Tail recursion and more about lists, structural induction and extensional equivalence

15-150

Lecture 4: September 5, 2024

Stephanie Balzer
Carnegie Mellon University

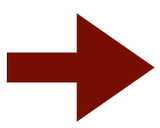
Let’s reconsider our length function

Let's reconsider our length function

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

Let’s reconsider our length function

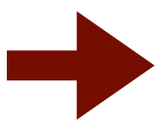
```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```



Is this function efficient?

Let's reconsider our length function

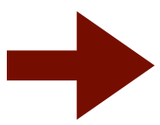
```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```



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)
```

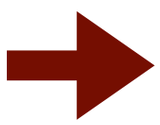


Is this function efficient? Well, let's look at an evaluation trace:

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

Let's reconsider our length function

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

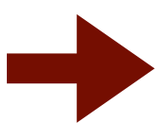


Is this function efficient? Well, let's look at an evaluation trace:

```
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)
```

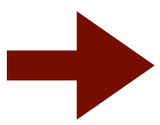
Is this function efficient? Well, let's look at an evaluation trace:

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

[…, 3/x, [4,5]/xs] 1 + length(xs)

Let's reconsider our length function

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

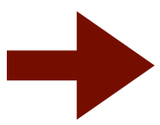


Is this function efficient? Well, let's look at an evaluation trace:

```
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)
```

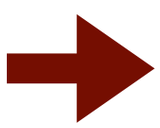


Is this function efficient? Well, let's look at an evaluation trace:

```
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)
```

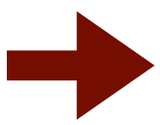


Is this function efficient? Well, let's look at an evaluation trace:

```
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)
```

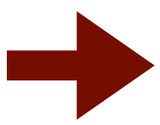


Is this function efficient? Well, let's look at an evaluation trace:

```
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)
```

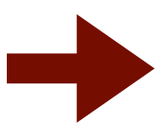


Is this function efficient? Well, let's look at an evaluation trace:

```
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)
```

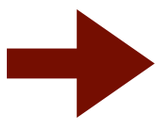


Is this function efficient? Well, let's look at an evaluation trace:

```
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)
```

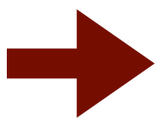


Is this function efficient? Well, let's look at an evaluation trace:

```
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:

