

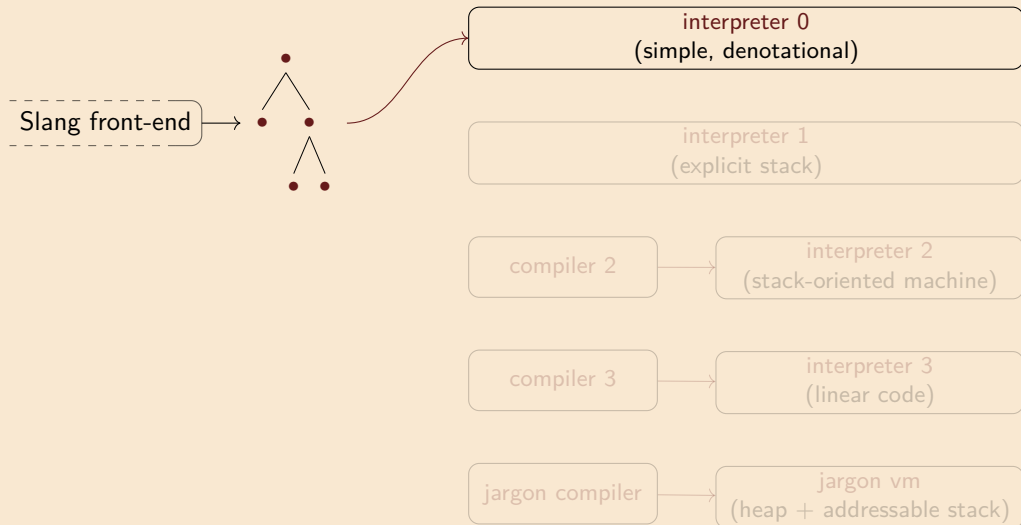
Compiler Construction

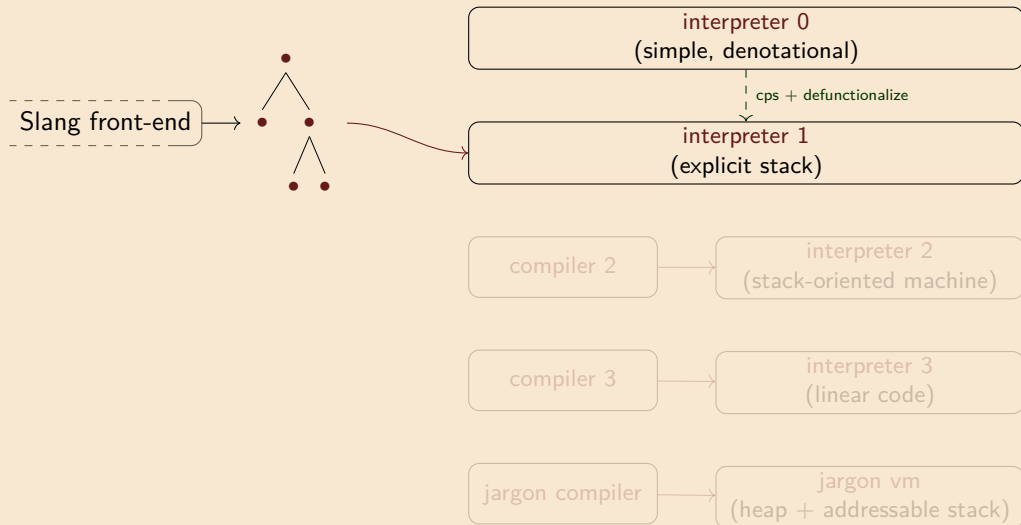
Lecture 8: CPS & defunctionalization

Jeremy Yallop

`jeremy.yallop@cl.cam.ac.uk`

Lent 2026





Source-to-source transformations

Source-to-source transformations

Source to
source



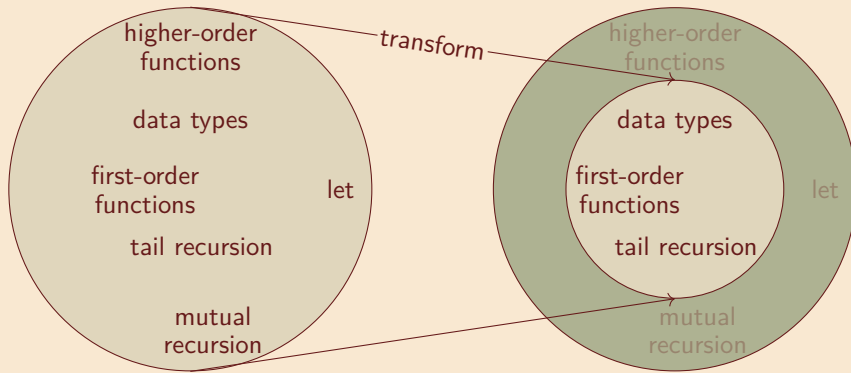
CPS

D17n

CPS +
D17n

Mutual
recursion

Source-to-source transformations map programs to a subset of the input language



Source-to-source transformations can show that some constructs are inessential

Source to
source



CPS

D17n

CPS +
D17n

Mutual
recursion

Interpreter 0 uses OCaml's stack and higher-order functions to implement Slang:

```
let rec interpret (e, env, store) =  
  match e with  
  ...  
  | Lambda(x, e) → FUN (fun (v, s) →  
                        interpret(e, update(env, (x, v)), s)), store  
  | App(e1, e2) →  
    let (v2, store1) = interpret(e2, env, store) in  
    let (v1, store2) = interpret(e1, env, store1) in  
    ...
```

Our aim: **transform the interpreter** so it doesn't use these features

Illustrating on fib

Source to
source



CPS

D17n

CPS +
D17n

Mutual
recursion

Fibonacci function

Illustrate ideas on fib function

Aim: apply ideas to interpreter 0

cps --- d17n --- indexed recursion -----> Fibonacci machine

Continuation-**P**assing **S**tyl

Continuation-passing style: motivation

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Programs in **continuation-passing style** have some useful properties:

Evaluation order is explicit

Every call is a tail call

$$f \ (g \ x) \rightsquigarrow g \ x \ (\text{fun } y \rightarrow f \ y \ k)$$

Every intermediate result is named

Every *continuation* is reified

CPS conversion of fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

let-bind function calls

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else let a = fib (m-1) in  
        let b = fib (m-2) in  
        a+b
```

CPS
convert

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

CPS conversion of fib: details

Source to
source

CPS

1. Add a continuation parameter k to each function
2. Apply k to values returned by the function
3. Replace each application let binding with a continuation argument

D17n

CPS +
D17n

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else let a = fib (m-1) in  
        let b = fib (m-2) in  
        a+b
```

CPS
convert

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

Mutual
recursion

