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
- Most languages follow cbv
- Number of lazy evaluation steps ≤ number of cbv steps

⇒ $O$-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

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

Justification:

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

Example: \( \text{length} \ (. \ : \ xs) = \text{length} \ xs + 1 \)
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 right-hand side instead)

Justification:

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

Example: \[ \text{length} \ (\_ : \text{xs}) = \text{length} \ \text{xs} + 1 \]
Reduce \[ \text{length} \ [1,2,3] \]

\[ \text{id} \ [] = [] \]
\[ \text{id} \ (x:xs) = x : \text{id} \ \text{xs} \]
Reduce \[ \text{id} \ [e1,e2] \]

Copies list but shares elements.
$T_2(n) =$ 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 $f$
$T_f(n) = \text{number of steps required for the evaluation of } f$
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

How to calculate (not mechanically!) $T_f(n)$:
1. From the equations for $f$ derive equations for $T_f$
How to calculate (not mechanically!) $T_f(n)$:

1. From the equations for $f$ derive equations for $T_f$
2. If the equations for $T_f$ are recursive, solve them

Example

- $[] ++ ys = ys$
- $(x:xs) ++ ys = x : (xs ++ ys)$

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

\[
\begin{align*}
\text{[] ++ ys} &= \text{ys} \\
(x:x:s) ++ y &= x: (x : s ++ y)
\end{align*}
\]

\[
\begin{align*}
T_{++}(0, n) &= O(1) \\
T_{++}(m + 1, n) &= T_{++}(m, n) + O(1)
\end{align*}
\]

\[
\Rightarrow T_{++}(m, n) = O(m)
\]

Example

\[
\begin{align*}
\text{[] ++ ys} &= \text{ys} \\
(x:x:s) ++ y &= x : (x : s ++ y)
\end{align*}
\]

\[
\begin{align*}
T_{++}(0, n) &= O(1) \\
T_{++}(m + 1, n) &= T_{++}(m, n) + O(1)
\end{align*}
\]

\[
\Rightarrow T_{++}(m, n) = O(m)
\]

Note: (+++) creates copy of first argument
Example

\[
\begin{align*}
[] & \;\, \, ++ \, \;\, ys \;\, = \;\, ys \\
(x:xs) & \;\, ++ \, \;\, ys \;\, = \;\, x : (xs \;\, ++ \;\, ys)
\end{align*}
\]

\[
\begin{align*}
T_{++}(0, n) & \;\, = \;\, O(1) \\
T_{++}(m + 1, n) & \;\, = \;\, T_{++}(m, n) + O(1)
\end{align*}
\]

\[
\Rightarrow T_{++}(m, n) = O(m)
\]

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

\[
\begin{align*}
\text{reverse} \;\, [ ] & \;\, = \;\, [ ] \\
\text{reverse} \;\, (x:xs) & \;\, = \;\, \text{reverse} \, \, xs \;\, ++ \;\, [x]
\end{align*}
\]

\[
\begin{align*}
T_{\text{reverse}}(0) & \;\, = \;\, O(1) \\
T_{\text{reverse}}(n + 1) & \;\, = \;\, T_{\text{reverse}}(n)
\end{align*}
\]

Example

\[
\begin{align*}
\text{reverse} \;\, [ ] & \;\, = \;\, [ ] \\
\text{reverse} \;\, (x:xs) & \;\, = \;\, \text{reverse} \, \, xs \;\, ++ \;\, [x]
\end{align*}
\]

\[
\begin{align*}
T_{\text{reverse}}(0) & \;\, = \;\, O(1) \\
T_{\text{reverse}}(n + 1) & \;\, = \;\, T_{\text{reverse}}(n) + T_{++}(n, 1)
\end{align*}
\]
Example

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

\[
T_{\text{reverse}}(0) = O(1)
\]
\[
T_{\text{reverse}}(n+1) = T_{\text{reverse}}(n) + T_{\text{++}}(n,1)
\]

\[\Rightarrow T_{\text{reverse}}(n) = O(n^2)\]

The worst case time complexity of an expression \(e\):

Sum up all \(T_f(n_1, ..., n_k)\)

where \(f \ e_1 \ldots e_n\) is a function call in \(e\)

The worst case time complexity of an expression \(e\):

Sum up all \(T_f(n_1, ..., n_k)\)

where \(f \ e_1 \ldots e_n\) is a function call in \(e\)

and \(n_i\) is the size of \(e_i\)
The worst case time complexity of an expression $e$:

$$\sum T_f(n_1, \ldots, n_k)$$

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 $\text{min}\;xs = \text{head}(\text{sort}\;xs)$
The worst case time complexity of an expression e:

Sum up all $T_f(n_1, \ldots, n_k)$
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 $\min\ xs = \text{head}\ (\text{sort}\ xs)$

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

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

---

13.2 Optimizing functional programs
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 equivalence to correct version

No duplication

Eliminate common subexpressions with where (or let)

Example

$$f \ x = \ g \ (h \ x) \ (h \ x)$$
No duplication

Eliminate common subexpressions with \textit{where} (or \textit{let})

Example

\begin{verbatim}
\f x = g \ (h \ x) \ (h \ x)
\h f x = g \ y \ y \ \text{where} \ y = h \ x
\end{verbatim}

Tail recursion / Endrekursion

The definition of a function \( \mathbf{f} \) is \textit{tail recursive / endrekursiv}

if every recursive call is in "end position",

if every recursive call is in "end position",

\( \begin{verbatim}
\text{it is the last function call before leaving } \mathbf{f},
\text{nothing happens afterwards}
\end{verbatim} \)
**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**

$$
\begin{align*}
\text{length} [\,] &= 0 \\
\text{length} (x:xs) &= \text{length} \, xs + 1
\end{align*}
$$