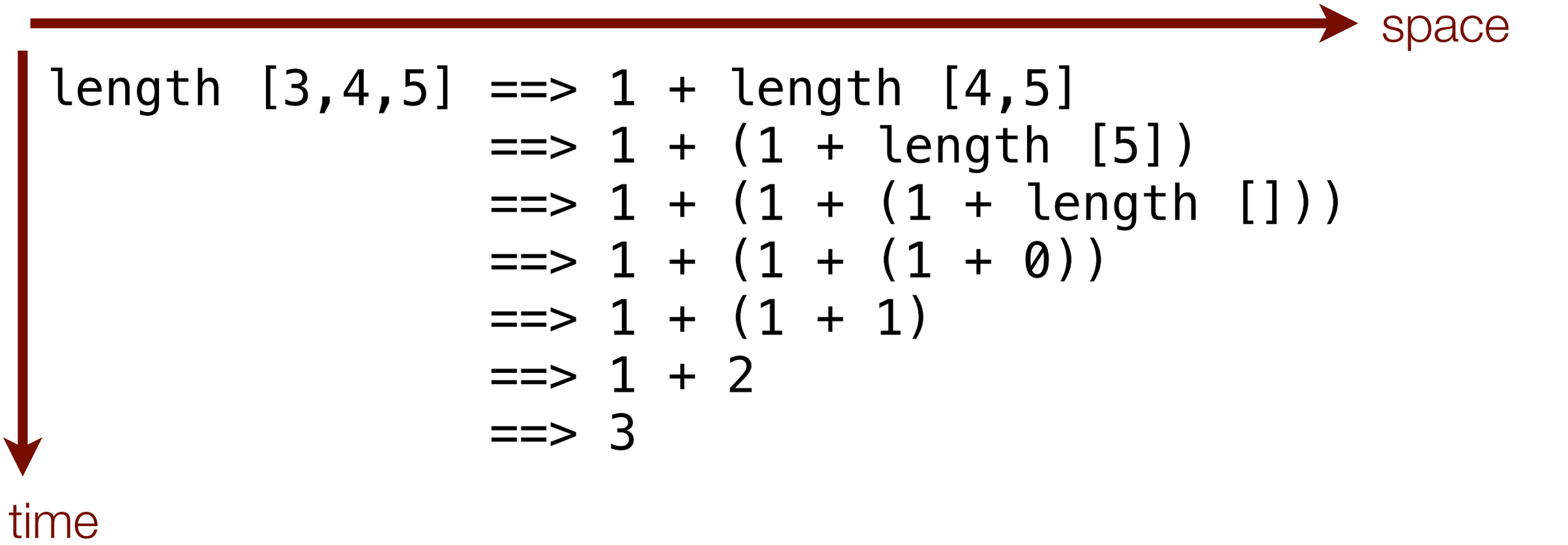
```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let's reconsider our length function

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

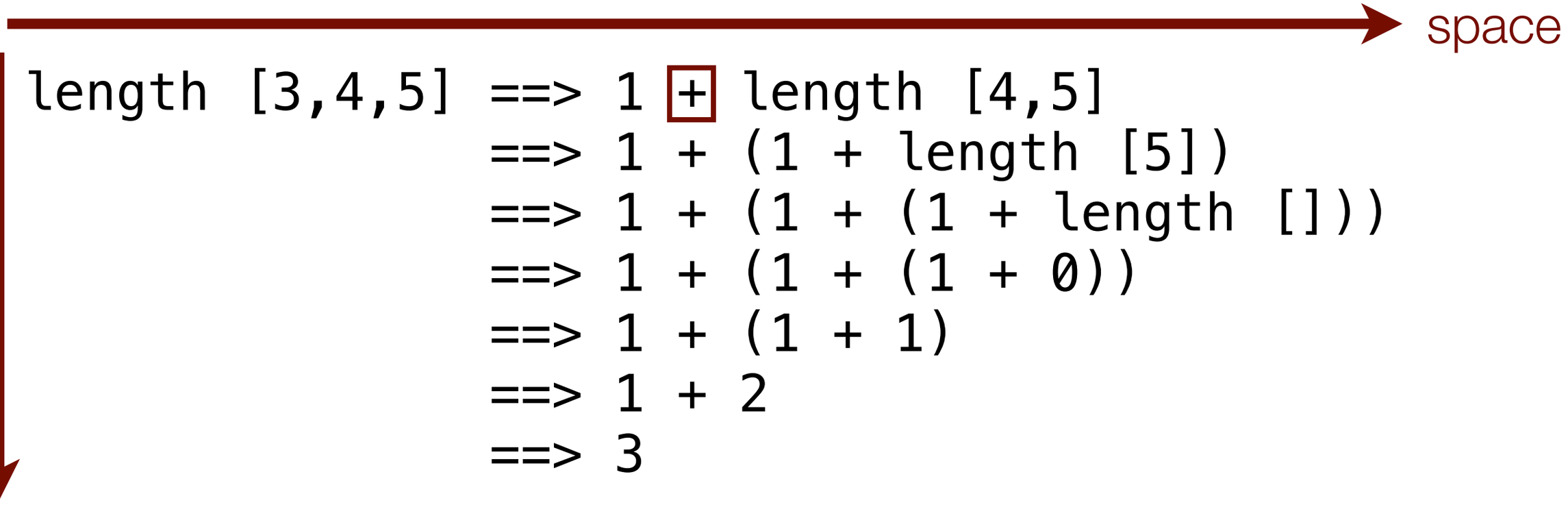
Is this function efficient? Well, let's look at an evaluation trace:



Let's reconsider our length function

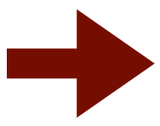
```
fun length ([] : int list) : int = 0
  | length (x::xs) = 1 + length(xs)
```

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:

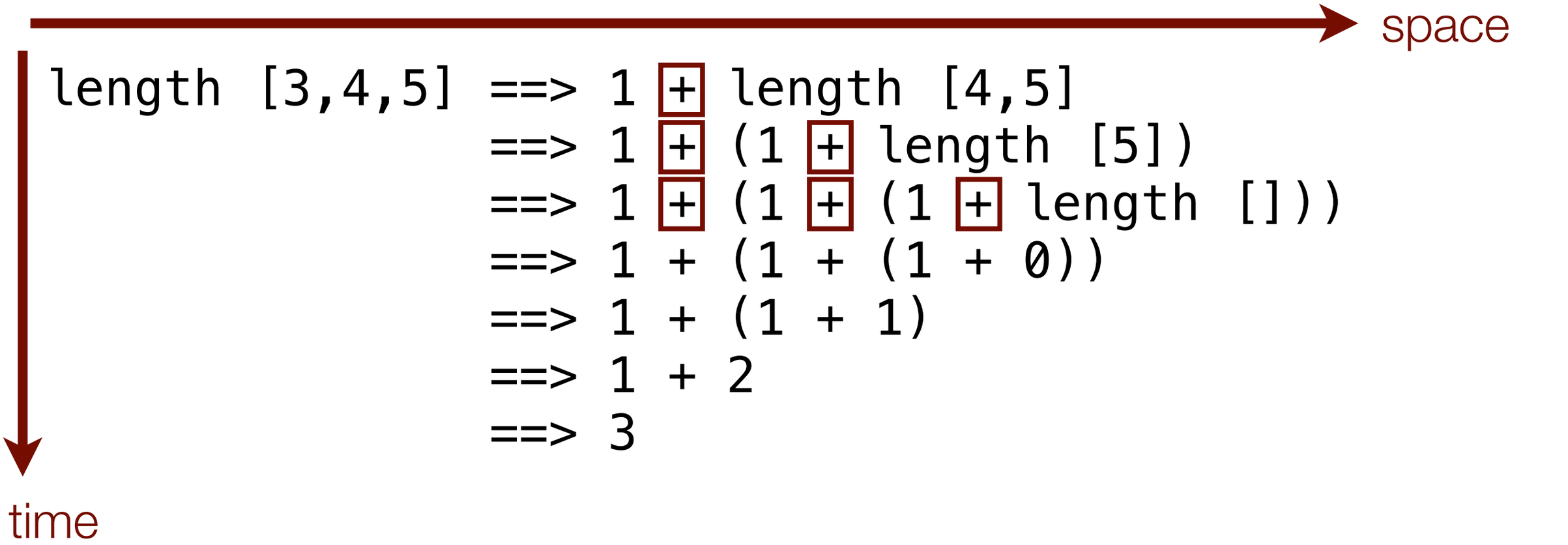
```
length [3,4,5] ==> 1 + length [4,5]
               ==> 1 + (1 + length [5])
               ==> 1 + (1 + (1 + length []))
               ==> 1 + (1 + (1 + 0))
               ==> 1 + (1 + 1)
               ==> 1 + 2
               ==> 3
```

time

Let's reconsider our length function

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

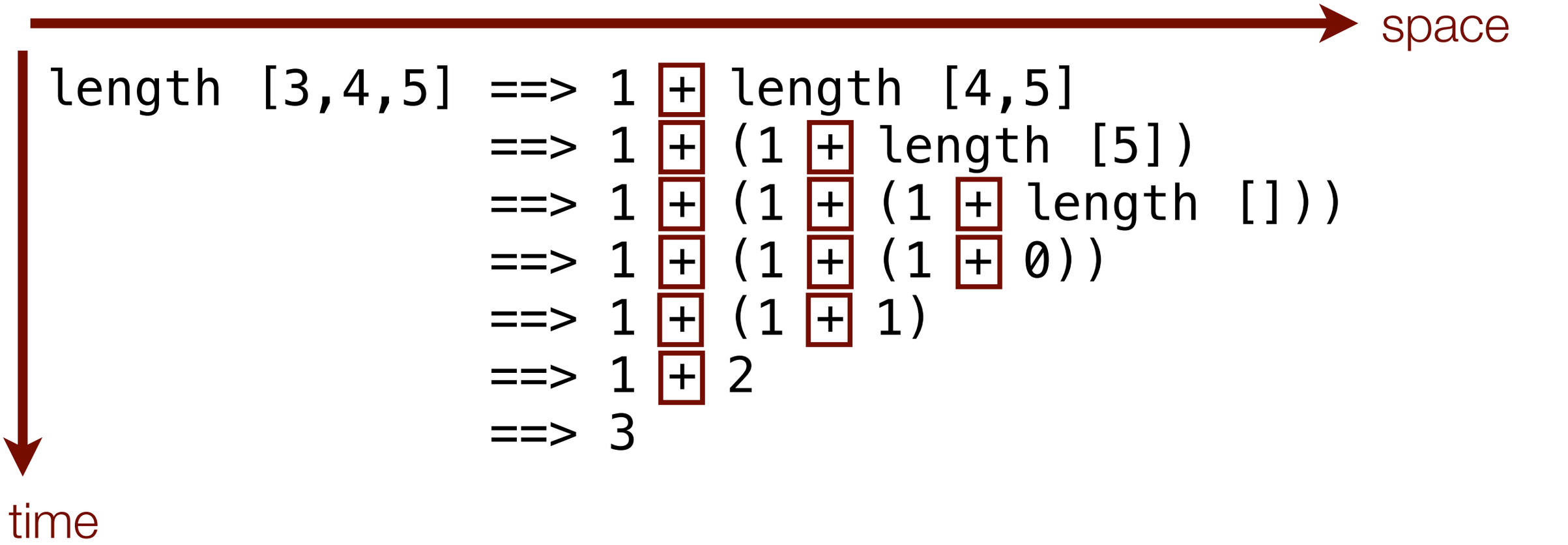
Is this function efficient? Well, let's look at an evaluation trace:



Let's reconsider our length function

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

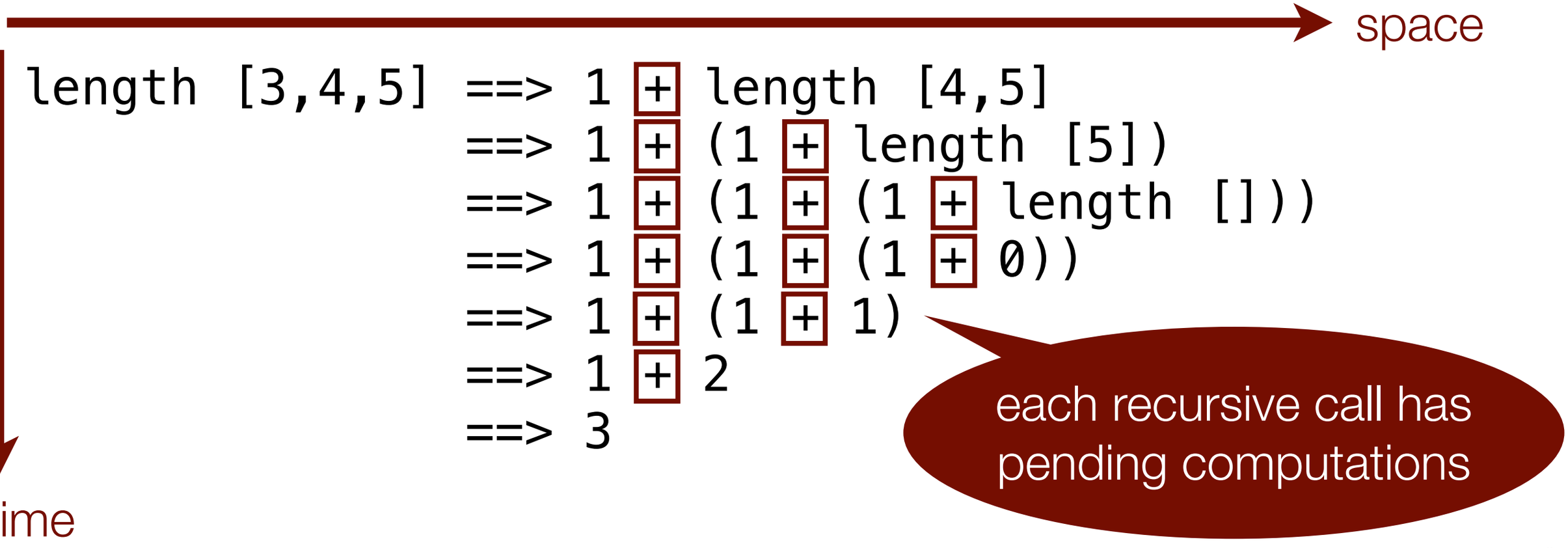
Is this function efficient? Well, let's look at an evaluation trace:



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:



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.

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

extensional equivalence

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
*)
```