CPS conversion of fib: details

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

1. Add a continuation parameter k to each function
2. Apply k to values returned by the function
3. Replace each application let binding with a continuation argument

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else let a = fib (m-1) in  
        let b = fib (m-2) in  
        a+b
```

CPS
convert

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

CPS conversion of fib: details

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

1. Add a continuation parameter k to each function
2. Apply k to values returned by the function
3. Replace each application let binding with a continuation argument

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else let a = fib (m-1) in  
        let b = fib (m-2) in  
        a+b
```

CPS
convert

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

CPS conversion of fib: details

Source to
source

CPS

1. Add a continuation parameter k to each function
2. Apply k to values returned by the function
3. Replace each application let binding with a continuation argument

D17n

CPS +
D17n

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else let a = fib (m-1) in  
        let b = fib (m-2) in  
        a+b
```

CPS
convert

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

Mutual
recursion

CPS conversion of fib: details

Source to
source

CPS

1. Add a continuation parameter k to each function
2. Apply k to values returned by the function
3. Replace each application let binding with a continuation argument

D17n

CPS +
D17n

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else let a = fib (m-1) in  
        let b = fib (m-2) in  
        a+b
```

CPS
convert

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

Mutual
recursion

Use the identity continuation

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

`fib_cps` has the type `int → (int → int) → int`.

To recover a function of type `int → int`, pass the identity continuation `fun x → x`:

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

```
let fib_1 x = fib_cps x (fun x → x)
```

Now `fib_1` can be used like `fib`:

```
List.map fib_1 [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10]  
↪ [1; 1; 2; 3; 5; 8; 13; 21; 34; 55; 89]
```

Correctness of CPS conversion for fib

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k \ (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k \ (\text{fib } 0)$.

