CSCI 334: Principles of Programming Languages

Lecture 14: Tail Calls and Continuations

Instructor: Dan Barowy

Williams

Announcements

Please download and use *new* HW6.

The one I handed out in class had a question from HW5 on it! Oops!

Activation Records

What purpose do they serve?

They are part of a data structure ("call stack") used to evaluate a program ("stack evaluation").

The alternative form of evaluation we've discussed is λ-calculus reduction.

The two are duals:

- activation records track definitions (λ abstraction)
- activation records track function calls (application)

Tail Recursion

A function is in "tail recursive form" when the last thing a function does is either:

1. return a value
2. call itself

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

Is this function tail recursive?

No.

Tail Recursion

Let's rewrite `sum` (using a curried +) to make it obvious why + is the "last thing done."

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

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

Tail Recursion

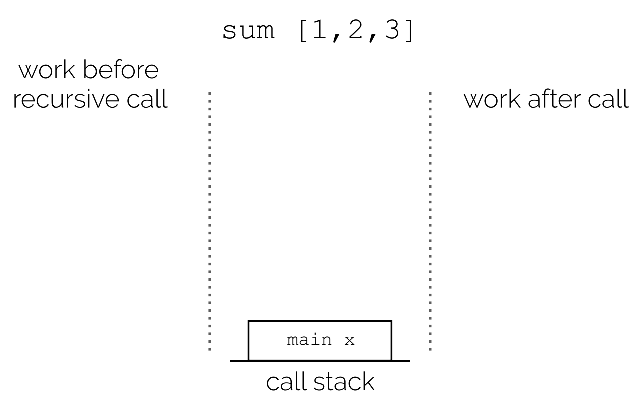
Tail recursive functions can often be automatically optimized by the language compiler; in fact, tail recursive functions aren't just *faster*; evaluation only takes *constant space*!

This form of optimization is call *tail call elimination*.

First, let's see why ordinary recursion is problematic.

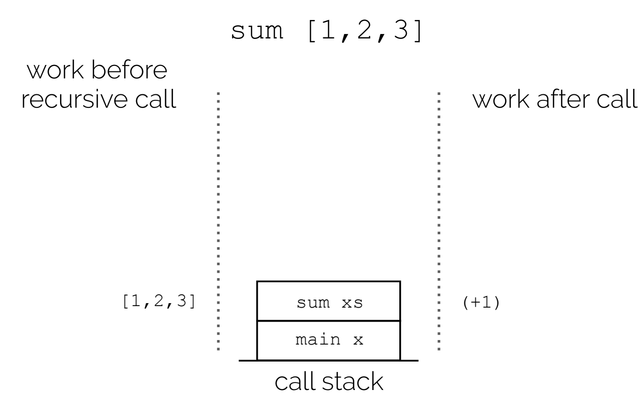
Evaluation of Ordinary Recursive Fn

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



Evaluation of Ordinary Recursive Fn

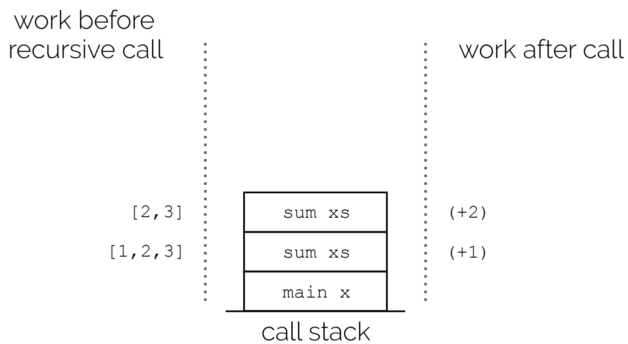
```
fun sum (x::xs) = x + sum xs
 |  sum []      = 0
```