**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**

$$
\begin{align*}
\text{length} [\,] &= 0 \\
\text{length} (x:xs) &= \text{length} \, xs + 1
\end{align*}
$$

$$
\begin{align*}
\text{length2} [\,] &= n \\
\text{length2} (x:xs) &= \text{length2} \, xs \, (n+1)
\end{align*}
$$
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
= ((0 + 1) + 1) + 1
= 3

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

Tail recursive definitions can be compiled into loops.
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

\[
\begin{align*}
\text{length2} \ [\ ] & \quad n = n \\
\text{length2} \ (x:xs) & \quad n = \text{length2} \ xs \ (n+1)
\end{align*}
\]
What does tail recursive mean for

\[ f \ x = \text{if } b \ \text{then } e_1 \ \text{else } e_2 \]

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

Tail recursive example:

\[ f \ x = \text{if } x > 0 \ \text{then } f(x-1) \ \text{else } f(x+1) \]
What does tail recursive mean for
\[ f \ x = \text{if } b \text{ then } e_1 \text{ else } e_2 \]
- \( f \) does not occur in \( b \)
- if \( f \) occurs in \( e_i \) then only at the outside: \( e_i = f \ldots \)

Tail recursive example:
\[ f \ x = \text{if } x > 0 \text{ then } f(x-1) \text{ else } f(x+1) \]

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

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

An accumulating parameter is a parameter where intermediate results are accumulated.
Purpose:
- tail recursion
  - replace \((++)\) by \((:)\)

Accumulating parameter: reverse

\[
\begin{align*}
\text{reverse} \; [] & = [] \\
\text{reverse} \; (x:xs) & = \text{reverse} \; xs \; ++ \; [x] \\
T_{\text{reverse}}(n) & = O(n^2) \\
\text{itrev} \; [] \; xs & = xs \\
\text{itrev} \; (x:xs) \; ys & = \text{itrev} \; xs \; (x:ys)
\end{align*}
\]

length2 \; [] \; n & = n \\
length2 \; (x:xs) \; n & = \text{length2} \; xs \; (n+1)
Accumulating parameter: reverse

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

T_{reverse}(n) = O(n^2)
```

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

Not just tail recursive also linear:

\[
T_{itrev}(0, n) = O(1),
T_{itrev}(m + 1, n) = T_{itrev}(m, n) + O(1)
\]

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

---

Accumulating parameter: tree flattening

```haskell
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)

\[
\frac{T_{\text{flat}}(1)}{T_{\text{flat}}(h+1)} = \frac{O(1)}{2 \cdot T_{\text{flat}}(h) + T_{++}}
\]
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)

\[
\begin{align*}
T_{flat}(1) &= O(1) \\
T_{flat}(h+1) &= 2 \cdot T_{flat}(h) + T_{++}(2^h, 2^h)
\end{align*}
\]

\[
\Rightarrow T_{flat}(h) = O(h \cdot 2^h)
\]
Accumulating parameter: \texttt{foldl}

\begin{align*}
\text{foldr } f \ z \ [\ ] & = z \\
\text{foldr } f \ z \ (x:xs) & = f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}

Tail recursive, second parameter accumulator:

\begin{align*}
\text{foldr } f \ z \ [\ ] & = z \\
\text{foldr } f \ z \ (x:xs) & = f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}

\begin{align*}
\text{foldr } f \ z \ [x_1, \ldots, x_n] & = x_1 \cdot f' \ (\ldots \cdot f' (x_n \cdot f' z) \ldots)
\end{align*}

Accumulating parameter: \texttt{foldl}

\begin{align*}
\text{foldr } f \ z \ [\ ] & = z \\
\text{foldr } f \ z \ (x:xs) & = f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}

Tail recursive, second parameter accumulator:

\begin{align*}
\text{foldl } f \ z \ [\ ] & = z \\
\text{foldl } f \ z \ (x:xs) & = \text{foldl } (f \ z \ x) \ xs
\end{align*}

\begin{align*}
\text{foldl } f \ z \ [x_1, \ldots, x_n] & = (\ldots (z \cdot f' x_1 \cdot f' \ldots) \cdot f' x_n
\end{align*}

Relationship between \texttt{foldr} and \texttt{foldl}:
Accumulating parameter: foldr

\[
\begin{align*}
\text{foldr } f \ z \ [] &= z \\
\text{foldr } f \ z \ (x:xs) &= f \ x \ (\text{foldr } f \ z \ xs)
\end{align*}
\]

\[
\begin{align*}
\text{foldr } f \ z \ [x_1, \ldots, x_n] &= x_1 \text{ } f \quad (\ldots \text{ } f \quad (x_n \text{ } f \quad z) \ldots)
\end{align*}
\]

Tail recursive, second parameter accumulator:

\[
\begin{align*}
\text{foldl } f \ z \ [] &= z \\
\text{foldl } f \ z \ (x:xs) &= \text{foldl } (f \ z \ x) \ xs
\end{align*}
\]

\[
\begin{align*}
\text{foldl } f \ z \ [x_1, \ldots, x_n] &= (\ldots (z \text{ } f \quad x_1 \text{ } f \quad \ldots) \text{ } f \quad x_n
\end{align*}
\]

Relationship between foldl and foldr:

**Lemma** \( \text{foldl } f \ e = \text{foldr } f \ e \)

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.

Avoid intermediate data structures

Typical example: map g . map f = map (g . f)
Avoid intermediate data structures

Typical example: \( \text{map } g \cdot \text{map } f = \text{map } (g \cdot f) \)

Another example: \( \text{sum } [n..m] \)

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

> 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

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!