extensional equivalence

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
*)
```

extensional equivalence

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)
```

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

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

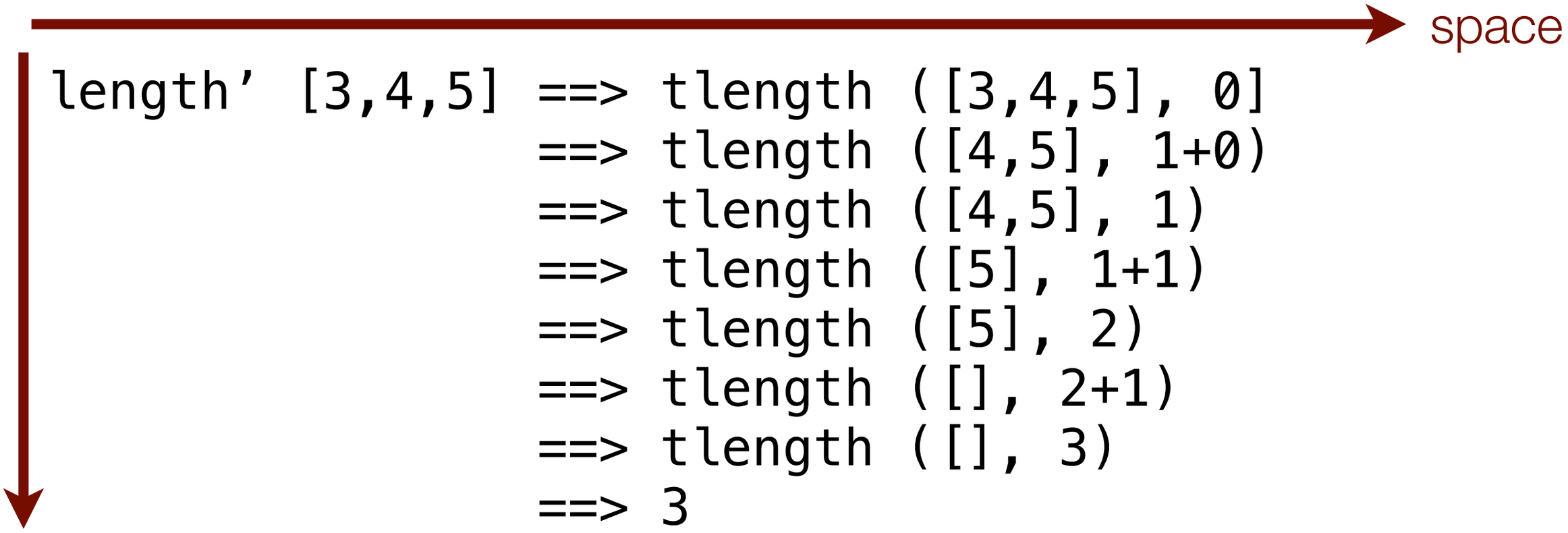
```
fun tlength([]: int list, acc: int): int = acc
  | tlength(x::xs, acc) = tlength(xs, 1 + acc)

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)
```




```
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)
```

space

time

```
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
```

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
*)
```

extensional equivalence

```
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
*)
```

extensional equivalence

```
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
*)
```

extensional equivalence

```
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`.

