (L: int list): int = tlength (L, 0)


length’ [3,4,5] ==> tlength ([3,4,5], 0]
                ==> tlength ([4,5], 1+0)
                ==> tlength ([4,5], 1)
                ==> tlength ([5], 1+1)
                ==> tlength ([5], 2)
                ==> tlength ([], 2+1)
                ==> tlength ([], 3)
                ==> 3
```

Improved space efficiency

```
fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

fun length' (L: int list): int = tlength (L, 0)
```

```
length’ [3,4,5] ==> tlength ([3,4,5], 0]
                ==> tlength ([4,5], 1+0)
                ==> tlength ([4,5], 1)
                ==> tlength ([5], 1+1)
                ==> tlength ([5], 2)
                ==> tlength ([], 2+1)
                ==> tlength ([], 3)
                ==> 3
```

time

Improved space efficiency

```
fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

fun length' (L: int list): int = tlength (L, 0)
```

space

```
length’ [3,4,5] ==> tlength ([3,4,5], 0]
                ==> tlength ([4,5], 1+0)
                ==> tlength ([4,5], 1)
                ==> tlength ([5], 1+1)
                ==> tlength ([5], 2)
                ==> tlength ([], 2+1)
                ==> tlength ([], 3)
                ==> 3
```

time

Improved space efficiency

```
fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

fun length' (L: int list): int = tlength (L, 0)
```

space

```
length’ [3,4,5] ==> tlength ([3,4,5], 0]
                ==> tlength ([4,5], 1+0)
                ==> tlength ([4,5], 1)
                ==> tlength ([5], 1+1)
                ==> tlength ([5], 2)
                ==> tlength ([], 2+1)
                ==> tlength ([], 3)
                ==> 3
```

time

recursive
calls without pending
computations

Let's prove space-efficient version correct!

Let’s prove space-efficient version correct!

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Let’s prove space-efficient version correct!

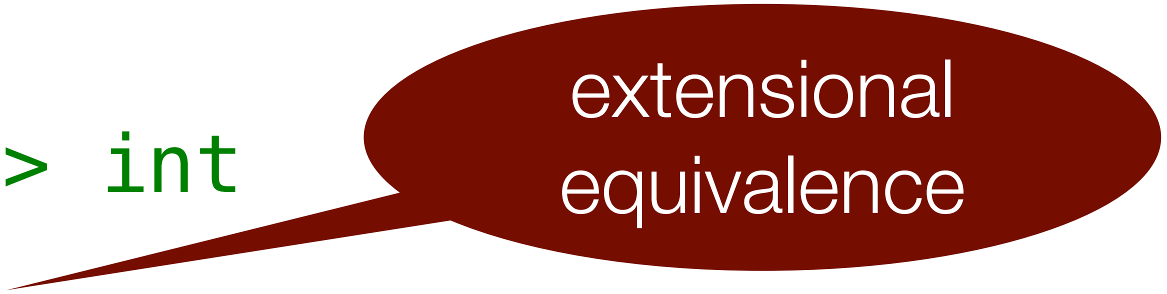
```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Let's prove space-efficient version correct!

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```



Let's prove space-efficient version correct!

extensional equivalence

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Theorem: For all values `L: int list` and `acc: int`,
`tlength(L,acc)` ≅ `(length L) + acc`.