Evaluation of Ordinary Recursive Fn

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

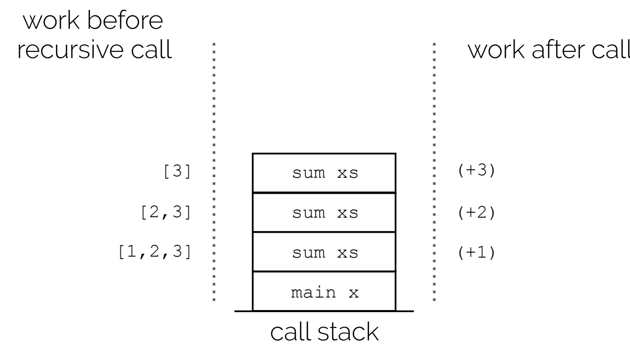
```
sum [1,2,3]
```



Evaluation of Ordinary Recursive Fn

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

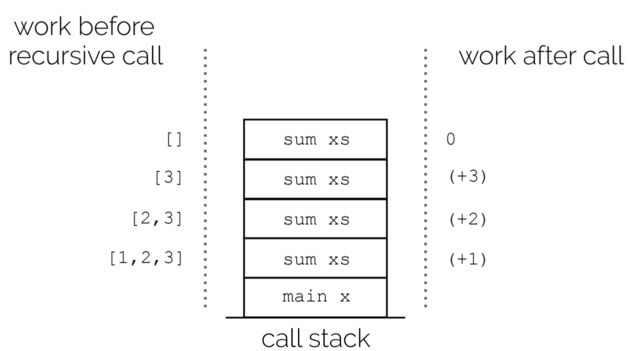
```
sum [1,2,3]
```



Evaluation of Ordinary Recursive Fn

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

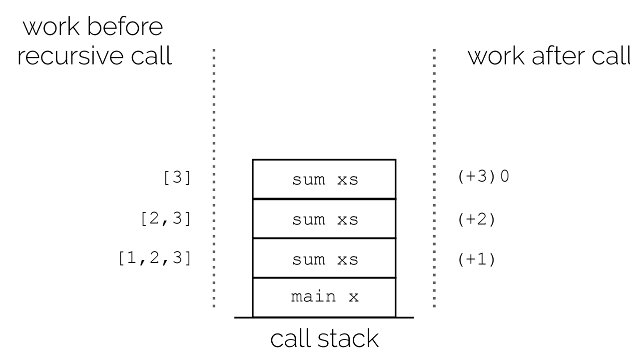
```
sum [1,2,3]
```



Evaluation of Ordinary Recursive Fn

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

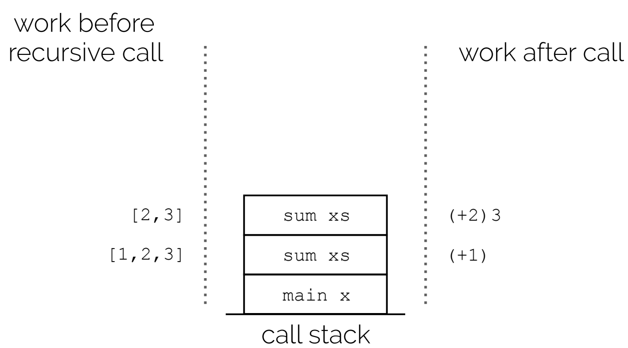
```
sum [1,2,3]
```



Evaluation of Ordinary Recursive Fn

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

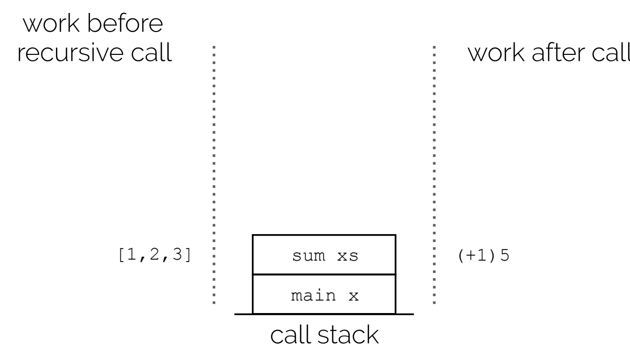
`sum [1,2,3]`



Evaluation of Ordinary Recursive Fn

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

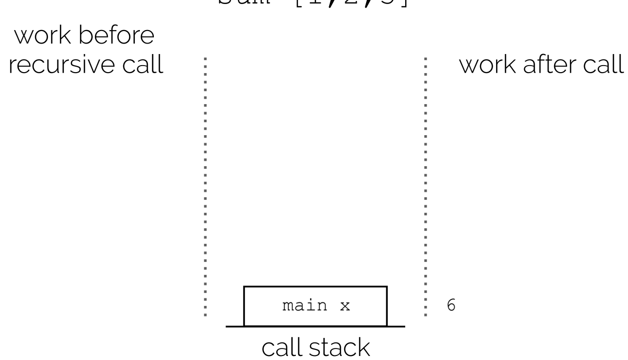
`sum [1,2,3]`



Evaluation of Ordinary Recursive Fn

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

`sum [1,2,3]`



Rewrite in tail form

Let's rewrite `sum` to make the recursive call the "last thing done."