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k \ (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k \ (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k \ (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k \ (\text{fib } (m+1))$.

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

$\text{fib_cps } (m+1) \ k$

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k \ (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k \ (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k \ (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k \ (\text{fib } (m+1))$.

$\text{fib_cps } (m+1) \ k$

$\equiv (\text{expand fib_cps}) \dots$

$\text{if } m+1 = 1 \text{ then } k \ 1 \text{ else fib_cps } ((m+1)-1) \ (\text{fun } a \rightarrow \text{fib_cps } ((m+1)-2) \ (\text{fun } b \rightarrow k \ (a+b)))$

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

```
if m+1 = 1 then k 1 else fib_cps ((m+1)-1) (fun a → fib_cps ((m+1)-2) (fun b → k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

```
if m+1 = 1 then k 1 else fib_cps ((m+1)-1) (fun a → fib_cps ((m+1)-2) (fun b → k (a+b)))
```

$\equiv$ (arithmetic) ...

```
if m+1 = 1 then k 1 else fib_cps m (fun a → fib_cps (m-1) (fun b → k (a+b)))
```

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k \ (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k \ (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k \ (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k \ (\text{fib } (m+1))$.

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

```
if m+1 = 1 then k 1 else fib_cps m (fun a → fib_cps (m-1) (fun b → k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

```
if m+1 = 1 then k 1 else fib_cps m (fun a → fib_cps (m-1) (fun b → k (a+b)))
```

```
≡ (inductive assumption for m-1 and k = (fun b → k (a+b))) ...
```

```
if m+1 = 1 then k 1 else fib_cps m (fun a → (fun b → k (a+b)) (fib (m-1)))
```

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

```
if m+1 = 1 then k 1 else fib_cps m (fun a → (fun b → k (a+b)) (fib (m-1)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

```
if m+1 = 1 then k 1 else fib_cps m (fun a → (fun b → k (a+b)) (fib (m-1)))  
≡ (inductive assumption for m and k = (fun a → (fun b → k (a+b)) (fib (m-1)))) ...  
if m+1 = 1 then k 1 else (fun a → (fun b → k (a+b)) (fib (m-1))) (fib m)
```

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

```
if m+1 = 1 then k 1 else (fun a → (fun b → k (a+b)) (fib (m-1))) (fib m)
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

```
if m+1 = 1 then k 1 else (fun a → (fun b → k (a+b)) (fib (m-1))) (fib m)
```

$\equiv$ (beta reduction $\times 2$) ...

```
if m+1 = 1 then k 1 else k (fib m + fib (m-1))
```

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

```
if m+1 = 1 then k 1 else k (fib m + fib (m-1))
```

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

```
if m+1 = 1 then k 1 else k (fib m + fib (m-1))
≡ (if e1 then k e2 else k e3 ≡ k (if e1 then e2 else e3)) ...
k (if m+1 = 1 then 1 else fib m + fib (m-1))
```

Proof

By strong induction on m .

```
let rec fib m =
  if m = 0 then 1
  else if m = 1 then 1
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =
  if m = 0 then k 1
  else if m = 1 then k 1
  else fib_cps (m-1) (fun a →
    fib_cps (m-2) (fun b →
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

$k (\text{if } m+1 = 1 \text{ then } 1 \text{ else fib } m + \text{fib } (m-1))$

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

$k (\text{if } m+1 = 1 \text{ then } 1 \text{ else fib } m + \text{fib } (m-1))$

$\equiv (\text{definition of fib}) \dots$

$k (\text{fib } (m+1))$

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

$k (\text{fib } (m+1))$

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Correctness of CPS conversion for fib

Source to
source

CPS



D17n

CPS +
D17n

Mutual
recursion

Claim

For all $m \geq 0$,
for all $k : \text{int} \rightarrow \text{int}$,
 $\text{fib_cps } m \ k = k (\text{fib } m)$.

Base case ($m = 0$): $\text{fib_cps } 0 \ k = k \ 1 = k (\text{fib } 0)$.

Inductive step:

Assume for all $n \leq m$, $k (\text{fib } n) = \text{fib_cps } n \ k$.

We want to show: $\text{fib_cps } (m+1) \ k = k (\text{fib } (m+1))$.

$k (\text{fib } (m+1))$

QED

Proof

By strong induction on m .