Let's prove space-efficient version correct!

```
(* tlength : int list * int -> int
   REQUIRES: true
   ENSURES: tlength(L, acc) == length(L) + acc
*)

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

extensional equivalence

Theorem: For all values `L: int list` and `acc: int`, `tlength(L,acc)` ≅ `(length L) + acc`.

Before doing the proof, let's review extensional equivalence!

Extensional equivalence, revisited

Extensional equivalence, revisited

Definition:

- **Expression** of type `int` are **extensionally equivalent**,
  - if they evaluate to the same integer, or
  - if they both loop forever, or
  - if they both raise the same exception.
- **Functions** of type `int -> int` are **extensionally equivalent**, if they yield extensionally equivalent results, **if** applied to extensionally equivalent arguments.

Extensional equivalence, revisited

Definition:

- **Expression** of type `int` are **extensionally equivalent**,
  - if they evaluate to the same integer, or
  - if they both loop forever, or
  - if they both raise the same exception.
- **Functions** of type `int -> int` are **extensionally equivalent**, if they yield extensionally equivalent results, **if** applied to extensionally equivalent arguments.

Facts:

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$.

Extensional equivalence, revisited

Facts:

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$.

Extensional equivalence, revisited

Facts:

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$.

However:

From $e_1 \cong e_2$ it does **not** necessarily follow that
$e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$!

Extensional equivalence, revisited

Facts:

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$.

However:

From $e_1 \cong e_2$ it does **not** necessarily follow that $e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$!

eg, $1+1+7 \cong 9$

Extensional equivalence, revisited

Facts:

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$.

However:

From $e_1 \cong e_2$ it does **not** necessarily follow that
$e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$!
eg, `1+1+7` $\cong$ `9`

Moreover:

For f: `int -> int`, it does **not** necessarily follow that
`f(1) + f(2)` $\cong$ `f(2) + f(1)`!

Extensional equivalence, revisited

Facts:

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, then $e_1 \cong e_2$.
- If $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$.

However:

From $e_1 \cong e_2$ it does **not** necessarily follow that $e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$!

eg, $1+1+7 \cong 9$

f may loop or raise an exception (on some inputs)

Moreover:

For f: `int -> int`, it does **not** necessarily follow that `f(1) + f(2)` $\cong$ `f(2) + f(1)`!

Let’s prove space-efficient version correct!

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Theorem: For all values `L: int list` and `acc: int`, `tlength(L,acc)` $\cong$ `(length L) + acc`.

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Theorem: For all values `L: int list` and `acc: int`, `tlength(L,acc)` ≅ `(length L) + acc`.

Proof: By ???

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Theorem: For all values `L: int list` and `acc: int`,
`tlength(L,acc)` ≅ `(length L) + acc`.

Proof: By structural induction on L.

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Base case: L = `[]`.

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Base case: L = `[]`.

Need to show: for all values `acc:int`,

`tlength([],acc)` $\cong$ `length ([]) + acc`.

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Base case: L = `[]`.

Need to show: for all values `acc:int`,

`tlength([],acc)` $\cong$ `length ([]) + acc`.

Showing:

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Base case: L = `[]`.

Need to show: for all values `acc:int`,

`tlength([],acc)` ≅ `length ([]) + acc`.

Showing:

Evaluating the left side:

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Base case: L = `[]`.

Need to show: for all values `acc:int`,

`tlength([],acc)` ≅ `length ([]) + acc`.

Showing:

Evaluating the left side:

`tlength ([], acc)`

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Base case: L = `[]`.

Need to show: for all values `acc:int`,

`tlength([],acc)` $\cong$ `length ([]) + acc`.

Showing:

Evaluating the left side:

```
    tlength ([], acc)
⟹  acc
```

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Proof: By structural induction on L.

Base case: L = `[]`.

Need to show: for all values `acc:int`,

`tlength([],acc)` ≅ `length ([]) + acc`.

Showing:

Evaluating the left side:

`tlength ([], acc)`

⟹ `acc` (1st clause of `tlength`)

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

Evaluating the left side:

$\quad$ `tlength ([], acc)`

$\Longrightarrow$ `acc` (1st clause of `tlength`)

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

Evaluating the left side:

```
    tlength ([], acc)
⟹  acc
```

(1st clause of `tlength`)

Evaluating the right side:

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

<u>Showing</u>:

Evaluating the left side:

    tlength ([], acc)
$\Longrightarrow$ `acc` (1st clause of `tlength`)

Evaluating the right side:

    length([]) + acc

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

<u>Showing</u>:

Evaluating the left side:

`tlength ([], acc)`

$\Longrightarrow$ `acc` (1st clause of `tlength`)

Evaluating the right side:

`length([]) + acc`

$\Longrightarrow$ `0 + acc`

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

Evaluating the left side:

`tlength ([], acc)`

$\Longrightarrow$ `acc` (1st clause of `tlength`)

Evaluating the right side:

`length([]) + acc`

$\Longrightarrow$ `0 + acc` (1st clause of `length`)

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

<u>Showing</u>:

Evaluating the left side:

`tlength ([], acc)`

$\Longrightarrow$ `acc` (1st clause of `tlength`)

Evaluating the right side:

`length([]) + acc`

$\Longrightarrow$ `0 + acc` (1st clause of `length`)

$\Longrightarrow$ `acc`

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

<u>Showing</u>:

Evaluating the left side:

	`tlength ([], acc)`	
$\Longrightarrow$	`acc`	(1st clause of `tlength`)

Evaluating the right side:

	`length([]) + acc`	
$\Longrightarrow$	`0 + acc`	(1st clause of `length`)
$\Longrightarrow$	`acc`	(SML’s arithmetic)

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

<u>Showing</u>:

Evaluating the left side:

`tlength ([], acc)`

$\Longrightarrow$ `acc` (1st clause of `tlength`)

Evaluating the right side:

`length([]) + acc`

$\Longrightarrow$ `0 + acc` (1st clause of `length`)

$\Longrightarrow$ `acc` (SML's arithmetic)

equivalent because both reduce to the same value.

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

<u>IH</u>: for all values `acc’:int`,

`tlength(xs,acc’)` $\cong$ `length (xs) + acc’`.

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

IH: for all values `acc':int`,

`tlength(xs,acc')` $\cong$ `length (xs) + acc'`.

Need to show: for all values `acc:int`,

`tlength(x::xs,acc)` $\cong$ `length (x::xs) + acc`.

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

IH: for all values `acc':int`,

`tlength(xs,acc')` ≅ `length (xs) + acc'`.

Need to show: for all values `acc:int`,

`tlength(x::xs,acc)` ≅ `length (x::xs) + acc`.

Showing:

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

IH: for all values `acc’:int`,

`tlength(xs,acc’)` ≅ `length (xs) + acc’`.

Need to show: for all values `acc:int`,

`tlength(x::xs,acc)` ≅ `length (x::xs) + acc`.

Showing:

```
tlength(x::xs, acc)
```

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

IH: for all values `acc’:int`,

`tlength(xs,acc’)` ≅ `length (xs) + acc’`.

Need to show: for all values `acc:int`,

`tlength(x::xs,acc)` ≅ `length (x::xs) + acc`.

Showing:

```
  tlength(x::xs, acc)
≅ tlength(xs, 1 + acc)
```

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

IH: for all values `acc':int`,

`tlength(xs,acc')` $\cong$ `length (xs) + acc'`.

Need to show: for all values `acc:int`,

`tlength(x::xs,acc)` $\cong$ `length (x::xs) + acc`.

Showing:

`tlength(x::xs, acc)`

$\cong$ `tlength(xs, 1 + acc)` (step, 2nd clause of `tlength`)

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

IH: for all values `acc’:int`,

`tlength(xs,acc’)` ≅ `length (xs) + acc’`.

Need to show: for all values `acc:int`,

`tlength(x::xs,acc)` ≅ `length (x::xs) + acc`.

Showing:

`tlength(x::xs, acc)`

≅ `tlength(xs, 1 + acc)` (step, 2nd clause of `tlength`)

≅ `length(xs) + (1 + acc)`

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

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

IH: for all values `acc':int`,

`tlength(xs,acc')` ≅ `length (xs) + acc'`.

Need to show: for all values `acc:int`,

`tlength(x::xs,acc)` ≅ `length (x::xs) + acc`.

Showing:

	`tlength(x::xs, acc)`	
≅	`tlength(xs, 1 + acc)`	(step, 2nd clause of `tlength`)
≅	`length(xs) + (1 + acc)`	(IH, assume + total)

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

<u>Showing</u>:

`tlength(x::xs, acc)`

≅ `tlength(xs, 1 + acc)` (step, 2nd clause of `tlength`)

≅ `length(xs) + (1 + acc)` (IH, assume + total)

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

$\quad$ `tlength(x::xs, acc)`

$\cong$ `tlength(xs, 1 + acc)` (step, 2nd clause of `tlength`)

$\cong$ `length(xs) + (1 + acc)` (IH, assume **+** total)

$\cong$ `(1 + length(xs)) + acc`

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

	`tlength(x::xs, acc)`	
$\cong$	`tlength(xs, 1 + acc)`	(step, 2nd clause of `tlength`)
$\cong$	`length(xs) + (1 + acc)`	(IH, assume `+` total)
$\cong$	`(1 + length(xs)) + acc`	(`+` commutative, associative, totality of `length`)

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

	`tlength(x::xs, acc)`	
$\cong$	`tlength(xs, 1 + acc)`	(step, 2nd clause of `tlength`)
$\cong$	`length(xs) + (1 + acc)`	(IH, assume **+** total)
$\cong$	`(1 + length(xs)) + acc`	(**+** commutative, associative, totality of `length`)
$\cong$	`length(x::xs) + acc`	

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

<u>Showing</u>:

	`tlength(x::xs, acc)`	
$\cong$	`tlength(xs, 1 + acc)`	(step, 2nd clause of `tlength`)
$\cong$	`length(xs) + (1 + acc)`	(IH, assume + total)
$\cong$	`(1 + length(xs)) + acc`	(+ commutative, associative, totality of `length`)
$\cong$	`length(x::xs) + acc`	(step, 2nd clause of `length`)

Let’s prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

	`tlength(x::xs, acc)`	
$\cong$	`tlength(xs, 1 + acc)`	(step, 2nd clause of `tlength`)
$\cong$	`length(xs) + (1 + acc)`	(IH, assume `+` total)
$\cong$	`(1 + length(xs)) + acc`	(`+` commutative, associative, totality of `length`)
$\cong$	`length(x::xs) + acc`	(step, 2nd clause of `length`)

because: if $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

	`tlength(x::xs, acc)`	
$\cong$	`tlength(xs, 1 + acc)`	(step, 2nd clause of `tlength`)
$\cong$	`length(xs) + (1 + acc)`	(IH, assume **+** total)
$\cong$	`(1 + length(xs)) + acc`	(**+** commutative, associative, totality of `length`)
$\cong$	`length(x::xs) + acc`	(step, 2nd clause of `length`)

Let's prove space-efficient version correct!

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

fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)
```

