Nested CPR

How to Keep an MR Intern Busy

Sebastian Graf, `sebastian.graf@kit.edu`

8 July 2019

Karlsruhe Institute of Technology

CPR – Motivation

```
fac :: Int -> Int
fac 1 = 1
fac n = n * fac (n-1)
```

CPR – Motivation

```
fac :: Int -> Int
fac (I# n) = case n of
  1# -> I# 1#
  _  -> case fac (I# (n -# 1#)) of
    I# m -> I# (n *# m)
```

CPR – Motivation

```
fac :: Int -> Int
```

Worker/Wrapper Transformation

```
f x = <e>
```

$$\Downarrow$$

```
f x = case fw x of
  (# a, b ... #) -> C a b ...
fw x = case <e> of
  C a b ... -> (# a, b, ... #)
```

Worker/Wrapper Transformation

```
fac (I# n) = case n of
  1# -> I# 1#
  _  -> case fac (I# (n -# 1#)) of
    I# m -> I# (n *# m)
```

Worker/Wrapper Transformation

```
fac n = I# (facw n)
facw (I# n) = case (case n of
  1# -> I# 1#
  _  -> case fac (I# (n -# 1#)) of
    I# m -> I# (n *# m)) of
  I# res -> res
```

Worker/Wrapper Transformation

```
fac n = I# (facw n)
facw (I# n) = case n of
  1# ->
    case I# 1# of I# res -> res
  _  -> case fac (I# (n -# 1#)) of
    I# m ->  case I# (n *# m) of
      I# res -> res
```

Worker/Wrapper Transformation

```
fac n = I# (facw n)
facw (I# n) = case n of
  1# -> 1#
  _  -> case fac (I# (n -# 1#)) of
    I# m -> (n *# m)
```

Worker/Wrapper Transformation

```
fac n = I# (facw n)
facw (I# n) = case n of
  1# -> 1#
  _  -> facw (I# (n -# 1#))
```

Worker/Wrapper Transformation

```
fac (I# n) = I# (facw n)
facw n = case n of
  1# -> 1#
  _  -> facw (n -# 1#)
```

When is this beneficial?

```haskell
data E = Add E E | Lit Int
eval (Add l r) = eval l + eval r
eval (Lit n) = n
```

When is this beneficial?

```haskell
data E = Add E E | Lit Int
eval e = Int# (evalw e)
evalw (Add l r) = evalw l +# evalw r
evalw (Lit n) = case n of I# n' -> n'
```

When is this beneficial?

```
data E = Add E E | Lit Int
eval e = Int# (evalw e)
evalw (Add l r) = evalw l +# evalw r
evalw (Lit n) = case n of I# n' -> n'

sum $ map (eval . Lit) [1..1000000]
```

Nested CPR – Motivation

```haskell
facIO :: Int -> IO Int
facIO 1 = pure 1
facIO n = do
  m <- facIO (n-1)
  pure (n * m)
```

Nested CPR – Motivation

```
facIO :: Int -> World -> (World, Int)
facIO 1 s = (s, 1)
facIO n s =
  let (s', m) = facIO (n-1) s in
  (s', n * m)
```

Nested CPR – Motivation

```
facIO :: Int -> World -> (World, Int)
facIO (I# n) s = case facIOw n s of
  (# s, n #) -> (s, n)
facIOw  :: Int# -> World -> (# World, Int #)
facIOw  1# s = (# s, 1 #)
facIOw  n s = case facIOw (n -# 1#) s of
  (# s', m #) -> (# s', I# n * m #)
```

Nested CPR – Motivation

```
facIO :: Int -> World -> (World, Int)
facIO (I# n) s = case facIOw n s of
  (# s, n #) -> (s, n)
facIOw  :: Int# -> World -> (# World, Int #)
facIOw  1# s = (# s, I# 1 #)
facIOw  n s = case facIOw (n -# 1#) s of
  (# s', I# m #) -> (# s', I# (n *# m) #)
```

Nested CPR – Motivation

```
facIO :: Int -> World -> (World, Int)
facIO (I# n) s = case facIOww n s of
  (# s, n #) -> (s, I# n)
facIOww :: Int# -> World -> (# World, Int# #)
facIOww 1# s = (# s, 1# #)
facIOww n s = case facIOww (n -# 1#) s of
  (# s', m #) -> (# s', n *# m #)
```

When to apply this?

```haskell
f :: Int -> (Int, Int)
f x = (x-1, x+1)
```

Careful!

- Divergence

When to apply this?

Careful!

- Divergence

```
f :: Int -> (Int, Int)
f x = (x-1, x+1)

f' :: Int -> (Int, Int)
f' (I# x) = case f'w x of
  (# a, b #) -> (I# a, I# b)
f'w x = (# x -# 1#, x +# 1# #)
```

When to apply this?

Careful!

- Divergence

```haskell
f :: Int -> (Int, Int)
f x = (x-1, x+1)

f' :: Int -> (Int, Int)
f' (I# x) = case f'w x of
  (# a, b #) -> (I# a, I# b)
f'w x = (# x -# 1#, x +# 1# #)

> f (error "💣") `seq` ()
()
> f' (error "💣") `seq` ()
💣
```

When to apply this?

Careful!

- Divergence
- Speculation

```haskell
f :: Int -> (Int, Int)
f x = x `seq` (x-1, g x)

g :: Int -> Int
g n = 0
g n = n + g (n-1)
```

When to apply this?

Careful!

- Divergence
- Speculation

```
f :: Int -> (Int, Int)
f x = case fw x of
  (# a, b #) -> (I# a, I# b)
fw x = (# x -# 1#, gw x #)

gw :: Int# -> Int#
g n = 0#
g n = n +# g (n -# 1#)
```

When to apply this?

Careful!

- Divergence
- Speculation

```haskell
f :: Int -> (Int, Int)
f x = case fw x of
  (# a, b #) -> (I# a, I# b)
fw x = (# x -# 1#, gw x #)

gw :: Int# -> Int#
g n = 0#
g n = n +# g (n -# 1#)

> sum $ map (fst . f) [0..1000000]
⌛
```

A different angle

```haskell
data E = Add E E | Lit Int
eval (Add l r) = eval l + eval r
eval (Lit n) = n
```

A different angle

```
data E = Add E E | Lit Int
eval e = Int# (evalw e)
evalw (Add l r) = evalw l +# evalw r
evalw (Lit n) = case n of I# n' -> n'
```

A different angle

```haskell
data E = Add E E | Lit Int
eval e = Int# (evalw e)
evalw (Add l r) = evalw l +# evalw r
evalw (Lit n) = case n of I# n' -> n'

sum $ map (eval . Lit) [1..1000000]
```

A different angle

```
data E = Add E E | Lit Int
eval (Add l r) = eval l + eval r
eval (Lit n) = n
```

A different angle

```
data E = Add E E | Lit Int
eval (Add l r) = I# (eval_ub l #+ eval_ub r)
eval (Lit n) = n
eval_ub (Add l r) = eval_ub l #+ eval_ub r
eval_ub (Lit n) = case n of I# n' -> n'

sum $ map (eval . Lit) [1..1000000]
```

$\Longrightarrow$ “Return-pattern specialisation”

Other Possible Topics

- Reliable Constructor Specialisation for functions/lambdas
    - Zero-overhead `concat` in Stream Fusion!
- Pattern-match Checking
- Backend work?
- Frontend work?