```
let rec fib m =  
  if m = 0 then 1  
  else if m = 1 then 1  
  else fib (m-1) + fib (m-2)
```

```
let rec fib_cps m k =  
  if m = 0 then k 1  
  else if m = 1 then k 1  
  else fib_cps (m-1) (fun a →  
    fib_cps (m-2) (fun b →  
      k (a+b)))
```

NB: We approximate OCaml functions by mathematical functions, ignoring side effects etc.

Defunctionalization

Defunctionalization properties

Source to
source

CPS

D17n



CPS +
D17n

Mutual
recursion

Defunctionalized programs have some useful properties:

No higher-order functions

All values are data



All control-flow is first order

Every function is named

Defunctionalization: example

Source to
source

CPS

D17n



CPS +
D17n

Mutual
recursion

1. Add a constructor to `fn` for each `fun`
2. Replace each `fun` with its constructor
3. Add a case to `apply` for each `fun`
4. Replace each application `p x` with `apply p x`

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if p x  
    then x :: filter p xs  
    else filter p xs  
  
let f l y =  
  filter (fun x → x < 3) l  
  @ filter (fun x → x > y) l
```

```
type fn = Lt_three  
       | Gt of int
```

```
let apply fn x =  
  match fn, x with  
  | Lt_three, x → x < 3  
  | Gt y, x → x > y
```

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if apply p x  
    then x :: filter p xs  
    else filter p xs
```

```
let f l y =  
  filter Lt_three l  
  @ filter (Gt y) l
```

Defunctionalization: example

Source to
source

CPS

D17n



CPS +
D17n

Mutual
recursion

1. Add a constructor to `fn` for each `fun`
2. Replace each `fun` with its constructor
3. Add a case to `apply` for each `fun`
4. Replace each application `p x` with `apply p x`

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if p x  
    then x :: filter p xs  
    else filter p xs  
  
let f l y =  
  filter (fun x → x < 3) l  
  @ filter (fun x → x > y) l
```

```
type fn = Lt_three  
       | Gt_of int
```

```
let apply fn x =  
  match fn, x with  
  | Lt_three, x → x < 3  
  | Gt y, x → x > y
```

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if apply p x  
    then x :: filter p xs  
    else filter p xs
```

```
let f l y =  
  filter Lt_three l  
  @ filter (Gt y) l
```

Defunctionalization: example

Source to
source

CPS

D17n



CPS +
D17n

Mutual
recursion

1. Add a constructor to fn for each fun
2. Replace each fun with its constructor
3. Add a case to apply for each fun
4. Replace each application p x with apply p x

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if p x  
    then x :: filter p xs  
    else filter p xs  
  
let f l y =  
  filter (fun x → x < 3) l  
  @ filter (fun x → x > y) l
```

```
type fn = Lt_three  
       | Gt of int
```

```
let apply fn x =  
  match fn, x with  
  | Lt_three, x → x < 3  
  | Gt y, x → x > y
```

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if apply p x  
    then x :: filter p xs  
    else filter p xs
```

```
let f l y =  
  filter Lt_three l  
  @ filter (Gt y) l
```

Defunctionalization: example

Source to
source

CPS

D17n



CPS +
D17n

Mutual
recursion

1. Add a constructor to `fn` for each `fun`
2. Replace each `fun` with its constructor
3. Add a case to `apply` for each `fun`
4. Replace each application `p x` with `apply p x`

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if p x  
    then x :: filter p xs  
    else filter p xs  
  
let f l y =  
  filter (fun x → x < 3) l  
  @ filter (fun x → x > y) l
```

```
type fn = Lt_three  
       | Gt of int
```

```
let apply fn x =  
  match fn, x with  
  | Lt_three, x → x < 3  
  | Gt y, x → x > y
```

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if apply p x  
    then x :: filter p xs  
    else filter p xs
```

