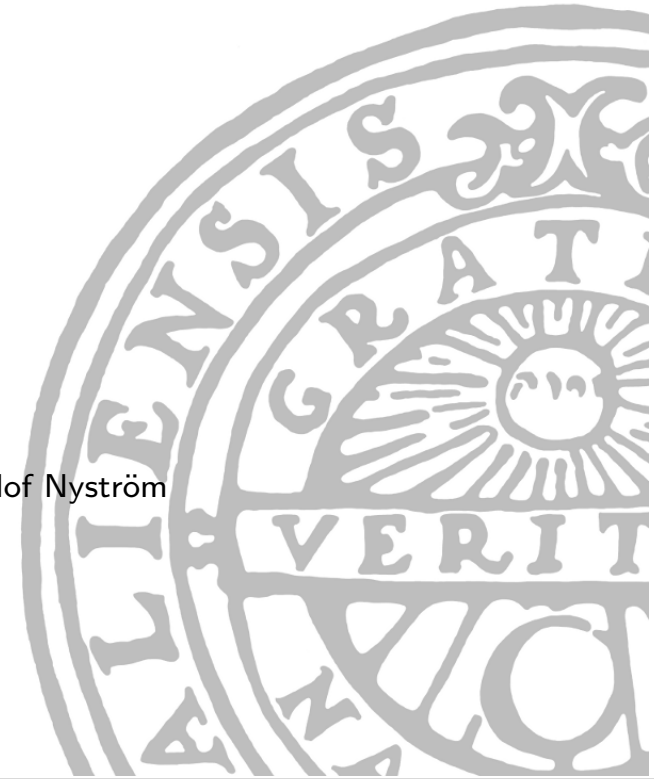
Complexity and Tail Recursion

Lars-Henrik Eriksson

Functional Programming 1

Original slides by Tjark Weber based on notes by Sven-Olof Nyström



Watch Out!

Complexity and tail recursion are two independent concepts. Please don't confuse them!

Today we will talk about complexity first, and about tail recursion during the second part of the lecture.

Complexity

Complexity

We are interested in the amount of work to evaluate a function call in relation to the size of its input(s).

Since many of you already know about algorithm analysis (and others have never heard about it), I will try to keep things simple and focus on the functional programming aspect.

Constant Time

Example:

```
fun second xs = hd (tl xs)
```

This function requires the same number of operations to compute its result (or to throw an exception) no matter what the length of the argument list xs is.

We say that a function has **constant time** complexity if the number of operations performed is bounded by a constant.

Linear Complexity

Example:

```
fun last [x] = x
  | last (_::xs) = last xs
```

Finding the last element of a list of length n requires $n - 1$ recursive calls.

We say that a function has **linear** (time) complexity if the number of operations performed is bounded by $c \cdot n$ for some constant c (where n is the size of the input).

Other examples: append, member, length

Quadratic Complexity

Example:

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]
```

If the input list has length n, reverse will call append (@) for lists of length $n-1$, $n-2$, ..., 0, and append itself has linear complexity.

When we add these terms, we find that the total number of operations required to reverse a list is approximately $\sum_{i=0}^{n-1} i = 1/2 \cdot (n^2 - n)$.

We say that a function has **quadratic** (time) complexity if the number of operations performed is bounded by $c \cdot n^2$ for some constant c (where n is the size of the input).

Polynomial Complexity

Complexity	Number of operations (at most)
constant	constant
linear	proportional to n
quadratic	proportional to n^2

Polynomial Complexity

Complexity	Number of operations (at most)
constant	constant
linear	proportional to n
quadratic	proportional to n^2
cubic	proportional to n^3
$\vdots$	$\vdots$

We say that a function has **polynomial** (time) complexity if the number of operations performed is bounded by n^k for some constant k (where n is the size of the input).

Beyond Polynomial Complexity

We say that a function has **exponential** complexity if the number of operations performed is bounded by $2^{\mathrm{poly}(n)}$.

Beyond Polynomial Complexity

We say that a function has **exponential** complexity if the number of operations performed is bounded by $2^{\mathrm{poly}(n)}$.

We say that a function has **elementary** complexity if the number of operations performed is bounded by $2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$.

Beyond Polynomial Complexity

We say that a function has **exponential** complexity if the number of operations performed is bounded by $2^{\mathrm{poly}(n)}$.

We say that a function has **elementary** complexity if the number of operations performed is bounded by $2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}$.

Ackermann function:

```
fun acker 0 m = m+1
  | acker n 0 = acker (n-1) 1
  | acker n m = acker (n-1) (acker n (m-1))
```

Non-elementary, e.g. acker 4 2 $= 2^{65536} - 3 \approx 10^{19728}$

Complexity: Some Fine Points

Time complexity does not actually measure run-time: whether you execute an algorithm on a slow or fast computer, its complexity is the same. It describes how run-time *changes* with change in input size.

We typically ignore constant factors: an algorithm that requires $10^{10} \cdot n$ operations still has linear complexity.

We typically ignore lower-order terms: an algorithm that requires $n^2 + n$ operations still has quadratic complexity.

We typically consider asymptotic behavior: any finite number of "small" inputs may be ignored.

We typically consider worst-case behavior: those execution branches that require the most operations.

Complexity of Built-in Operations

When we discuss complexity, we assume that basic operations (for instance: a function call, applying a constructor, arithmetic, pattern matching, branching) all take **constant time**. (So we don't worry about whether multiplication is more expensive than addition, for example.)

Naturally, one must be aware of built-in operations that are *not* constant time (for example, `@` and `^`).

For a more careful analysis, one would have to take into account that even the run-time of arithmetic operations depends on the size of the numbers passed as arguments.

Example: Fast reverse

Here is an efficient reverse function:

```
fun rev' acc [] = acc
  | rev' acc (x::xs) = rev' (x::acc) xs

fun rev xs = rev' [] xs
```

It is easy to see that rev will perform $n + 1$ function calls for a list with n elements. The other work is constant time. So it is easy to show that the total amount of work is linear in the size of the input.

Measurements

Here is a simple function to help measure time. The argument to time is a function which does the work we want to measure when called. The result of time is a record that gives various bits of information.

```
fun time f =
  let
    val timer = Timer.startCPUTimer ()
    val _ = f () (* do the actual work *)
  in
    Timer.checkCPUTimes timer
  end
```

Long Lists

To make it easy to create long lists, use this function:

```
fun make 0 = []
  | make n = n :: make (n-1)
```

As you can see, it builds a list of length n.

A Simple Test

Let's time reverse. (All measurements on my laptop.)

```
> val xs = make 10000;
> time (fn () => reverse xs);
val it =
    {gc = {sys = 0.007, usr = 0.046},
     nongc = {sys = 0.088, usr = 0.910}}: ...
```

The time information is divided in two parts, "gc" (garbage collection) and "nongc". Each of these is further divided into system (sys) and user (usr). We are mostly interested in the nongc/usr time. (The other times to a major extent depend on outside factors.)
In this case, it is 0.910 seconds.

A Simple Test (cont.)

Let's try a list that is twice as long:

```
> val xs = make 20000;
> time (fn () => reverse xs);
val it =
    {gc = {sys = 0.064, usr = 0.384},
     nongc = {sys = 0.418, usr = 3.667}}: ...
```

Here the time is 3.667 seconds. (Our earlier analysis suggested that doubling the length of the list would give a timing about 4 times longer. $0.910 \times 4 = 3.640$ so the analysis is quite accurate.)

A Simple Test (cont.)

What about the more efficient reverse?

```
> val xs = make 10000;
> time (fn () => rev xs);
val it = {gc = {sys = 0.000, usr = 0.000},
          nongc = {sys = 0.000, usr = 0.000}}: ...

> val xs = make 1000000;
> time (fn () => rev xs);
val it = {gc = {sys = 0.050, usr = 1.677},
          nongc = {sys = 0.020, usr = 0.014}}: ...

> val xs = make 2000000;
> time (fn () => rev xs);
val it = {gc = {sys = 0.047, usr = 0.478},
          nongc = {sys = 0.031, usr = 0.030}}: ...
```

A Simple Test, Summary

Complexity can have a real impact on performance.

We had to try large inputs (i.e., long lists) to make the slowdown of the naive implementation noticeable.

In practice, programs with bad complexity often work well for small inputs, but become so slow that they are hard to use when the input becomes large.

Fast Fibonacci

Let's look at another example.

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n-1) + fib (n-2)
```

It can be shown that the cost of computing `fib n` is exponential in n.

The issue is that for many values of k, `fib k` is computed many times. For example,

```
fib 10 = fib 9 + fib 8 = (fib 8 + fib 7) + fib 8 ...
```

One way to avoid this is to solve a **more general** problem!

