Script generated by TTT

Title: Nipkow: Info2 (23.01.2015)

Date: Fri Jan 23 08:31:27 CET 2015

Duration: 86:44 min

Pages: 117

How to analyze and improve the time (and space) complexity of functional programs

How to analyze and improve the time (and space) complexity of functional programs

Based largely on Richard Bird's book
Introduction to Functional Programming using Haskell.

Assumption in this section:

How to analyze and improve the time (and space) complexity of functional programs

Based largely on Richard Bird's book
Introduction to Functional Programming using Haskell.

Assumption in this section:

Reduction strategy is innermost (call by value, cbv)

How to analyze and improve the time (and space) complexity
of functional programs

Based largely on Richard Bird's book
Introduction to Functional Programming using Haskell.

Assumption in this section:

Reduction strategy is innermost (call by value, cbv)

- Analysis much easier

How to analyze and improve the time (and space) complexity
of functional programs

Based largely on Richard Bird's book
Introduction to Functional Programming using Haskell.

Assumption in this section:

Reduction strategy is innermost (call by value, cbv)

- Analysis much easier
- Most languages follow cbv

How to analyze and improve the time (and space) complexity
of functional programs

Based largely on Richard Bird's book
Introduction to Functional Programming using Haskell.

Assumption in this section:

Reduction strategy is innermost (call by value, cbv)

- Analysis much easier
- Most languages follow cbv
- Number of lazy evaluation steps $\leq$ number of cbv steps

How to analyze and improve the time (and space) complexity
of functional programs

Based largely on Richard Bird's book
Introduction to Functional Programming using Haskell.

Assumption in this section:

Reduction strategy is innermost (call by value, cbv)

- Analysis much easier
- Most languages follow cbv
- Number of lazy evaluation steps $\leq$ number of cbv steps
  $\Longrightarrow$ O-analysis under cbv also correct for Haskell
  but can be too pessimistic

13.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

13.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

(No guards on the left-hand side of an equation,
`if-then-else` on the righ-hand side instead)

13.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

(No guards on the left-hand side of an equation,
`if-then-else` on the righ-hand side instead)

Justification:

The implementation does not copy data structures
but works with pointers and sharing

13.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

(No guards on the left-hand side of an equation,
`if-then-else` on the righ-hand side instead)

Justification:

The implementation does not copy data structures
but works with pointers and sharing

Example: `length (_ : xs) = length xs + 1`

13.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

(No guards on the left-hand side of an equation,
`if-then-else` on the righ-hand side instead)

Justification:

The implementation does not copy data structures
but works with pointers and sharing

Example: `length (_ : xs) = length xs + 1`
Reduce `length [1,2,3]`

13.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

(No guards on the left-hand side of an equation,
`if-then-else` on the righ-hand side instead)

Justification:

The implementation does not copy data structures
but works with pointers and sharing

Example: `length (_ : xs) = length xs + 1`
Reduce `length [1,2,3]`

Compare:

```
id [] = []
id (x:xs) = x : id xs
```


13.1 Time complexity analysis

Basic assumption:

One reduction step takes one time unit

(No guards on the left-hand side of an equation,
`if-then-else` on the righ-hand side instead)

Justification:

The implementation does not copy data structures
but works with pointers and sharing

Example: `length (_ : xs) = length xs + 1`
Reduce `length [1,2,3]`

Compare:

```
id [] = []
id (x:xs) = x : id xs
```


Reduce `id [e1,e2]`
Copies list but shares elements.

$T_{\texttt{f}}(n) =$ number of steps required for the evaluation of `f` when applied to an argument of size n

$T_{\texttt{f}}(n) =$ number of steps required for the evaluation of `f` when applied to an argument of size n in the worst case

What is "size"?

- Number of bits. Too low level.

$T_{\texttt{f}}(n) =$ number of steps required for the evaluation of `f` when applied to an argument of size n in the worst case

What is "size"?

- Number of bits. Too low level.
- Better:

$T_{\texttt{f}}(n) =$ number of steps required for the evaluation of `f` when applied to an argument of size n in the worst case

What is "size"?

- Number of bits. Too low level.
- Better: specific measure based on the argument type of `f`

$T_{\texttt{f}}(n) =$ number of steps required for the evaluation of `f` when applied to an argument of size n in the worst case

What is "size"?