Showing:

	`tlength(x::xs, acc)`	
$\cong$	`tlength(xs, 1 + acc)`	(step, 2nd clause of `tlength`)
$\cong$	`length(xs) + (1 + acc)`	(IH, assume + total)
$\cong$	`(1 + length(xs)) + acc`	(+ commutative, associative, totality of `length`)
$\cong$	`length(x::xs) + acc`	(step, 2nd clause of `length`)

Proved by structural induction on lists.

Appending lists

Appending lists

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)
```

Appending lists

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)
```

Appending lists

already predefined!

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)
```

Appending lists

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)
```

Appending lists

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)

val [] : int list = append([],[])
val [1,2] = append([], [1,2])
val [1,2,5,6] = append([1,2], [5,6])
```

Appending lists

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)

val [] : int list = append([],[])
val [1,2] = append([], [1,2])
val [1,2,5,6] = append([1,2], [5,6])
```

What is the time complexity of append?

Appending lists

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)

val [] : int list = append([],[])
val [1,2] = append([], [1,2])
val [1,2,5,6] = append([1,2], [5,6])
```

What is the time complexity of append?

Appending lists

```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

fun append ([] : int list, R : int list) : int list = R
  | append (x::xs, R) = x::append(xs,R)

val [] : int list = append([],[])
val [1,2] = append([], [1,2])
val [1,2,5,6] = append([1,2], [5,6])
```

What is the time complexity of append?

append(X,Y) has time complexity O(|X|).

Reversing a list

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)
```

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)
```

```
val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
```

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)

fun rev ([] : int list) : int list =




val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
```

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)

fun rev ([] : int list) : int list = []



val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
```

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)

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


val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
```

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)

fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]


val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
```

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)

fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]


val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
```

What is the time complexity of reverse?

Reversing a list

```
(* rev: int list -> int list
   REQUIRES: true
   ENSURES: rev L returns the elements of L
            in reverse order.
*)

fun rev ([] : int list) : int list = []
  | rev (x::xs) = (rev xs) @ [x]


val [] : int list = rev []
val [4,3,2,1] : int list = rev [1,2,3,4]
```

What is the time complexity of reverse?

reverse(X) has time complexity $O(|X|^2)$.

Reversing a list: can we do better?

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:
*)
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:
*)
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:
*)
```

accumulator

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:
*)
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) ==
*)
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)


fun trev ([] : int list, acc : int list) : int list =
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)


fun trev ([] : int list, acc : int list) : int list = acc
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)


fun trev ([] : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) =
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)


fun trev ([] : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) = trev(xs, x::acc)
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)


fun trev ([] : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) = trev(xs, x::acc)

fun reverse (L : int list) : int list = trev(L, [])
```

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)
```

```
fun trev ([] : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) = trev(xs, x::acc)

fun reverse (L : int list) : int list = trev(L, [])
```

What is the time complexity of trev?

Reversing a list: can we do better?

```
(* trev : int list * int list -> int list
   REQUIRES: true
   ENSURES:  trev(L, acc) == (rev L) @ acc
*)


fun trev ([] : int list, acc : int list) : int list = acc
  | trev (x::xs, acc) = trev(xs, x::acc)

fun reverse (L : int list) : int list = trev(L, [])
```

What is the time complexity of trev?

tref(X, Y) has time complexity O(|X|).

Are reverse and rev extensional equivalent?

Are reverse and rev extensional equivalent?

Theorem: For all values `L: int list` and `acc: int list`, `trev(L,acc)` $\cong$ `(rev L) @ acc`.

Are reverse and rev extensional equivalent?

Theorem: For all values `L: int list` and `acc: int list`, `trev(L,acc)` $\cong$ `(rev L) @ acc`.

Prove this theorem as an exercise!

Are reverse and rev extensional equivalent?

Theorem: For all values `L: int list` and `acc: int list`,
`trev(L,acc)` $\cong$ `(rev L) @ acc`.

- Prove this theorem as an exercise!
- We provide the solution in the notes (rev.pdf). But try first!

That's all for today. Have a good weekend!