Tail Recursion

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Objectives

- Identify expressions that have subexpressions in tail position.
- Explain the tail call optimization.
- Convert a direct style recursive function into an equivalent tail recursive function.

Tail Calls

Tail Position A subexpression *s* of expressions *e*, if it is evaluated, will be taken as the value of *e*. Consider this code:

- `if x > 3 then x + 2 else x - 4`
- `f (x * 3)` – no (proper) tail position here

Tail Call A function call that occurs in tail position

- `if h x then h x else x + g x`

Your Turn

Find the tail calls!

Example Code

```
fact1 0 = 1
fact1 n = n * fact1 (n-1)

fact2 n = aux n 1
   where aux 0 a = a
         aux n a = aux (n-1) (a*n)

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

Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?

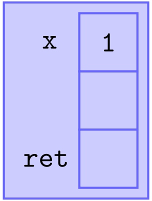
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



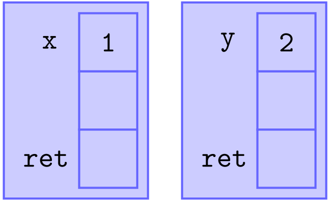
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



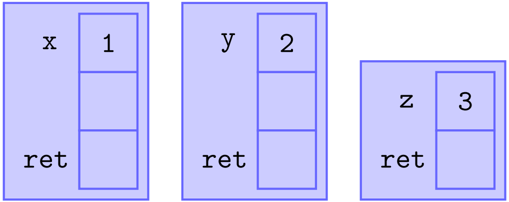
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



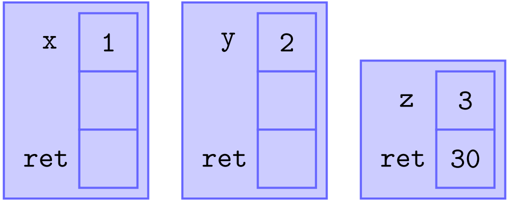
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



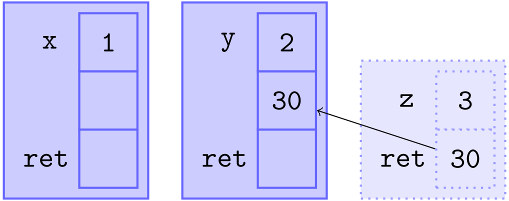
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



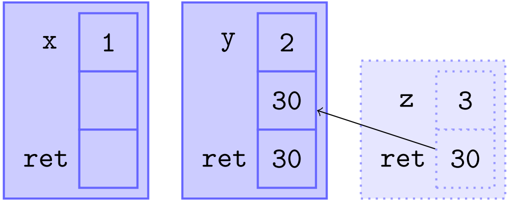
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



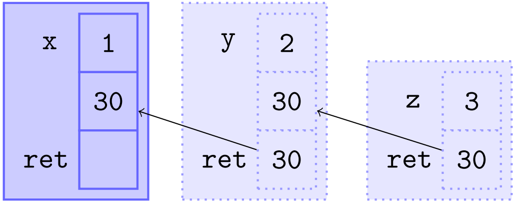
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



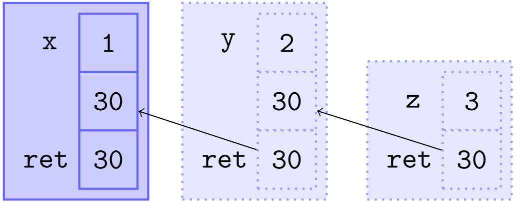
Tail Call Example

- If one function calls another in tail position, we get a special behavior.

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- What happens when we call `foo 1`?



The Tail Call Optimization

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

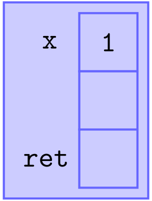
- If that's the case, we can cut out the middle man …

The Tail Call Optimization

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- If that’s the case, we can cut out the middle man …

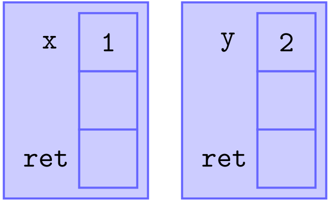


The Tail Call Optimization

Example

```haskell
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- If that's the case, we can cut out the middle man ...