- Number of bits. Too low level.
- Better: specific measure based on the argument type of `f`
- Measure may differ from function to function.

$T_{\mathtt{f}}(n)$ = number of steps required for the evaluation of `f` when applied to an argument of size n in the worst case

What is "size"?

- Number of bits. Too low level.
- Better: specific measure based on the argument type of `f`
- Measure may differ from function to function.
- Frequent measure for functions on lists: the length of the list
  We use this measure unless stated otherwise

$T_{\mathtt{f}}(n)$ = number of steps required for the evaluation of `f` when applied to an argument of size n in the worst case

What is "size"?

- Number of bits. Too low level.
- Better: specific measure based on the argument type of `f`
- Measure may differ from function to function.
- Frequent measure for functions on lists: the length of the list
  We use this measure unless stated otherwise
  Sufficient if `f` does not compute with the elements of the list

---

$T_{\mathtt{f}}(n)$ = number of steps required for the evaluation of `f` when applied to an argument of size n in the worst case

What is "size"?

- Number of bits. Too low level.
- Better: specific measure based on the argument type of `f`
- Measure may differ from function to function.
- Frequent measure for functions on lists: the length of the list
  We use this measure unless stated otherwise
  Sufficient if `f` does not compute with the elements of the list
  Not sufficient for function ...

How to calculate (not mechanically!) $T_{\mathtt{f}}(n)$:

1. From the equations for `f` derive equations for $T_{\mathtt{f}}$

How to calculate (not mechanically!) $T_{\texttt{f}}(n)$:

1. From the equations for `f` derive equations for $T_{\texttt{f}}$
2. If the equations for $T_{\texttt{f}}$ are recursive, solve them

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```


Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```


$$T_{++}(0, n) = O(1)$$

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$

$$T_{++}(m+1, n) =$$

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$
$$T_{++}(m+1, n) = T_{++}(m, n) + O(1)$$

$$\implies T_{++}(m, n) = O(m)$$

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$
$$T_{++}(m+1, n) = T_{++}(m, n) + O(1)$$

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$
$$T_{++}(m+1, n) = T_{++}(m, n) + O(1)$$

$$\implies T_{++}(m, n) = O(m)$$

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

$$T_{++}(0, n) = O(1)$$
$$T_{++}(m+1, n) = T_{++}(m, n) + O(1)$$

$$\implies T_{++}(m, n) = O(m)$$

Note: (++) creates copy of first argument

Example

```
[] ++ ys       =  ys
(x:xs) ++ ys   =  x : (xs ++ ys)
```

$$\begin{aligned} T_{++}(0, n) &= O(1) \\ T_{++}(m+1, n) &= T_{++}(m, n) + O(1) \end{aligned}$$

$$\Longrightarrow T_{++}(m, n) = O(m)$$

Note: (++) creates copy of first argument

Principle:

Every constructor of an algebraic data type takes time $O(1)$.
A constant amount of space needs to be allocated.

Example

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$\begin{aligned} T_{\texttt{reverse}}(0) &= O(1) \\ T_{\texttt{reverse}}(n+1) &= \end{aligned}$$

Example

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$\begin{aligned} T_{\texttt{reverse}}(0) &= O(1) \\ T_{\texttt{reverse}}(n+1) &= T_{\texttt{reverse}}(n) \end{aligned}$$

Example

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$\begin{aligned} T_{\texttt{reverse}}(0) &= O(1) \\ T_{\texttt{reverse}}(n+1) &= T_{\texttt{reverse}}(n) + T_{++}(n, 1) \end{aligned}$$

Example

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$T_{\texttt{reverse}}(0) = O(1)$$
$$T_{\texttt{reverse}}(n+1) = T_{\texttt{reverse}}(n) + T_{\texttt{++}}(n,1)$$

$$\Longrightarrow T_{\texttt{reverse}}(n) = O(n^2)$$

The worst case time complexity of an expression e:

Sum up all $T_{\texttt{f}}(n_1, ..., n_k)$

The worst case time complexity of an expression e:

Sum up all $T_{\texttt{f}}(n_1, ..., n_k)$
where $f\ e_1\ \dots e_n$ is a function call in e

The worst case time complexity of an expression e:

Sum up all $T_{\texttt{f}}(n_1, ..., n_k)$
where $f\ e_1\ \dots e_n$ is a function call in e
and n_i is the size of e_i

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, ..., n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

(assumption: no higher-order functions)

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, ..., n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

(assumption: no higher-order functions)

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$,

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, ..., n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

(assumption: no higher-order functions)

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$,
both for cbv and lazy evaluation. (Why?)