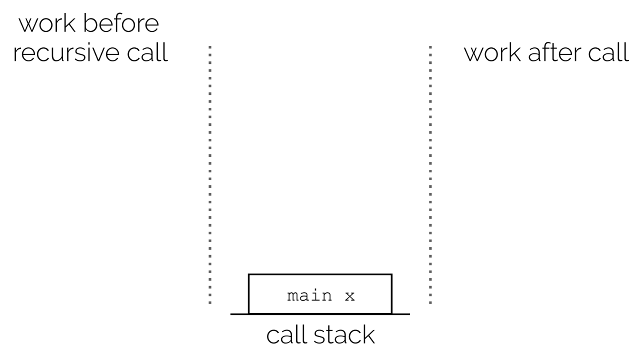
```
fun sum (x::xs) = x + sum xs
 |  sum []      = 0
```

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

Evaluation of Tail Recursive Fn

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

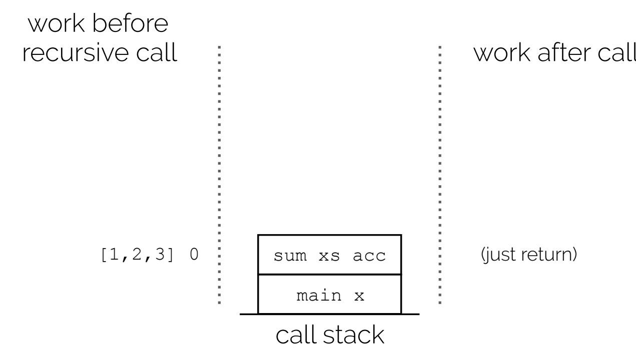
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

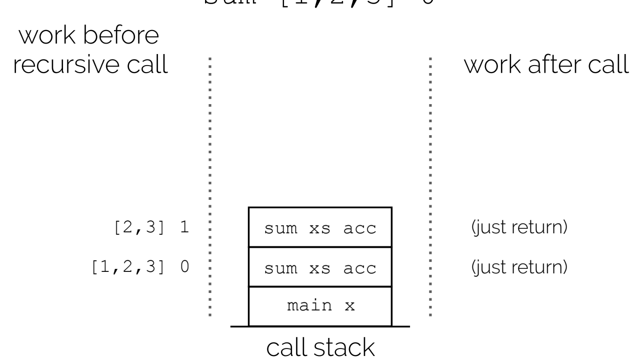
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

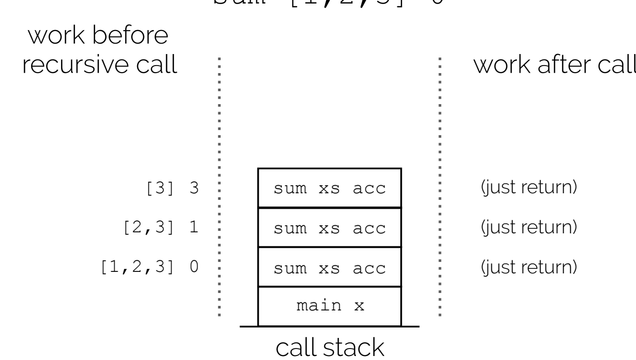
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

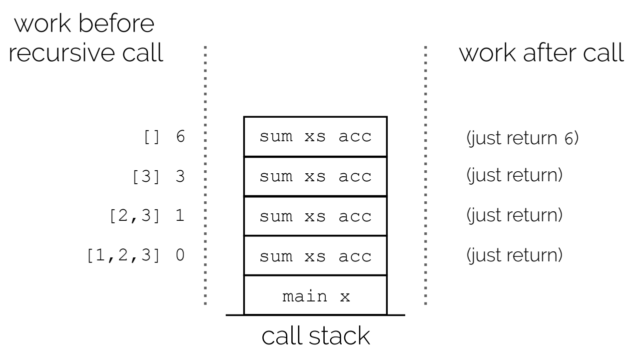
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

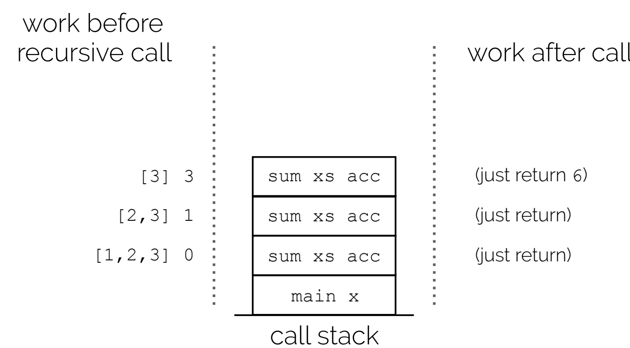
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