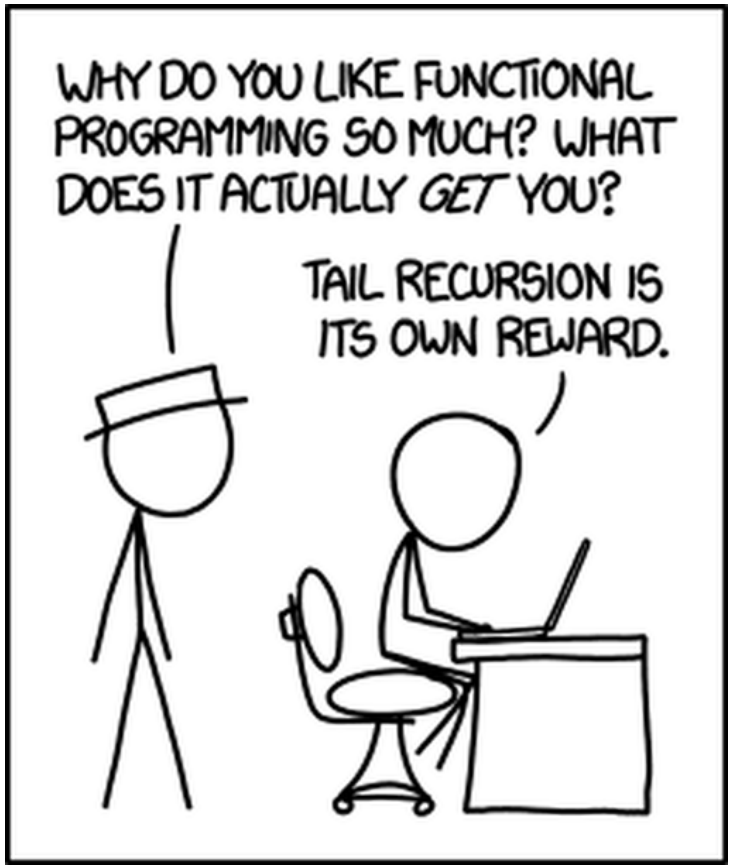
Fast Fibonacci (cont.)

This function computes fib n *and* fib (n+1):

```
(* ffib n
   TYPE: int -> int * int
   PRE: n >= 0
   POST: (fib(n), fib(n+1))
 *)
fun ffib 0 = (0, 1)
  | ffib n =
    let val (a, b) = ffib (n-1) in (b, a+b) end
```

It is easy to show that this function is linear in n, and that it computes the result indicated by the postcondition.

Tail Recursion



http://xkcd.com/1270/

Tail Recursion: Background

In most programming language implementations, function calls are implemented by pushing an **activation record** on a stack.

The activation record is used for keeping track of local variables, return values, and other things necessary to manage data local to a function call. The activation record is removed when the function call returns.

Example: length

One can get an idea of what is going on by considering a reduction.

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

length [1,2,3]
$\longrightarrow$ 1 + length [2,3]
$\longrightarrow$ 1 + (1 + length [3])
$\longrightarrow$ 1 + (1 + (1 + length []))
$\longrightarrow$ 1 + (1 + (1 + 0))

When we evaluate length [1,2,3] , the expression grows for each recursive call. For each recursive call we need to remember that we must add 1 to the result before we can return it.

Tail-Recursive length

A better implementation of length would be:

```
fun length' acc [] = acc
  | length' acc (x::xs) = length' (acc+1) xs
fun length xs = length' 0 xs
```

length [1,2,3]
$\longrightarrow$ length' 0 [1,2,3]
$\longrightarrow$ length' 1 [2,3]
$\longrightarrow$ length' 2 [3]
$\longrightarrow$ length' 3 []
$\longrightarrow$ 3

we see that the expression does *not* grow in the recursive calls.

Many implementations of functional languages take advantage of this observation, and optimize certain function calls to re-use the current activation record. This is known as **tail-call optimization**.

The first argument to length is called an **accumulator**.

Tail Position

When is a call optimized? It is not always easy to tell, but I'll give you some rules.

A call is optimized if it occurs in **tail position**. A call is in tail position if its return value, once computed, is **immediately returned** by the calling function. For instance,

- the right-hand side of a clause of a function definition,
- the then- or else- branch of an if-then-else-expression (provided the if-then-else-expression itself is in tail position),
- any branch of a case expression (provided the case expression itself is in tail position).

A call is *not* in tail position if it is an argument to another function call, arithmetic operator or datatype constructor.

Tail Position: Example

This call of f(...) is in tail position (provided the if-then-else expression itself is in tail position):

```
if ... then f(...) else ...
```

and this one is not:

```
if ... then (42, f(...)) else ...
```

Tail Position: Boolean Operators

In expressions constructed using **orelse** and **andalso**, a function call in the first argument is *not* in tail position, but a call in the second argument *is* (provided the expression itself is in tail position).

Example:

```
fun member v [] = false
  | member v (x::xs) = (v=x) orelse member v xs
```

An easy way to remember this is to view an expression a **orelse** b as a shorthand for **if** a **then** true **else** b.

Importance of Tail Recursion

How important is tail recursion in practice?

Here, length is the version which is *not* tail recursive, and length' is the one which is.

```
> val xs = make 100000000;

> time (fn () => length xs);
val it = {gc = {sys = 0.116, usr = 33.149},
          nongc = {sys = 0.000, usr = 3.524}}: ...

> time (fn () => length' xs);
val it = {gc = {sys = 0.000, usr = 0.000},
          nongc = {sys = 0.000, usr = 0.585}}: ...
```

Importance of Tail Recursion (cont.)