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, ..., n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

(assumption: no higher-order functions)

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$,
both for cbv and lazy evaluation. (Why?)

Consider `min xs = head(sort xs)`

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, \ldots, n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

(assumption: no higher-order functions)

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$, both for cbv and lazy evaluation. (Why?)

Consider `min xs = head(sort xs)`

$$T_{\mathtt{min}}(n) = T_{\mathtt{sort}}(n) + T_{\mathtt{head}}(n)$$

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, \ldots, n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

(assumption: no higher-order functions)

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$, both for cbv and lazy evaluation. (Why?)

Consider `min xs = head(sort xs)`

$$T_{\mathtt{min}}(n) = T_{\mathtt{sort}}(n) + T_{\mathtt{head}}(n)$$

For cbv also a lower bound, but not for lazy evaluation.

The worst case time complexity of an expression e:

Sum up all $T_{\mathtt{f}}(n_1, \ldots, n_k)$
where $f\ e_1\ \ldots e_n$ is a function call in e
and n_i is the size of e_i

(assumption: no higher-order functions)

Note: examples so far equally correct with $\Theta(.)$ instead of $O(.)$, both for cbv and lazy evaluation. (Why?)

Consider `min xs = head(sort xs)`

$$T_{\mathtt{min}}(n) = T_{\mathtt{sort}}(n) + T_{\mathtt{head}}(n)$$

For cbv also a lower bound, but not for lazy evaluation.

Complexity analysis is *compositional* under cbv

13.2 Optimizing functional programs

13.2 Optimizing functional programs

Premature optimization is the root of all evil

13.2 Optimizing functional programs

Premature optimization is the root of all evil
Don Knuth

But we are in week $n - 1$ now ;-)

The ideal of program optimization:

1. Write (possibly) inefficient but correct code
2. Optimize your code *and prove equivelence to correct version*

No duplication

Eliminate common subexpressions with `where` (or `let`)

No duplication

Eliminate common subexpressions with `where` (or `let`)

Example

~~`f x = g (h x) (h x)`~~

No duplication

Eliminate common subexpressions with `where` (or `let`)

Example

~~`f x = g (h x) (h x)`~~

```
f x = g y y  where  y = h x
```

Tail recursion / Endrekursion

The definition of a function `f` is tail recursive / endrekursiv

Tail recursion / Endrekursion

The definition of a function `f` is tail recursive / endrekursiv
if every recursive call is in "end position",

Tail recursion / Endrekursion

The definition of a function `f` is tail recursive / endrekursiv
if every recursive call is in "end position",
= it is the last function call before leaving `f`,
= nothing happens afterwards

Tail recursion / Endrekursion

The definition of a function `f` is tail recursive / endrekursiv
if every recursive call is in "end position",
= it is the last function call before leaving `f`,
= nothing happens afterwards
= no call of `f` is nested in another function call

Tail recursion / Endrekursion

The definition of a function `f` is tail recursive / endrekursiv
if every recursive call is in "end position",
= it is the last function call before leaving `f`,
= nothing happens afterwards
= no call of `f` is nested in another function call

Example

```
length []       =  0
length (x:xs)   =  length xs + 1
```

Tail recursion / Endrekursion

The definition of a function `f` is tail recursive / endrekursiv
if every recursive call is in "end position",
= it is the last function call before leaving `f`,
= nothing happens afterwards
= no call of `f` is nested in another function call

Example

```
length []       =  0
length (x:xs)   =  length xs + 1
```


```
length2 []      n  =  n
length2 (x:xs)  n  =  length2 xs (n+1)
```

```
length []       =  0
length (x:xs)   =  length xs + 1

length2 []      n  =  n
length2 (x:xs)  n  =  length2 xs (n+1)
```

```
length []       =  0
length (x:xs)   =  length xs + 1

length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

Compare executions:

```
length [a,b,c]
= length [b,c] + 1
= (length [c] + 1) + 1
= ((length [] + 1) + 1) + 1
= ((0 + 1) + 1) + 1
= 3

length2 [a,b,c] 0
```

```
length []       =  0
length (x:xs)   =  length xs + 1

length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

Compare executions:

```
length [a,b,c]
= length [b,c] + 1
= (length [c] + 1) + 1
= ((length [] + 1) + 1) + 1
= ((0 + 1) + 1) + 1
= 3

length2 [a,b,c] 0
= length2 [b,c] 1
= length2 [c]   2
= length2 []    3
```

```
length []       =  0
length (x:xs)   =  length xs + 1

length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

Compare executions:

```
length [a,b,c]
= length [b,c] + 1
= (length [c] + 1) + 1
= ((length [] + 1) + 1) + 1
= ((0 + 1) + 1) + 1
= 3

length2 [a,b,c] 0
= length2 [b,c] 1
= length2 [c]   2
= length2 []    3
= 3
```

Fact Tail recursive definitions can be compiled into loops.

Fact Tail recursive definitions can be compiled into loops.
Not just in functional languages.

Fact Tail recursive definitions can be compiled into loops.
Not just in functional languages.

No (additional) stack space is needed
to execute tail recursive functions

Fact Tail recursive definitions can be compiled into loops.
Not just in functional languages.

No (additional) stack space is needed
to execute tail recursive functions

Example

```
length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

Fact Tail recursive definitions can be compiled into loops.
Not just in functional languages.

No (additional) stack space is needed
to execute tail recursive functions

Example

```
length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```


$\leadsto$

```
loop: if null xs then return n
      xs := tail xs
      n := n+1
      goto loop
```

What does tail recursive mean for

f `x = if` b `then` e_1 `else` e_2

What does tail recursive mean for

f `x = if` b `then` e_1 `else` e_2

- f does not occur in b

What does tail recursive mean for

f `x = if` b `then` e_1 `else` e_2

- f does not occur in b
- if f occurs in e_i then only at the outside: $e_i = f \ldots$

What does tail recursive mean for

f `x = if` b `then` e_1 `else` e_2

- f does not occur in b
- if f occurs in e_i then only at the outside: $e_i = f \ldots$

Tail recursive example:

```
f x = if x > 0 then f(x-1) else f(x+1)
```

What does tail recursive mean for

f `x = if` b `then` e_1 `else` e_2

- f does not occur in b
- if f occurs in e_i then only at the outside: $e_i = f \ldots$

Tail recursive example:

```
f x = if x > 0 then f(x-1) else f(x+1)
```

Similar for guards and `case` e `of`:

- f does not occur in e
- if f occurs in any branch then only at the outside: $f \ldots$

Accumulating parameters

Accumulating parameters

An accumulating parameter is a parameter where intermediate results are accumulated.

```
length []       =  0
length (x:xs)   =  length xs + 1

length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

Compare executions:

```
length [a,b,c]
= length [b,c] + 1
= (length [c] + 1) + 1
= ((length [] + 1) + 1) + 1
= ((0 + 1) + 1) + 1
= 3

length2 [a,b,c] 0
= length2 [b,c] 1
= length2 [c]   2
= length2 []    3
= 3
```

Accumulating parameters

An accumulating parameter is a parameter where intermediate results are accumulated.
Purpose:

- tail recursion

Accumulating parameters

An accumulating parameter is a parameter where intermediate results are accumulated.
Purpose:

- tail recursion
- replace (++) by (:)

Accumulating parameters

An accumulating parameter is a parameter where intermediate results are accumulated.
Purpose:

- tail recursion
- replace (++) by (:)

```
length2 []     n  =  n
length2 (x:xs) n  =  length2 xs (n+1)
```

Accumulating parameter: reverse

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$T_{\texttt{reverse}}(n) = O(n^2)$$

```
itrev [] xs     = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

Accumulating parameter: `reverse`

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$T_{\texttt{reverse}}(n) = O(n^2)$$