```
let f l y =  
  filter Lt_three l  
  @ filter (Gt y) l
```

Defunctionalization: example

Source to
source

CPS

D17n



CPS +
D17n

Mutual
recursion

1. Add a constructor to `fn` for each `fun`
2. Replace each `fun` with its constructor
3. Add a case to `apply` for each `fun`
4. Replace each application `p x` with `apply p x`

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if p x  
    then x :: filter p xs  
    else filter p xs  
  
let f l y =  
  filter (fun x → x < 3) l  
  @ filter (fun x → x > y) l
```

```
type fn = Lt_three  
       | Gt of int
```

```
let apply fn x =  
  match fn, x with  
  | Lt_three, x → x < 3  
  | Gt y, x → x > y
```

```
let rec filter p l =  
  match l with  
  | [] → []  
  | x :: xs →  
    if apply p x  
    then x :: filter p xs  
    else filter p xs
```

```
let f l y =  
  filter Lt_three l  
  @ filter (Gt y) l
```

Combining CPS & defunctionalization

Defunctionalizing fib_cps

Source to
source

```
let rec fib_cps m k =
```

```
  if m = 0 then k 1
```

```
  else if m = 1 then k 1
```

```
  else fib_cps (m-1) (fun a → (* K1 *))
```

```
    fib_cps (m-2) (fun b → (* K2 *))
```

```
    k (a+b)))
```

CPS

D17n

```
let fib_1 x = fib_cps x (fun x → x) (* ID *)
```

To defunctionalize `fib_cps`, define a constructor for each `fun`:

```
type cont = ID | K1 of int * cont | K2 of int * cont
```

CPS +
D17n

Constructor arguments are free variables, and we treat `k2` as free in `k1`:

```
let k2 = fun a b → k (a+b)
```

```
let k1 = fun a → fib_cps (m-2) (k2 a)
```

Mutual
recursion



Source to
source

Now define an apply function of type $\text{cont} \rightarrow \text{int} \rightarrow \text{int}$

```
type cont = ID | K1 of int * cont | K2 of int * cont
```

CPS

```
let rec apply_cont k v = match k, v with  
| ID, a → a  
| K1 (m, k), a → fib_cps_defun (m-2) (K2 (a, k))  
| K2 (a, k), b → apply_cont k (a+b)
```

D17n

and call `apply_cont` at every application of a continuation:

```
and fib_cps_defun m k =  
  if m = 0 then apply_cont k 1  
  else if m = 1 then apply_cont k 1  
  else fib_cps_defun (m-1) (K1 (m, k))
```

```
let fib_2 m = fib_cps_defun m ID
```

CPS +
D17n



Mutual
recursion

Correctness of fib_cps defunctionalization

Source to
source

CPS

Claim

Let $\langle c \rangle$ represent a continuation $c : \text{int} \rightarrow \text{int}$ constructed by `fib_cps`.
Then

D17n

$$\text{apply_cont } \langle c \rangle \ m \quad = \quad c \ m$$

and

$$\text{fib_cps } n \ c \quad = \quad \text{fib_cps_defun } n \ \langle c \rangle$$

CPS +
D17n



Mutual
recursion

(**Proof** left as an exercise)

Observation: continuations have list (stack) structure

Source to
source

CPS

```
type int_list =  
  NIL  
  | CONS of int * int_list
```

```
type cont =  
  ID (* 'Nil ' *)  
  | K1 of int * cont (* 'Cons ' *)  
  | K2 of int * cont (* 'Cons ' *)
```

D17n

Idea: replace `cont` with standard lists:

```
type tag = SUB2 of int (* K1: k (a+b) *)  
          | PLUS of int (* K2: fib_cps (m-2) (k2 a) *)
```

```
type tag_list_cont = tag list
```

CPS +
D17n



Mutual
recursion

fib_cps_defun revisited, using lists for continuations

Source to
source

CPS

D17n

CPS +
D17n

```
type tag = SUB2 of int | PLUS of int
type tag_list_cont = tag list
```

```
let rec apply_tag_list_cont k v = match k, v with
  | [], a → a
  | SUB2 m :: k, a → fib_cps_defun_tags (m-2) (PLUS a :: k)
  | PLUS a :: k, b → apply_tag_list_cont k (a+b)
```

```
and fib_cps_defun_tags m k =
  if m = 0 then apply_tag_list_cont k 1
  else if m = 1 then apply_tag_list_cont k 1
  else fib_cps_defun_tags (m-1) (SUB2 m :: k)
```

```
let fib_3 m = fib_cps_defun_tags m []
```

Mutual
recursion

Mutual recursion

Mutual recursion $\rightsquigarrow$ single recursion

Source to
source

Mutual recursion can be eliminated using **indexing**.