→ 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 map extensionally equivalent arguments to extensionally equivalent results.

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 map extensionally equivalent arguments to extensionally equivalent results.

Facts:

- If $\mathtt{e}_1 \hookrightarrow \mathtt{v}$ and $\mathtt{e}_2 \hookrightarrow \mathtt{v}$, then $\mathtt{e}_1 \cong \mathtt{e}_2$.
- If $\mathtt{e}_1 \Longrightarrow \mathtt{e}$ and $\mathtt{e}_2 \Longrightarrow \mathtt{e}$, then $\mathtt{e}_1 \cong \mathtt{e}_2$.
- If $\mathtt{e}_1 \Longrightarrow \mathtt{e}_2$, then $\mathtt{e}_1 \cong \mathtt{e}_2$.

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`

Moreover:

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

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

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

Showing:

Evaluating the left side:

```
    tlength ([], acc)
```

$\Longrightarrow$ `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)
```

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:

`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)`

$\Longrightarrow$ `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)
```

Showing:

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)
```

Showing:

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)
```

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`)

$\Longrightarrow$ `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)
```

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`)

$\Longrightarrow$ `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)
```

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`)

$\Longrightarrow$ `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`.

IH: for all values `acc':int`,
`tlength(xs,acc')` ≅ `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')` ≅ `length (xs) + acc'`.

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

`tlength(x::xs,acc)` ≅ `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')` ≅ `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)
```

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)
```

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)
```

Showing:

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

Showing:

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

→ Proved by structural induction on lists.

Appending lists

Appending lists

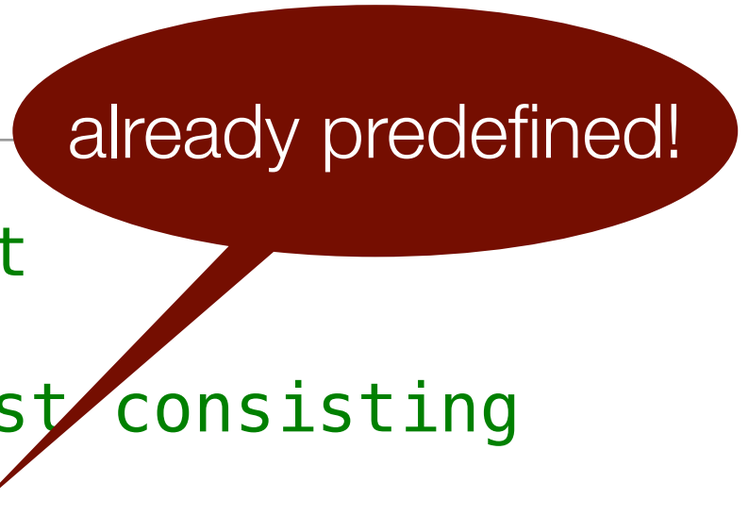
```
(* append: int list * int list -> int list
   REQUIRES: true
   ENSURES: append(L,R) evaluates to a list consisting
            of L followed by R
   NOTE: predefined in SML as the right-associative
         infix operator @.
*)

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




```
(* 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:
*)
```

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|).

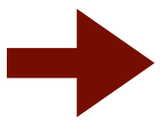
Are reverse and rev extensional equivalent?

Are reverse and rev extensional equivalent?

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

Are reverse and rev extensional equivalent?

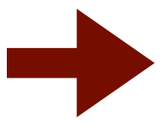
Theorem: For all values `L: int list` and `acc: int list`, `trev(L,acc)` ≅ `(rev L) @ acc`.



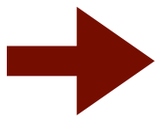
Prove this theorem as an exercise!

Are reverse and rev extensional equivalent?

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



Prove this theorem as an exercise!



We provide the solution in the notes (rev.pdf). But try first!