```
itrev [] xs     = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

Not just tail recursive also linear:

$$\begin{aligned} T_{\texttt{itrev}}(0, n) &= O(1) \\ T_{\texttt{itrev}}(m+1, n) &= T_{\texttt{itrev}}(m, n) + O(1) \end{aligned}$$

Accumulating parameter: `reverse`

```
reverse []      =  []
reverse (x:xs)  =  reverse xs ++ [x]
```

$$T_{\texttt{reverse}}(n) = O(n^2)$$

```
itrev [] xs     = xs
itrev (x:xs) ys = itrev xs (x:ys)
```

Not just tail recursive also linear:

$$\begin{aligned} T_{\texttt{itrev}}(0, n) &= O(1) \\ T_{\texttt{itrev}}(m+1, n) &= T_{\texttt{itrev}}(m, n) + O(1) \end{aligned}$$

$$\Longrightarrow T_{\texttt{itrev}}(m, n) = O(m)$$

Accumulating parameter: tree flattening

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)
```

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` = 1)

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` = 1)

$$T_{\texttt{flat}}(1) = O(1)$$

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` = 1)

$$\begin{aligned} T_{\texttt{flat}}(1) &= O(1) \\ T_{\texttt{flat}}(h+1) &= 2 * T_{\texttt{flat}}(h) + \end{aligned}$$

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` = 1)

$$\begin{aligned} T_{\texttt{flat}}(1) &= O(1) \\ T_{\texttt{flat}}(h+1) &= 2 * T_{\texttt{flat}}(h) + T_{++} \end{aligned}$$

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` $= 1$)

$$\begin{array}{lcl} T_{\texttt{flat}}(1) & = & O(1) \\ T_{\texttt{flat}}(h+1) & = & 2 * T_{\texttt{flat}}(h) + T_{++}(2^h, 2^h) \end{array}$$

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` $= 1$)

$$\begin{array}{lcl} T_{\texttt{flat}}(1) & = & O(1) \\ T_{\texttt{flat}}(h+1) & = & 2 * T_{\texttt{flat}}(h) + T_{++}(2^h, 2^h) \\ & = & 2 * T_{\texttt{flat}}(h) + O(2^h) \end{array}$$

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` $= 1$)

$$\begin{array}{lcl} T_{\texttt{flat}}(1) & = & O(1) \\ T_{\texttt{flat}}(h+1) & = & 2 * T_{\texttt{flat}}(h) + T_{++}(2^h, 2^h) \\ & = & 2 * T_{\texttt{flat}}(h) + O(2^h) \end{array}$$

$$\Longrightarrow T_{\texttt{flat}}(h) = O(h * 2^h)$$

Accumulating parameter: tree flattening

```
data Tree a = Tip a | Node (Tree a) (Tree a)

flat (Tip a)       =  [a]
flat (Node t1 t2)  =  flat t1 ++ flat t2
```

Size measure: height of tree (height of `Tip` $= 1$)

$$\begin{array}{lcl} T_{\texttt{flat}}(1) & = & O(1) \\ T_{\texttt{flat}}(h+1) & = & 2 * T_{\texttt{flat}}(h) + T_{++}(2^h, 2^h) \\ & = & 2 * T_{\texttt{flat}}(h) + O(2^h) \end{array}$$

$$\Longrightarrow T_{\texttt{flat}}(h) = O(h * 2^h)$$

With accumulating parameter:

```
flat2 :: Tree a -> [a] -> [a]
```

Accumulating parameter: foldl

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)
```

Accumulating parameter: foldl

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr f z [x1,...,xn] = x1 `f` (... `f` (xn `f` z)...)
```

Tail recursive, second parameter accumulator:

Accumulating parameter: foldl

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr f z [x1,...,xn] = x1 `f` (... `f` (xn `f` z)...)
```

Tail recursive, second parameter accumulator:

```
foldl f z [] = z
foldl f z (x:xs) = foldl (f z x) xs
```

Accumulating parameter: foldl

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr f z [x1,...,xn] = x1 `f` (... `f` (xn `f` z)...)
```

Tail recursive, second parameter accumulator:

```
foldl f z [] = z
foldl f z (x:xs) = foldl (f z x) xs

foldl f z [x1,...,xn] = (...(z `f` x1) `f` ...) `f` xn
```

Relationship between `foldr` and `foldl`:

Accumulating parameter: foldl

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr f z [x1,...,xn] = x1 `f` (... `f` (xn `f` z)...)
```