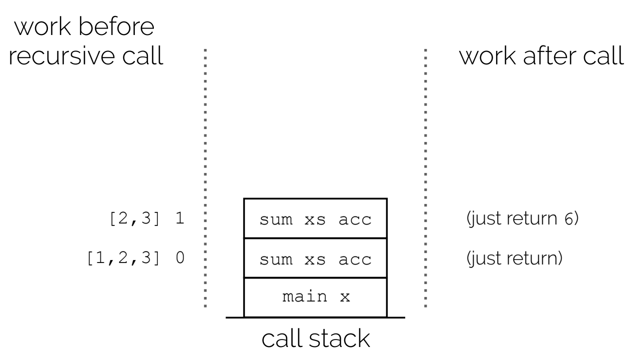
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

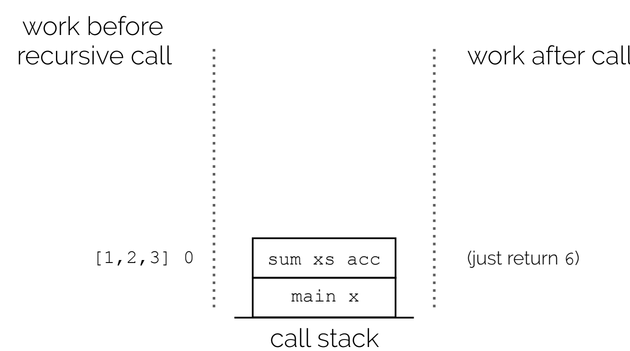
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

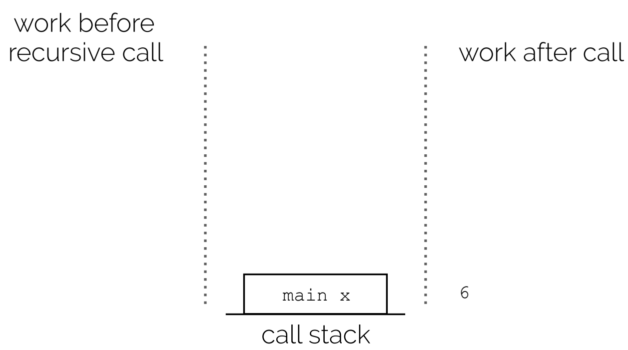
```
sum [1,2,3] 0
```



Evaluation of Tail Recursive Fn

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

```
sum [1,2,3] 0
```



Tail Call Elimination

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

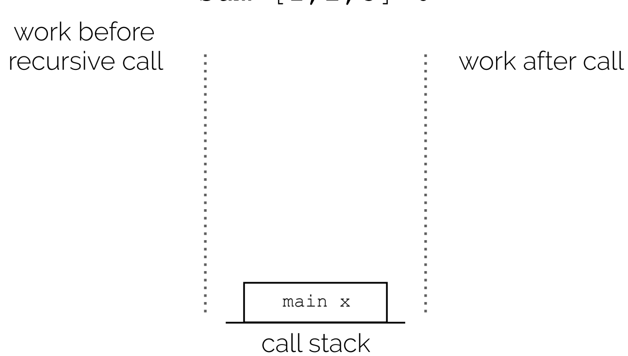
If no work is being done after a recursive call, the activation record does not need to be kept around.

In this example, we can "goto main" directly.

