Core Design Concepts Discussed:

Recursive Functions and their Evaluation

Harley Eades III

Welcome back! In this lecture we will be learning how to define recursive functions in OCaml, but then we will turn to how modern day compilers evaluate recursive functions, and from there, we will learn how compiler implementations optimize recursive functions so that they can be just as performant as using loops.

Core Design Concepts:

Recursive Functions in OCaml

Syntax

```
# let rec f x = e;;
```

In OCaml, a recursive function is defined using the let-rect syntax. Here the name of the function is...

Core Design Concepts:

Syntax

Recursive Functions in OCaml

function name

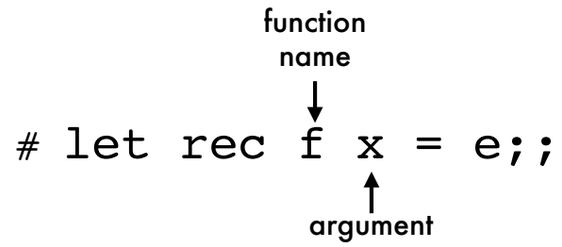
```
# let rec f x = e;;
```

f, and the function argument is..

Core Design Concepts:

Syntax

Recursive Functions in OCaml

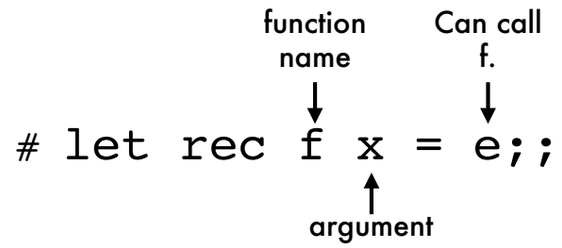


x while just as before we call e...

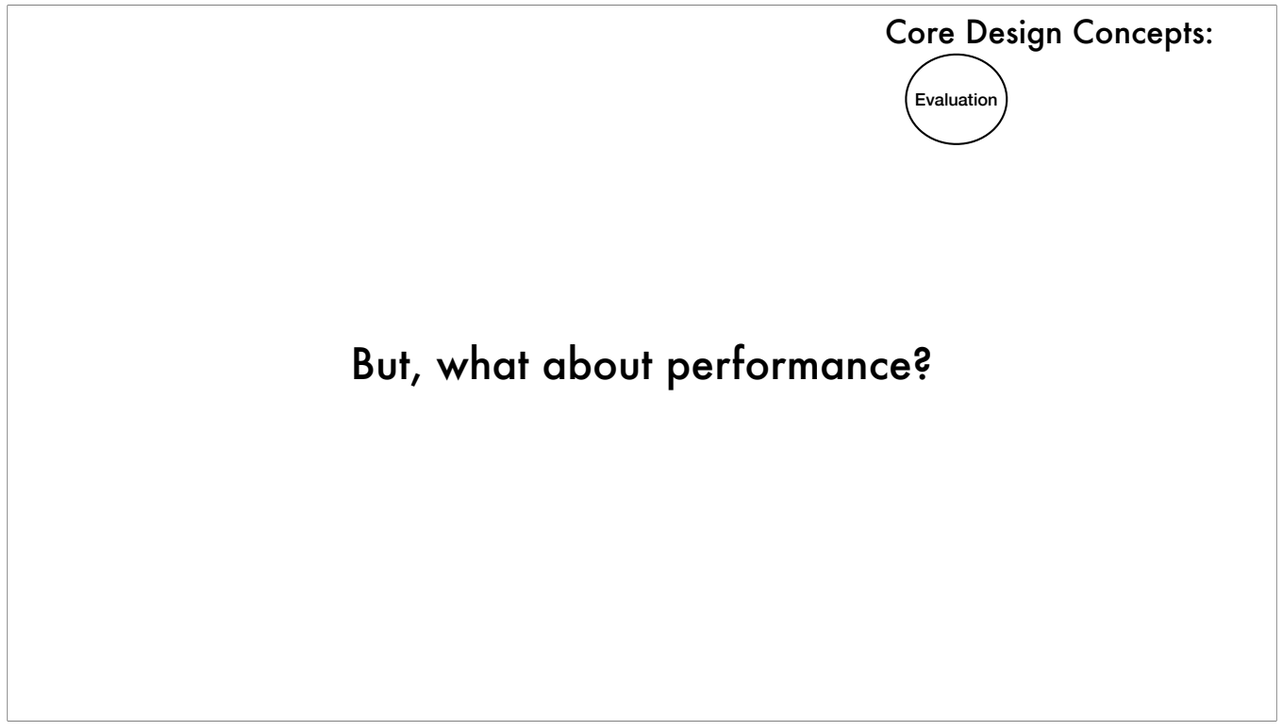
Core Design Concepts:

Syntax

Recursive Functions in OCaml



the body, but now e is allowed to call f. We will do several examples during lecture using recursion, but for now, let's turn to the most popular question people have when they learn that functional programming uses recursion over loops which is...



- Talk about OOP and the difference with FP.