Tail recursive, second parameter accumulator:

```
foldl f z [] = z
foldl f z (x:xs) = foldl (f z x) xs

foldl f z [x1,...,xn] = (...(z `f` x1) `f` ...) `f` xn
```

Relationship between `foldr` and `foldl`:

Lemma `foldl f e = foldr f e`

Accumulating parameter: foldl

```
foldr f z []     =  z
foldr f z (x:xs) =  f x (foldr f z xs)

foldr f z [x1,...,xn] = x1 `f` (... `f` (xn `f` z)...)
```

Tail recursive, second parameter accumulator:

```
foldl f z [] = z
foldl f z (x:xs) = foldl (f z x) xs

foldl f z [x1,...,xn] = (...(z `f` x1) `f` ...) `f` xn
```

Relationship between `foldr` and `foldl`:

Lemma `foldl f e = foldr f e`
if `f` is associative and ``e `f` x = x `f` e``.

Tupling of results

Tupling of results

Typical application:

Avoid multiple traversals of the same data structure

Tupling of results

Typical application:

Avoid multiple traversals of the same data structure

```
average :: [Float] -> Float
average xs = (sum xs) / (length xs)
```

Requires two traversals of the argument list.

Tupling of results

Typical application:

Avoid multiple traversals of the same data structure

```
average :: [Float] -> Float
average xs = (sum xs) / (length xs)
```

Requires two traversals of the argument list.

Avoid intermediate data structures

Avoid intermediate data structures

Typical example: `map g . map f = map (g . f)`

Avoid intermediate data structures

Typical example: `map g . map f = map (g . f)`

Another example: `sum [n..m]`

Precompute expensive computations

Precompute expensive computations

```
search :: String -> String -> Bool
```

Precompute expensive computations

```
search :: String -> String -> Bool
search text s =
  table_search (hash_table text) (hash s,s)
```

Precompute expensive computations

```
search :: String -> String -> Bool
search text s =
  table_search (hash_table text) (hash s,s)

bsearch = search bible
```

Precompute expensive computations

```
search :: String -> String -> Bool
search text s =
  table_search (hash_table text) (hash s,s)

bsearch = search bible

> map bsearch ["Moses", "Goethe"]
```

Precompute expensive computations

```
search :: String -> String -> Bool
search text s =
  table_search (hash_table text) (hash s,s)

bsearch = search bible

> map bsearch ["Moses", "Goethe"]
```

Better:

```
search text = \s -> table_search ht (hash s,s)
  where ht = hash_table text
```

Lazy evaluation

Lazy evaluation

Not everything that is good for cbv is good for lazy evaluation

Lazy evaluation

Not everything that is good for cbv is good for lazy evaluation

Example: `length2` under lazy evaluation

```
length []       =  0
length (x:xs)   =  length xs + 1

length2 []      n  =  n
length2 (x:xs)  n  =  length2 xs (n+1)
```

Compare executions:

```
length [a,b,c]
= length [b,c] + 1
= (length [c] + 1) + 1
= ((length [] + 1) + 1) + 1
= ((0 + 1) + 1) + 1
= 3

length2 [a,b,c] 0
= length2 [b,c] 1
= length2 [c]   2
= length2 []    3
```

Lazy evaluation

Not everything that is good for cbv is good for lazy evaluation

Example: `length2` under lazy evaluation

In general: tail recursion not always better under lazy evaluation

Problem: lazy evaluation may leave many expressions unevaluated until the end, which requires more space

Lazy evaluation

Not everything that is good for cbv is good for lazy evaluation

Example: `length2` under lazy evaluation

In general: tail recursion not always better under lazy evaluation

Problem: lazy evaluation may leave many expressions unevaluated until the end, which requires more space

Space is time because it requires garbage collection — not counted by number of reductions!

Lazy evaluation

Not everything that is good for cbv is good for lazy evaluation

Example: `length2` under lazy evaluation

In general: tail recursion not always better under lazy evaluation

Problem: lazy evaluation may leave many expressions unevaluated until the end, which requires more space

Space is time because it requires garbage collection — not counted by number of reductions!