Optimized Evaluation of Tail Recursive Fn

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

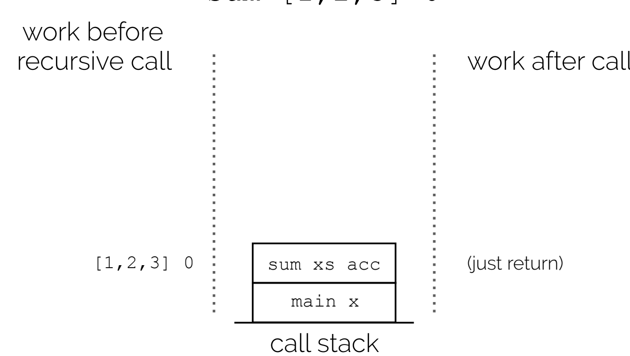
```
sum [1,2,3] 0
```



Optimized Evaluation of Tail Recursive Fn

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

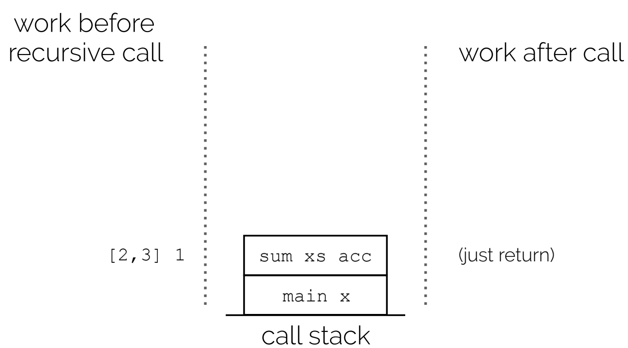
```
sum [1,2,3] 0
```



Optimized Evaluation of Tail Recursive Fn

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

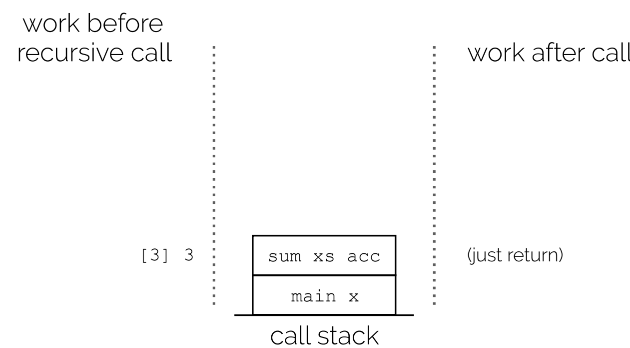
```
sum [1,2,3] 0
```



Optimized Evaluation of Tail Recursive Fn

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

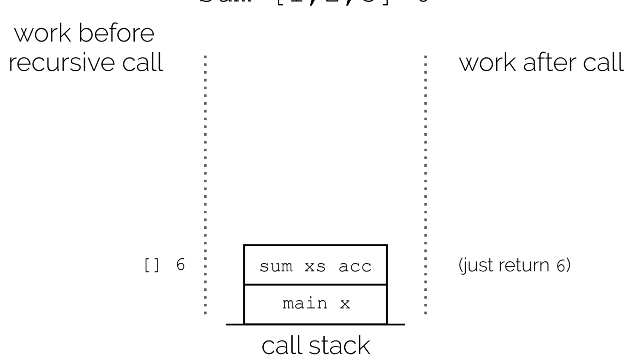
```
sum [1,2,3] 0
```



Optimized Evaluation of Tail Recursive Fn

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

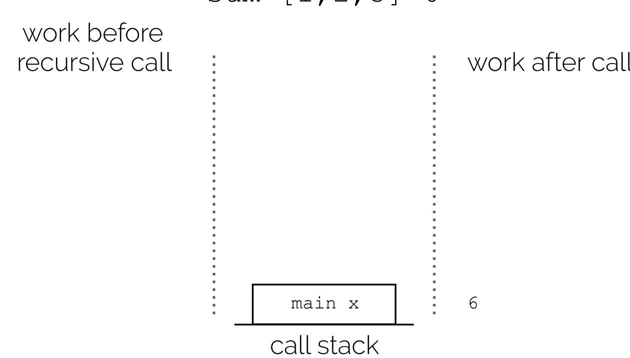
```
sum [1,2,3] 0
```



Optimized Evaluation of Tail Recursive Fn

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

```
sum [1,2,3] 0
```



By the way!

`foldl` is always tail recursive.

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

fun sum xs =
  foldl (fn (x,acc) => acc + x) xs
```

`foldr` is not! Typical implementation:

```
fun foldr f acc (x::xs) = f (x,foldr f acc xs)
 |  foldr f acc      [] = acc
```

Continuations

- Controlling evaluation order gives us more flexibility (e.g., optimization opportunities).
- One reason evaluation order is hard to think about is that much of it is implicit. What happens after a function is called depends on who calls it.
- A *continuation* makes control flow explicit.

Continuations

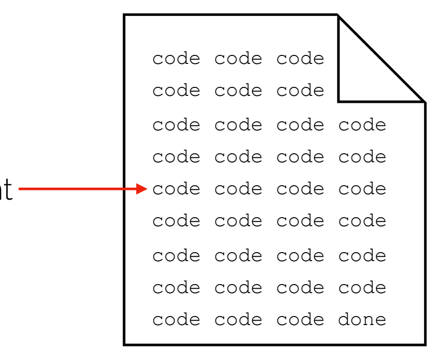
A *continuation* is a function that represents "the rest of the program" from a point in a given program.

A continuation is "what to do next."

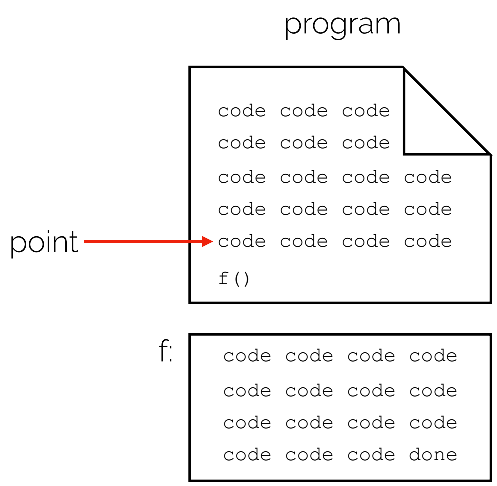
Continuations

program

point



Continuations



Example

```
fun run() =
  (print "What is your name? ";
   let val name = readline() in
     print ("Hello " ^ name)
   end)
```

Example

```
fun f() =
  let val name = readline() in
    print ("Hello " ^ name)
  end

fun run() =
  (print "What is your name? "; f())
```

Note: `run` "just returns" when `f` returns.

I.e., `run` is in "tail form."

Rewrite in continuation-passing style

Let's rewrite `sum` to use a continuation.

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

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  | sum      [] k = k 0
```

Note that `sum` is in tail form.

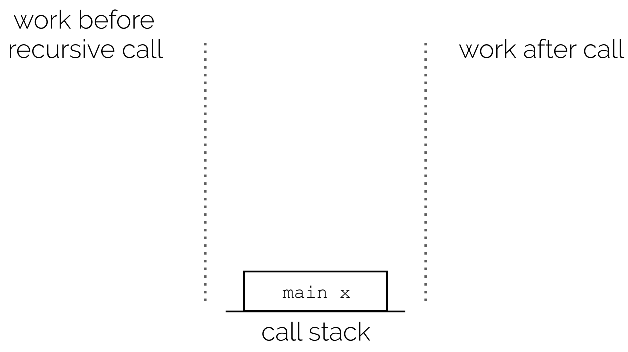
We either call `sum` or `k` as the last thing we do.

`sum` "just returns" after calls to `sum` and `k`.

Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |   sum       [] k = k 0
```

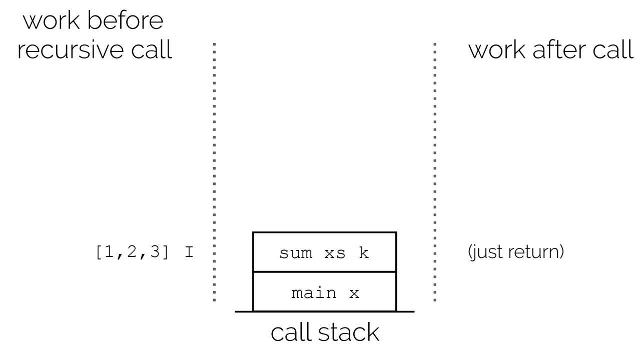
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |   sum       [] k = k 0
```

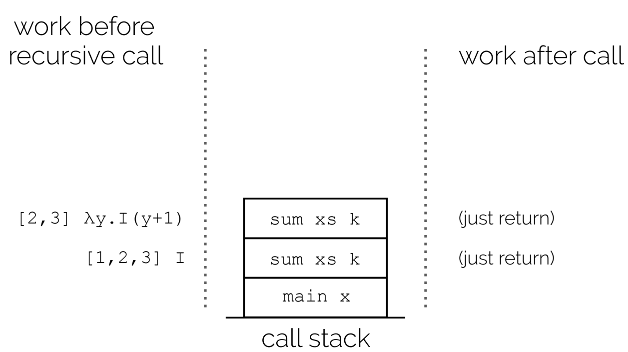
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |   sum       [] k = k 0
```

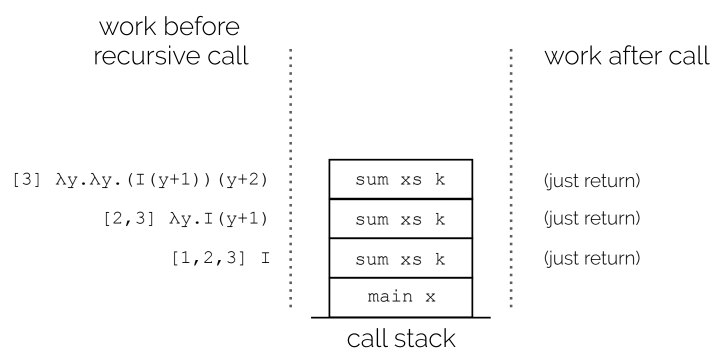
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |   sum       [] k = k 0
```