In this case, the difference in performance is quite striking. Note the substantial garbage collection time for the version that is not tail-recursive, which hints at its much larger (stack) memory requirements.

You are encouraged to do your own measurements! I recommend looking at slightly more complicated functions.

Besides efficiency, tail recursion gives you a way to express iteration. If you already have an iterative algorithm, you might find that the most natural way to express it in SML is as a tail-recursive function.

Tail Recursion: More Examples

Are these functions tail-recursive?

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]
```

Tail Recursion: More Examples

Are these functions tail-recursive?

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]
```

```
fun rev acc [] = acc
  | rev acc (x::xs) = rev (x::acc) xs
```

Tail Recursion: More Examples

Are these functions tail-recursive?

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]
```

```
fun rev acc [] = acc
  | rev acc (x::xs) = rev (x::acc) xs
```

```
fun last [x] = x
  | last (x::xs) = last xs
```

Tail Recursion: More Examples

Are these functions tail-recursive?

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]
```

```
fun rev acc [] = acc
  | rev acc (x::xs) = rev (x::acc) xs
```

```
fun last [x] = x
  | last (x::xs) = last xs
```

```
fun ffib 0 = (0,1)
  | ffib n =
    let val (a, b) = ffib (n-1) in (b, a+b) end
```

Tail Recursion: More Examples

Are these functions tail-recursive?

```
fun reverse [] = []
  | reverse (x::xs) = (reverse xs) @ [x]
```

```
fun rev acc [] = acc
  | rev acc (x::xs) = rev (x::acc) xs
```

```
fun last [x] = x
  | last (x::xs) = last xs
```

```
fun ffib 0 = (0,1)
  | ffib n =
    let val (a, b) = ffib (n-1) in (b, a+b) end
```

A Tail-Recursive Fibonacci Implementation

```
(* tfib' (m, p, q, n)
   PRE: 1 <= m <= n, fib(m-1)=p, fib(m)=q
   POST: fib(n)
*)
fun tfib' (m, p, q, n) =
    if m = n then q else
        tfib' (m+1, q, p+q, n)

fun tfib 0 = 0
  | tfib n = tfib' (1, 0, 1, n)
```

Check the pre- and postcondition of the help function.

Try translating this tail-recursive Fibonacci implementation into imperative code (say Java or C/C++).

Translating Imperative Into Functional Code

Here is an imperative implementation of the factorial function:

```
int fac(int n) {
  int r = 1;

  for (int i=1; i<=n; i++) {
    r = r * i;
  }

  // now r is equal to factorial of n
  return r;
}
```

Translating Imperative Into Functional Code (cont.)

Let's get rid of the fancy control structures:

```
int fac(int n) {
  int r = 1;

  int i = 1;
loop:
  if i>n goto end;
  r = r * i;
  i = i + 1;
  goto loop;

end:
  // now r is equal to factorial of n
  return r;
}
```

Translating Imperative Into Functional Code (cont.)

Each label in the imperative solution becomes a function in the functional solution and each variable becomes a function argument:

```
fun fac_end r i n = r

fun fac_loop r i n =
  if i>n then
    fac_end r i n
  else
    fac_loop (r*i) (i+1) n

fun fac n = fac_loop 1 1 n
```

In principle, any imperative program can be converted into a functional one. (Of course, in practice it is better to write proper functional code.)

Tail Recursion, Another Example: Tree Traversal

A type of binary trees (with labeled nodes):

```
datatype 'a bTree =
    Leaf
  | Node of 'a bTree * 'a * 'a bTree;
```

Example:

```
Node (Node (Leaf, 5, Leaf),
      8,
      Node (Node (Leaf, 2, Leaf),
            4,
            Leaf));
```

Tail Recursion, Another Example: Tree Traversal (cont.)

Given a binary tree, we want to compute the list of its labels (in-order).
Simple solution:

```
fun inorder Leaf =
      []
  | inorder (Node (l, x, r)) =
      (inorder l) @ (x :: inorder r)
```

This solution uses append. What is the (worst-case) complexity?

Another Example: Tree Traversal (cont.)

A better solution, using an accumulator:

```
fun inorder' acc Leaf =
      acc
  | inorder' acc (Node (l, x, r)) =
      inorder' (x :: inorder' acc r) l

fun inorder t = inorder' [] t
```

Linear complexity (why?). Is it tail-recursive?

Traversals like this one are quite common in functional programming.

Exercise: A Simple Matrix

Can you find a linear tail-recursive function, a linear function that is *not* tail-recursive, a quadratic but tail-recursive function, and a quadratic function that is not tail-recursive?

	Tail-recursive	Not tail-recursive
Linear	?	?
Quadratic	?	?