CPS

Given a set of mutually-recursive functions:

```
let rec is_even n = n = 0 || is_odd (n - 1)
    and is_odd n = n < 0 && is_even (n - 1)
```

D17n

define an index datatype with one constructor for each function:

```
type eo = Even | Odd
```

CPS +
D17n

and define a function that maps an index argument to a corresponding body:

```
let rec is f n =
  match f with
  | Even → n = 0 || is Odd (n - 1)
  | Odd → n < 0 && is Even (n - 1)
```

Mutual
recursion



Mutual recursion $\rightsquigarrow$ single recursion for fib

Source to
source

CPS

```
type state_type =  
| FIB (* for right-hand-sides starting with fib_ *)  
| APP (* for right-hand-sides starting with apply_ *)
```

```
type state = (state_type * int * tag_list_cont) → int
```

D17n

```
(* eval acts as either apply_tag_list_cont or fib_cps_defun_tags *)  
let rec eval = function  
| FIB, 0, k → eval (APP, 1, k)  
| FIB, 1, k → eval (APP, 1, k)  
| FIB, m, k → eval (FIB, m-1, SUB2 m :: k)  
| APP, a, SUB2 m :: k → eval (FIB, m-2, PLUS a :: k)  
| APP, b, PLUS a :: k → eval (APP, a+b, k)  
| APP, a, [] → a
```

CPS +
D17n

```
let fib_4 m = eval (FIB, m, [])
```

Mutual
recursion



Eliminate tail recursion to obtain *The Fibonacci Machine*

Source to
source

CPS

D17n

CPS +
D17n

```
(* step : state → state *)
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
| _ → failwith "step : runtime error!"

let rec driver = function (* clearly tail recursive! *)
| APP, a, [] → a
| state → driver (step state)

(* fib_5 : int → int *)
let fib_5 m = driver (FIB, m, [])
```

Mutual
recursion



(This version makes the tail-recursive structure very explicit.)

Tracing of fib_5 4

Source to
source

FIB 4 []

CPS

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

D17n

CPS +
D17n

Mutual
recursion



Tracing of fib_5 4

Source to
source

FIB 4 []
FIB 3 [SUB2 4]

CPS

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

D17n

CPS +
D17n

Mutual
recursion



Tracing of fib_5 4

Source to
source

```
FIB 4 []  
FIB 3 [SUB2 4]  
FIB 2 [SUB2 3; SUB2 4]
```

CPS

```
let step = function  
| FIB, 0, k → (APP, 1, k)  
| FIB, 1, k → (APP, 1, k)  
| FIB, m, k → (FIB, m-1, SUB2 m :: k)  
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)  
| APP, b, PLUS a :: k → (APP, a+b, k)
```

D17n

CPS +
D17n

Mutual
recursion



Tracing of fib_5 4

Source to
source

```
FIB 4 []  
FIB 3 [SUB2 4]  
FIB 2 [SUB2 3; SUB2 4]  
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
```

CPS

```
let step = function  
| FIB, 0, k → (APP, 1, k)  
| FIB, 1, k → (APP, 1, k)  
| FIB, m, k → (FIB, m-1, SUB2 m :: k)  
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)  
| APP, b, PLUS a :: k → (APP, a+b, k)
```

D17n

CPS +
D17n

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
```

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

D17n

CPS +
D17n

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

```
FIB 4 []  
FIB 3 [SUB2 4]  
FIB 2 [SUB2 3; SUB2 4]  
FIB 1 [SUB2 2; SUB2 3; SUB2 4]  
APP 1 [SUB2 2; SUB2 3; SUB2 4]  
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
```

D17n

CPS +
D17n

Mutual
recursion