In some languages like C and Java, it is often said that you shouldn't use recursion, because you take a hit to performance, but in functional programming languages we use recursion almost always, so what are the differences, and what are the reasons behind this performance hit, and what can we do about it?

In the rest of this lecture we are going to learn how modern day compilers implement recursion, and why we get hit by a performance penalty during evaluation of recursive functions. Then in the next lecture we will learn about certain optimizations the compiler can use to remedy this performance problem.

Core Design Concepts:

Evaluation

The Function Call Stack

Activation Record:

The location in memory where an executing function stores its bindings.

Activation records are sometimes referred to as frames.

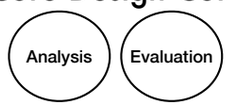
Consider evaluating the following function:

```
1: let cube n =
2:   let c = n*n*n in
3:       c;;
4: let main =
5:   let n = 5 in
6:   let ans = cube n in
7:       ans;;
8: main;;
```

During evaluation of a program the compiler keeps track of every function call in a stack called the Call Stack. The elements of this stack are often called "stack frames", but their actual name is activation records.

Let's first consider a simple non-recursive function called cube which is used to implement main. Upon initialization of the program an initial activation record is pushed onto the stack.

Core Design Concepts:



The Function Call Stack

Activation Record:

The location in memory where an executing function stores its binding.

Activation records are sometimes referred to as frames.

Activation Record: program initialization

Frame	Symbol	Value
init line: 8	`cube` `main`	`<fun>` `<fun>`

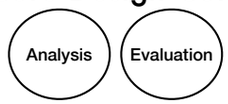
Consider evaluating the following function:

```
1: let cube n =
2:   let c = n*n*n in
3:       c;;
4: let main =
5:   let n = 5 in
6:   let ans = cube n in
7:       ans;;
8: main;;
```

It contains the line of the first function call to be evaluated, a list of functions, and a list of their values in memory. In this case, the first function call is to evaluate main on line 8. To evaluate main, we must first evaluate "cube n" on line 6. Thus, a new activation record is pushed onto the call stack...

The Function Call Stack

Core Design Concepts:



Activation Record:

The location in memory where an executing function stores its binding.

Activation records are sometimes referred to as frames.

Consider evaluating the following function:

```
1: let cube n =
2:   let c = n*n*n in
3:       c;;
4: let main =
5:   let n = 5 in
6:   let ans = cube n in
7:       ans;;
8: main;;
```

Activation Record: program initialization

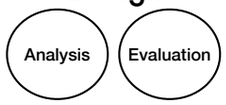
Frame	Symbol	Value
init line: 8	cube main	<fun> <fun>

Activation Record: after calling `main`

Frame	Symbol	Value
init line: 8	cube main	<fun> <fun>
main line: 6	n ans	5 ?

Now we are evaluating line 6, and to do so, we need to evaluate "cube n" to get a value of "ans", and we currently know that n is 5. And to evaluate "cube n" we must evaluate line 2 of cube...

Core Design Concepts:



The Function Call Stack

Activation Record:

The location in memory where an executing function stores its binding.

Activation records are sometimes referred to as frames.

Consider evaluating the following function:

```
1: let cube n =
2:   let c = n*n*n in
3:       c;;
4: let main =
5:   let n = 5 in
6:   let ans = cube n in
7:       ans;;
8: main;;
```

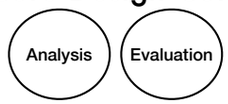
Activation Record: after calling `main`

Frame	Symbol	Value
init line: 8	`cube` `main`	`<fun>` `<fun>`
`main` line: 6	`n` `ans`	`5` `?`
`cube` line: 2	`n` `c`	`5` `125`

this adds yet another activation record for evaluating "cube n" on line 2 which equals 125. Now this value of 125 must move up in the stack as the value of ans...

The Function Call Stack

Core Design Concepts:



Activation Record:

The location in memory where an executing function stores its binding.

Activation records are sometimes referred to as frames.

Consider evaluating the following function:

```
1: let cube n =
2:   let c = n*n*n in
3:       c;;
4: let main =
5:   let n = 5 in
6:   let ans = cube n in
7:       ans;;
8: main;;
```

Activation Record: after calling `main`

Frame	Symbol	Value
init line: 8	`cube` `main`	`<fun>` `<fun>`
`main` line: 6	`n` `ans`	`5` `125`
`cube` line: 2	`n` `c`	`5` `125`

And at this point we have successfully evaluated all functions and can return the value of "ans" on line 7 which is 125...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Now let's evaluate using activation records a larger example that uses recursion. The example we will use is the ackermann function which is often used in bench marking compilers, because it's output grows exponentially very quickly, and it's fairly simple to implement. Upon initialization we are to evaluate line 17...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
init line: 17	`ackermann` `main`	`<fun>` `<fun>`

To evaluate main is to evaluate "ans1 + ans2" on line 15, but this requires us to first evaluate two function calls to "ackermann" on lines 13 and 14 respectively to get the values of ans1 and ans2. So the first step, is to add a new activation record for line 14 to the call stack...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
init line: 17	ackermann main	<fun> <fun>
main line: 14	m n ans1 ans2	1 0 ? ?

as we can see we must evaluate ans1 and ans2 which are both calls to ackermann. Thus, since m is 1 and n is 0 we will next have to evaluate ack on line 5 in order to return a value for line 13. This adds a new activation record for a call to ackermann...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
init line: 17	ackermann main	<fun> <fun>
main line: 14	m n ans1 ans2	1 0 ? ?
ackermann line: 5	m n ack	1 0 ?

