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 $\lambda$-calculus reduction.

The two are duals:

• activation records track definitions ($\lambda$ 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

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

Is this function tail recursive?

No.
Tail Recursion

Let’s rewrite \( \text{sum} \) (using a curried +) to make it obvious why + is the “last thing done.”

\[
\begin{align*}
\text{fun} \quad \text{sum} \ (x::xs) &= x + \text{sum} \ xs \\
&| \quad \text{sum} \ [] &= 0
\end{align*}
\]

\[
\begin{align*}
\text{fun} \quad \text{sum} \ (x::xs) &= (+x)(\text{sum} \ xs) \\
&| \quad \text{sum} \ [] &= 0
\end{align*}
\]

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 called \textit{tail call elimination}.

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

Evaluation of Ordinary Recursive Fn

\[
\begin{align*}
\text{fun} \quad \text{sum} \ (x::xs) &= x + \text{sum} \ xs \\
&| \quad \text{sum} \ [] &= 0
\end{align*}
\]

\[
\begin{align*}
\text{sum} \ [1,2,3] \\
\text{work before recursive call} &| \quad \text{work after call}
\end{align*}
\]

Evaluation of Ordinary Recursive Fn

\[
\begin{align*}
\text{fun} \quad \text{sum} \ (x::xs) &= x + \text{sum} \ xs \\
&| \quad \text{sum} \ [] &= 0
\end{align*}
\]

\[
\begin{align*}
\text{sum} \ [1,2,3] \\
\text{work before recursive call} &| \quad \text{work after call}
\end{align*}
\]

\[
\begin{align*}
[1,2,3] &| \quad \text{sum} \ xs \\
\text{main} \ x &| \quad (+1)
\end{align*}
\]

call stack

\[
\begin{align*}
\text{main} \ x \\
call stack
\end{align*}
\]

\[
\begin{align*}
\text{main} \ x \\
call stack
\end{align*}
\]
Evaluation of Ordinary Recursive Fn

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

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

**Work before recursive call**

- [2,3]
- [1,2,3]

**Work after call**

- [2,3]
- [1,2,3]
- main x

```
call stack
```

Evaluation of Ordinary Recursive Fn

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

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

**Work before recursive call**

- [3]
- [2,3]
- [1,2,3]

**Work after call**

- [3]
- [2,3]
- [1,2,3]
- main x

```
call stack
```

Evaluation of Ordinary Recursive Fn

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

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

**Work before recursive call**

- []
- [3]
- [2,3]
- [1,2,3]

**Work after call**

- []
- [3]
- [2,3]
- [1,2,3]
- main x

```
call stack
```
Evaluation of Ordinary Recursive Fn

\[
\text{fun sum } (x::xs) = x + \text{sum } xs \\
| \text{sum } [] = 0
\]

<table>
<thead>
<tr>
<th>sum [1,2,3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>work before recursive call</td>
</tr>
<tr>
<td>[2,3]</td>
</tr>
<tr>
<td>sum xs</td>
</tr>
<tr>
<td>[1,2,3]</td>
</tr>
<tr>
<td>sum xs</td>
</tr>
<tr>
<td>main x</td>
</tr>
<tr>
<td>call stack</td>
</tr>
</tbody>
</table>

| work after call |
| (+2)3 |
| (+1) |

Evaluation of Ordinary Recursive Fn

\[
\text{fun sum } (x::xs) = x + \text{sum } xs \\
| \text{sum } [] = 0
\]

<table>
<thead>
<tr>
<th>sum [1,2,3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>work before recursive call</td>
</tr>
<tr>
<td>[1,2,3]</td>
</tr>
<tr>
<td>sum xs</td>
</tr>
<tr>
<td>main x</td>
</tr>
<tr>
<td>call stack</td>
</tr>
</tbody>
</table>

| work after call |
| (+1)5 |

Evaluation of Ordinary Recursive Fn

\[
\text{fun sum } (x::xs) = x + \text{sum } xs \\
| \text{sum } [] = 0
\]

<table>
<thead>
<tr>
<th>sum [1,2,3]</th>
</tr>
</thead>
<tbody>
<tr>
<td>work before recursive call</td>
</tr>
<tr>
<td>main x</td>
</tr>
<tr>
<td>call stack</td>
</tr>
</tbody>
</table>

| work after call |
| (+1)5 |

Rewrite in tail form

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

\[
\text{fun sum } (x::xs) = x + \text{sum } xs \\
| \text{sum } [] = 0
\]

\[
\text{fun sum } (x::xs) = \text{sum } xs (\text{acc } + x) \\
| \text{sum } [] \text{ acc } = \text{acc}
\]
Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

main x

call stack

Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

[1,2,3] 0

sum xs acc

call stack

(just return)

Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

[1,2,3] 0

sum xs acc

call stack

(just return)

Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

[3] 3

sum xs acc

(just return)

[2,3] 1

sum xs acc

(just return)

[1,2,3] 0

sum xs acc

main x

call stack

(just return)

(just return)
Evaluation of Tail Recursive Fn

\[
\text{fun sum (x::xs) acc = sum xs (acc + x)} \\
| \text{sum } [\ ] \text{ acc = acc}
\]

\[
\text{sum } [1,2,3] \text{ 0}
\]

work before recursive call

work after call

\[
\begin{array}{c}
[1,2,3] 0 \\
(\text{just return 6})
\end{array}
\]

\[
\begin{array}{c}
[2,3] 1 \\
(\text{just return})
\end{array}
\]

\[
\begin{array}{c}
[1,2,3] 0 \\
(\text{just return})
\end{array}
\]

call stack
Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

main x

call stack

6

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

work before recursive call

work after call

main x

call stack

6

Optimized Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

[1,2,3] 0

sum xs acc

main x

(call stack)

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

sum [1,2,3] 0

work before recursive call

work after call

[1,2,3] 1

sum xs acc

main x

call stack

(just return)

Optimized Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

[3] 3

sum xs acc

main x

call stack

(just return)

Optimized Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

[] 6

sum xs acc

main x

call stack

(just return 6)

Optimized Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

[] 6

sum xs acc

main x

call stack

(just return 6)

Optimized Evaluation of Tail Recursive Fn

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

sum [1,2,3] 0

work before recursive call

work after call

main x

6
By the way!

*foldl* is always tail recursive.

\[
\text{fun sum (x::xs) acc = sum xs (acc + x)} \\
| \text{sum [] acc = acc}
\]

\[
\text{fun sum xs = foldl (fn (x,acc) => acc + x) xs}
\]

*foldr* is not! Typical implementation:

\[
\text{fun foldr f acc (x::xs) = f (x, foldr f acc xs)} \\
| \text{foldr f acc [] = acc}
\]

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

Example

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

Rewrite in continuation-passing style

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

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

Example

```
fun f() =
  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.”
```
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)

work before recursive call

work after call

[1,2,3] I

sum xs k

(call stack)

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)

work before recursive call

work after call

[1,2,3] I

sum xs k

(call stack)
Evaluation of Function Using Continuation

\[
\text{fun sum (x::xs) } k = \text{sum xs (fn y } \Rightarrow k(y + x)) \\
\mid \text{sum } [ ] k = k \ 0
\]

\[
\text{sum } [1,2,3](\text{fn } x \Rightarrow x)
\]

- work before recursive call
- work after call

Evaluation of Function Using Continuation

\[
\text{fun sum (x::xs) } k = \text{sum xs (fn y } \Rightarrow k(y + x)) \\
\mid \text{sum } [ ] k = k \ 0
\]

\[
\text{sum } [1,2,3](\text{fn } x \Rightarrow x)
\]

- work before recursive call
- work after call

Evaluation of Function Using Continuation

\[
\text{fun sum (x::xs) } k = \text{sum xs (fn y } \Rightarrow k(y + x)) \\
\mid \text{sum } [ ] k = k \ 0
\]

\[
\text{sum } [1,2,3](\text{fn } x \Rightarrow x)
\]

- work before recursive call
- work after call

Evaluation of Function Using Continuation

\[
\text{fun sum (x::xs) } k = \text{sum xs (fn y } \Rightarrow k(y + x)) \\
\mid \text{sum } [ ] k = k \ 0
\]

\[
\text{sum } [1,2,3](\text{fn } x \Rightarrow x)
\]

- work before recursive call
- work after call
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)
```

**work before recursive call**

**work after call**

```
[2,3] λy.I(y+1)
[1,2,3] I
```

```
sum xs k
sum xs k
main x
```

**call stack**

```
[1,2,3] I
```

**work before recursive call**

**work after call**

```
[1,2,3] I
```

```
sum xs k
main x
```

**call stack**

```
riiiiiight???
```

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)
```

**work before recursive call**

**work after call**

```
[] λy.λy.λy.(I(y+1))(y+2)(y+3)
[3] λy.λy.λy.(I(y+1))(y+2)
[2,3] λy.λy.(I(y+1))
[1,2,3] I
```

```
k
sum xs k
sum xs k
sum xs k
```

**call stack**

```
k
```

**work after call**

```
(main x)
```

**call stack**

```
riiiiiight???
```
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 \( \text{sum} \) 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 \( \text{sum} \) 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!).