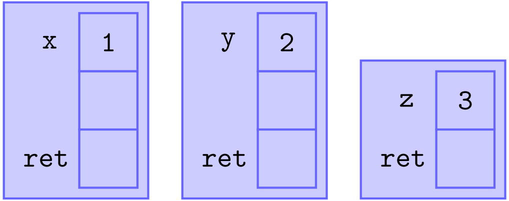


The Tail Call Optimization

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- If that's the case, we can cut out the middle man ...

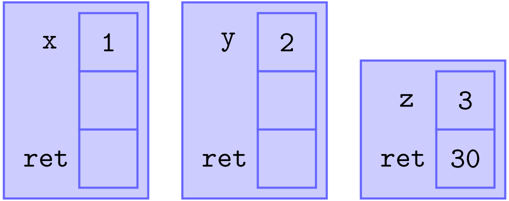


The Tail Call Optimization

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- If that's the case, we can cut out the middle man …

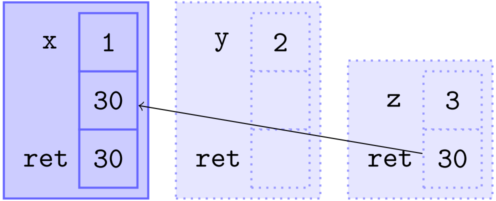


The Tail Call Optimization

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- If that's the case, we can cut out the middle man …



The Tail Call Optimization

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

- If that's the case, we can cut out the middle man ...
- Actually, we can do even better than that.

The Optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

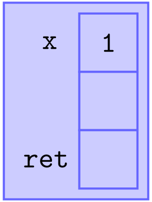
```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

The Optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

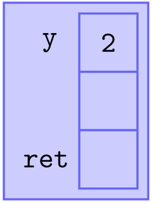


The Optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

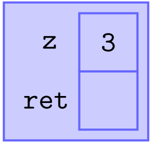


The Optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```



The Optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

z	3
ret	30

The Optimization

- When a function is in tail position, the compiler will *recycle the activation record*!

Example

```
foo x = bar (x+1)
bar y = baz (y+1)
baz z = z * 10
```

z	3
ret	30

- This allows recursive functions to be written as loops internally.

Direct-Style Recursion

- In recursion, you split the input into the "first piece" and the "rest of the input."
- In direct-style recursion: the recursive call computes the result for the rest of the input, and then the function combines the result with the first piece.
- In other words, you wait until the recursive call is done to generate your result.

Direct Style Summation

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

Accumulating Recursion

- In accumulating recursion: generate an intermediate result *now*, and give that to the recursive call.
- Usually this requires an auxiliary function.

Tail Recursive Summation

```
sum xx = aux xx 0
   where aux [] a = a
         aux (x:xs) a = aux xs (a+x)
```

Convert These Functions!

- Here are three functions. Try converting them to tail recursion.

```haskell
fun1  []  = 0
fun1 (x:xs) | even x = fun1 xs - 1
            | odd x  = fun1 xs + 1

fun2 1 = 0
fun2 n = 1 + fun2 (n `div` 2)

fun3 1 = 1
fun3 2 = 1
fun3 n = fun3 (n-1) + fun3 (n-2)
```

Solution for `fun1` and `fun2`

- Usually it's best to create a local auxiliary function.

```haskell
fun1 xx = aux xx 0
  where aux [] a = a
        aux (x:xs) | even x = aux xs (a-1)
                   | odd x = aux xs (a+1)

fun2 n = aux n 1
  where aux 1 a = a
        aux n a = aux (n `div` 2) (a+1)
```

Solution for `fun3`

- Because the recursion calls itself twice, we need *two* accumulators.

```
fun3 n = aux n 1 1
  where aux 0 f1 f2 = f1
        aux n f1 f2 = aux (n-1) f2 (f1+f2)
```

References

[DG05] Olivier Danvy and Mayer Goldberg. "There and Back Again". In: *Fundamenta Informaticae* 66.4 (Jan. 2005), pp. 397–413. ISSN: 0169-2968. URL: http://dl.acm.org/citation.cfm?id=1227189.1227194.

[Ste77] Guy Lewis Steele Jr. "Debunking the "Expensive Procedure Call" Myth or, Procedure Call Implementations Considered Harmful or, LAMBDA: The Ultimate GOTO". In: *Proceedings of the 1977 Annual Conference*. ACM '77. Seattle, Washington: ACM, 1977, pp. 153–162. URL: http://doi.acm.org/10.1145/800179.810196.