Now to return a value for ack we must recursively evaluate ackermann on line 5 which adds a new activation record to the call stack for line 3, because currently m is 1 and n is 0, but we pass in 0 and 1 to ackermann thus hitting the base case...

Evaluating Recursive Functions

Core Design Concepts:

Evaluation

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
init line: 17	ackermann main	<fun> <fun>
main line: 13	m n ans1 ans2	1 0 ? ?
ackermann line: 5	m n ack	1 0 ?
ackermann line: 3	m n ret	0 1 2

We can easily calculate the value of ret given that n is 1 which is 2. This now moves up the stack to ack...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
init line: 17	ackermann main	<fun> <fun>
main line: 13	m n ans1 ans2	1 0 ? ?
ackermann line: 5	m n ack	1 0 2
ackermann line: 3	m n ret	0 1 2

and then up again to ans1...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
init line: 17	ackermann main	<fun> <fun>
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 5	m n ack	1 0 2
ackermann line: 3	m n ret	0 1 2

The next evaluation point is to get a value for ans2, but this requires us to evaluate line 14 which is another call to ackermann...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 7	m n ack1 ack2	1 1 ? ?

so we add a new activation record for line 7, because we must evaluate ackermann m m from line 14 where m is 1 which results in needing to evaluate the step case on line 7 to get a value for ack2, but and this is really important, in order to evaluate the call to ackermann on line 7 we must first evaluate the call to ackermann on line 6, because ack1 is the second argument to the call to ackermann on line 7. Thus, there is a recursive call as an argument to a recursive call. This nested recursive call is an important point when we think of performance, and we will discuss it further later. Now we add a new activation record to the stack for line 5...

- The most important detail with regards to performance
    - Having to calculate ack1 as an argument to ack2

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 7	m n ack1 ack2	1 1 ? ?
ackermann line: 5	m n ack	1 0 ?

because on line 6 we reduce m by 1 resulting in passing in 1 and 0 to ackermann. Then on line 5, we must add another activation record for line 3 because we are calling ackermann again, but now with 0 1....

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 7	m n ack1 ack2	1 1 ? ?
ackermann line: 5	m n ack	1 0 ?
ackermann line: 3	m n ret	0 1 2

Thus, on line 3 we hit a base case, and we can return 2, because we simply add one to n which is 1. Thus, this then moves up the stack to ack...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 7	m n ack1 ack2	1 1 ? ?
ackermann line: 5	m n ack	1 0 2
ackermann line: 3	m n ret	0 1 2

and up again to ack1...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 7	m n ack1 ack2	1 1 2 ?
ackermann line: 5	m n ack	1 0 2
ackermann line: 3	m n ret	0 1 2

Now we must evaluate ack2 on line 7 which is a recursive call to ackermann, but now we have the value for ack1 and so it becomes a simple non-recursive function call...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 7	m n ack1 ack2	1 1 2 ?
ackermann line: 5	m n ack	1 0 2
ackermann line: 3	m n ret	0 1 2
ackermann line: 3	m n ret	0 2 3

So we add a new activation record for the base case again, but with arguments of 0 and 2, which results in a value for ret of 3. This then gets passed up the stack...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 ?
ackermann line: 7	m n ack1 ack2	1 1 2 3
ackermann line: 5	m n ack	1 0 2
ackermann line: 3	m n ret	0 1 2
ackermann line: 3	m n ret	0 2 3

to ack2, and then up to ans1...

Evaluating Recursive Functions

Core Design Concepts:

Consider evaluating the following recursive function:

```
1. let rec ackermann m n =
2.  if m == 0
3.  then let ret = n + 1 in ret
4.  else if n == 0
5.       then let ack  = ackermann (m - 1) 1 in ack
6.       else let ack1 = ackermann m (m - 1) in
7.            let ack2 = ackermann (m - 1) ack1 in
8.              ack2
9.
10. let main =
11.   let m = 1 in
12.    let n = 0 in
13.      let ans1 = ackermann m n in
14.        let ans2 = ackermann m m in
15.          ans1 + ans2
16.
17. main;;
```

Frame	Symbol	Value
main line: 14	m n ans1 ans2	1 0 2 3
ackermann line: 7	m n ack1 ack2	1 1 2 3
ackermann line: 5	m n ack	1 0 2
ackermann line: 3	m n ret	0 1 2
ackermann line: 3	m n ret	0 2 3

at this point we can now calculate the answer for ans1 + ans2 which is 5.

So that's the basics of how we evaluate recursive functions using the call stack and activation records. In the next lecture we will look at how performance can be hit by the call stack, and how we can remedy the problem using compiler optimizations. In class, we will be doing several examples of evaluating code using the call stack. Please also read chap. 2.