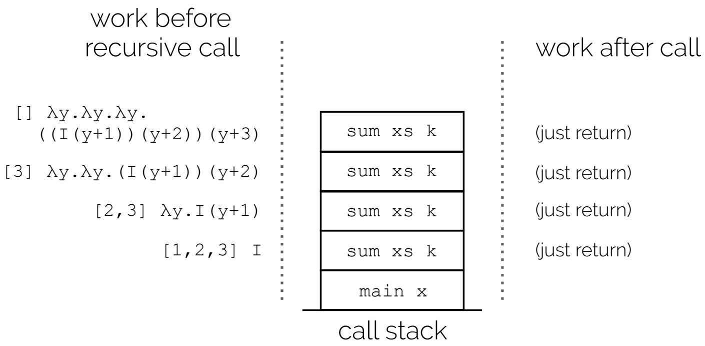
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |  sum      [] k = k 0
```

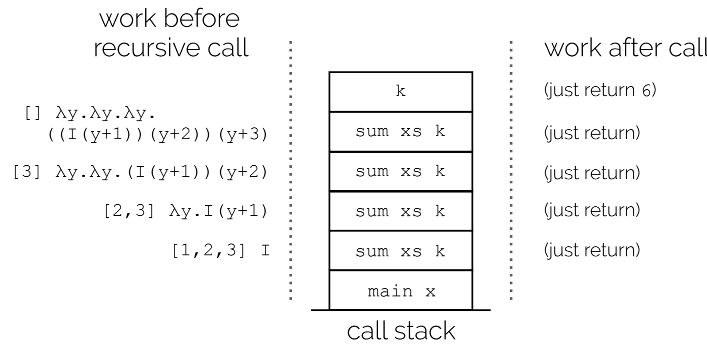
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |  sum      [] k = k 0
```

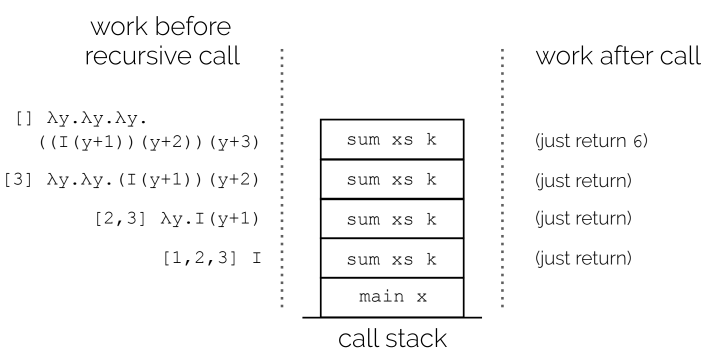
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |  sum      [] k = k 0
```

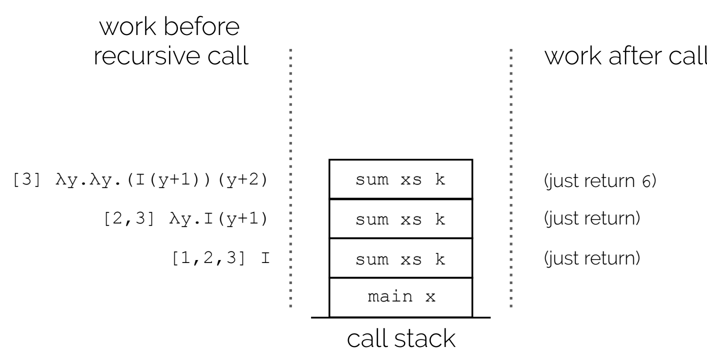
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |  sum      [] k = k 0
```