```
let step = function  
| FIB, 0, k → (APP, 1, k)  
| FIB, 1, k → (APP, 1, k)  
| FIB, m, k → (FIB, m-1, SUB2 m :: k)  
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)  
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Tracing of fib_5 4

Source to
source

CPS

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
```

D17n

CPS +
D17n

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
```

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
```

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
```

D17n

CPS +
D17n

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
```

```
let step = function
```

```
| FIB, 0, k → (APP, 1, k)
```

```
| FIB, 1, k → (APP, 1, k)
```

```
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
```

```
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
```

```
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
FIB 2 [PLUS 3]
```

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
FIB 2 [PLUS 3]
FIB 1 [SUB2 2; PLUS 3]
```

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
FIB 2 [PLUS 3]
FIB 1 [SUB2 2; PLUS 3]
APP 1 [SUB2 2; PLUS 3]
```

```
let step = function
```

```
| FIB, 0, k → (APP, 1, k)
```

```
| FIB, 1, k → (APP, 1, k)
```

```
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
```

```
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
```

```
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
FIB 2 [PLUS 3]
FIB 1 [SUB2 2; PLUS 3]
APP 1 [SUB2 2; PLUS 3]
FIB 0 [PLUS 1; PLUS 3]
```

```
let step = function
```

```
| FIB, 0, k → (APP, 1, k)
```

```
| FIB, 1, k → (APP, 1, k)
```

```
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
```

```
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
```

```
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Mutual
recursion



Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
FIB 2 [PLUS 3]
FIB 1 [SUB2 2; PLUS 3]
APP 1 [SUB2 2; PLUS 3]
FIB 0 [PLUS 1; PLUS 3]
APP 1 [PLUS 1; PLUS 3]
```

```
let step = function
```

```
| FIB, 0, k → (APP, 1, k)
```

```
| FIB, 1, k → (APP, 1, k)
```

```
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
```

```
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
```

```
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
FIB 2 [PLUS 3]
FIB 1 [SUB2 2; PLUS 3]
APP 1 [SUB2 2; PLUS 3]
FIB 0 [PLUS 1; PLUS 3]
APP 1 [PLUS 1; PLUS 3]
APP 2 [PLUS 3]
```

```
let step = function
| FIB, 0, k → (APP, 1, k)
| FIB, 1, k → (APP, 1, k)
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Tracing of fib_5 4

Source to
source

CPS

D17n

CPS +
D17n

Mutual
recursion

```
FIB 4 []
FIB 3 [SUB2 4]
FIB 2 [SUB2 3; SUB2 4]
FIB 1 [SUB2 2; SUB2 3; SUB2 4]
APP 1 [SUB2 2; SUB2 3; SUB2 4]
FIB 0 [PLUS 1; SUB2 3; SUB2 4]
APP 1 [PLUS 1; SUB2 3; SUB2 4]
APP 2 [SUB2 3; SUB2 4]
FIB 1 [PLUS 2; SUB2 4]
APP 1 [PLUS 2; SUB2 4]
APP 3 [SUB2 4]
FIB 2 [PLUS 3]
FIB 1 [SUB2 2; PLUS 3]
APP 1 [SUB2 2; PLUS 3]
FIB 0 [PLUS 1; PLUS 3]
APP 1 [PLUS 1; PLUS 3]
APP 2 [PLUS 3]
APP 5 []
```

```
let step = function
```

```
| FIB, 0, k → (APP, 1, k)
```

```
| FIB, 1, k → (APP, 1, k)
```

```
| FIB, m, k → (FIB, m-1, SUB2 m :: k)
```

```
| APP, a, SUB2 m :: k → (FIB, m-2, PLUS a :: k)
```

```
| APP, b, PLUS a :: k → (APP, a+b, k)
```

Source to
source

CPS

We turned the recursive fib into a function that **uses no OCaml stack space**

D17n

The transformed fib function **carries its own stack** as an extra argument

We **transformed fib incrementally**, with each step easily proved correct

CPS +
D17n

Mutual
recursion



Next time: application to **interpreter 0**