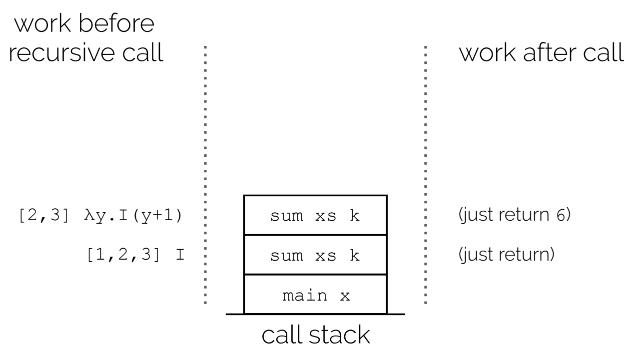
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  | sum       [] k = k 0
```

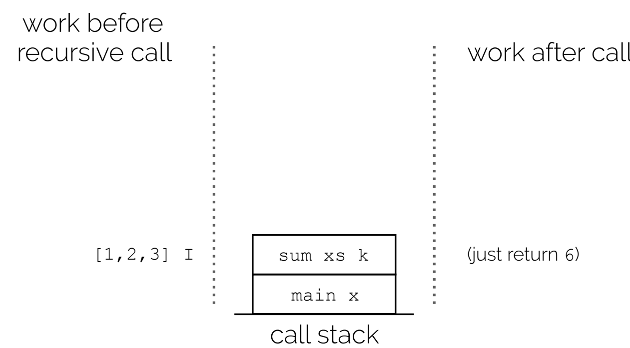
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  | sum       [] k = k 0
```

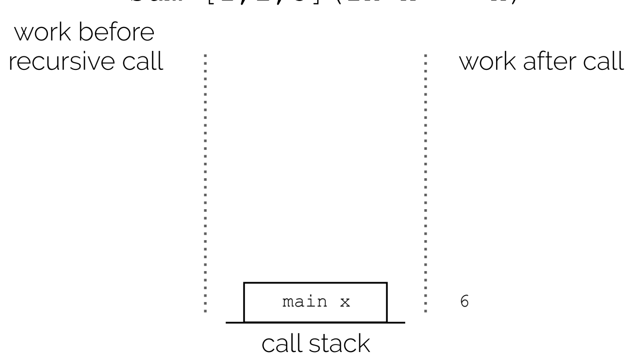
```
sum [1,2,3](fn x => x)
```



Evaluation of Function Using Continuation

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  | sum       [] k = k 0
```

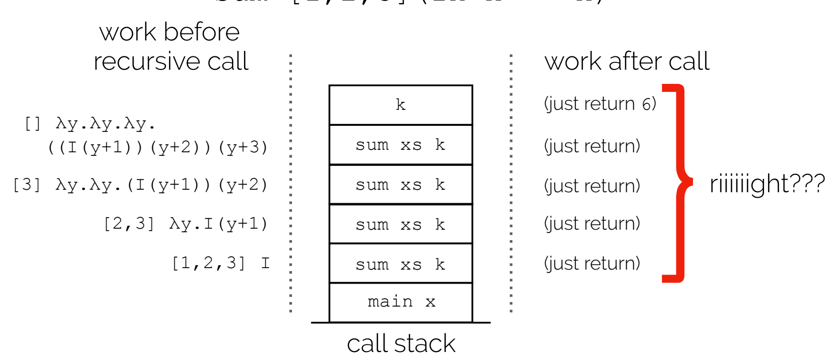
```
sum [1,2,3](fn x => x)
```



Awesome… Tail Call Elimination Time!

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  | sum       [] k = k 0
```

```
sum [1,2,3](fn x => x)
```



Awesome… Tail Call Elimination Time!

Sadly… no.

Not yet, anyway.

Why?

```
fun sum (x::xs) k = sum xs (fn y => k(y + x))
  |   sum      [] k = k 0
```

When we call this continuation,

how do we know what the value of `x` is?

Lexically scope: follow the access link for the function.

This means: we cannot eliminate activation records.

Awesome… `callcc/throw` time!

Which was why `callcc` and `throw` were invented.

`callcc`: "call with current continuation."

`throw`: used to call the continuation itself.

A function written with `callcc` and `throw` is

guaranteed to be tail-call optimizable.

Rewrite using `callcc/throw`

Let's rewrite `sum` to use continuation operators.

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

fun sum (x::xs) k = x + callcc(sum xs)
  |   sum []      k = throw k 0

        callcc (sum [1,2,3])
```

Note that `sum` doesn't look like a function in tail form.

We don't even need to build continuations ourselves.

Continuations Are Powerful

- Continuations are often referred to as a "functional goto".
- All forms of control flow can be emulated using continuations.
- They are not necessarily convenient or readable.
- Super interesting, useful applications:
  - Saving and restoring the call stack (more capable than `setjmp/longjmp`); i.e., "suspend and resume".
  - Very efficient backtracking search (AI algorithms).
  - Compile-time code transformation (used widely!).