Effects for Less

Alexis King

ZuriHac 2020

Feb 2016 ○ Alexis starts writing Haskell.

Feb 2016 ○ Alexis starts writing Haskell.

→ Mostly a CRUD app.

Feb 2016 ○ Alexis starts writing Haskell.

→ Mostly a CRUD app.

→ Performance not that important.

Feb 2016 — Alexis starts writing Haskell.

- → Mostly a CRUD app.
- → Performance not that important.

Dec 2017 — Effect management is getting complicated.

Feb 2016 — Alexis starts writing Haskell.

→ Mostly a CRUD app.

→ Performance not that important.

Dec 2017 — Effect management is getting complicated.

→ Start exploring effect systems.

Feb 2016 — Alexis starts writing Haskell.

- → Mostly a CRUD app.
- → Performance not that important.

Dec 2017 — Effect management is getting complicated.

- → Start exploring effect systems.
- → Clean up `freer`, release as `freer-simple`.

Feb 2016 — Alexis starts writing Haskell.

- → Mostly a CRUD app.
- → Performance not that important.

Dec 2017 — Effect management is getting complicated.

- → Start exploring effect systems.
- → Clean up `freer`, release as `freer-simple`.

Feb 2018 — Break from writing Haskell to write Racket.

Feb 2016 — Alexis starts writing Haskell.

- → Mostly a CRUD app.
- → Performance not that important.

Dec 2017 — Effect management is getting complicated.

- → Start exploring effect systems.
- → Clean up `freer`, release as `freer-simple`.

Feb 2018 — Break from writing Haskell to write Racket.

Jul 2019 — Back to writing Haskell.

HASURA

HASURA

“ Realtime GraphQL on PostgreSQL”

“ Realtime GraphQL on PostgreSQL”

Secretly: a GraphQL to SQL JIT compiler

HASURA

" Realtime GraphQL on PostgreSQL"

Secretly: a GraphQL to SQL JIT compiler

“Realtime GraphQL on PostgreSQL”

Secretly: a GraphQL to SQL JIT compiler

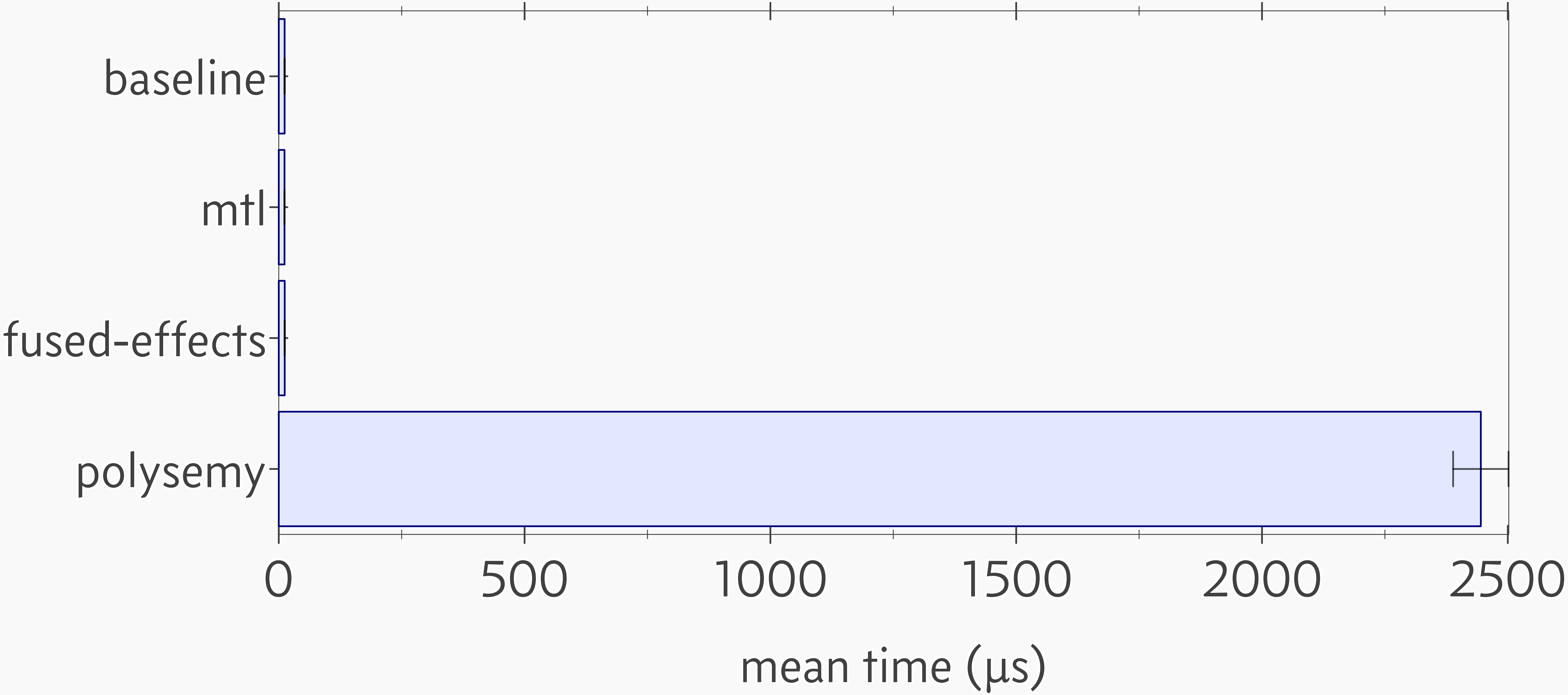
Performance is *really* important!

Can we afford to use an effect system?

BENCHMARKS

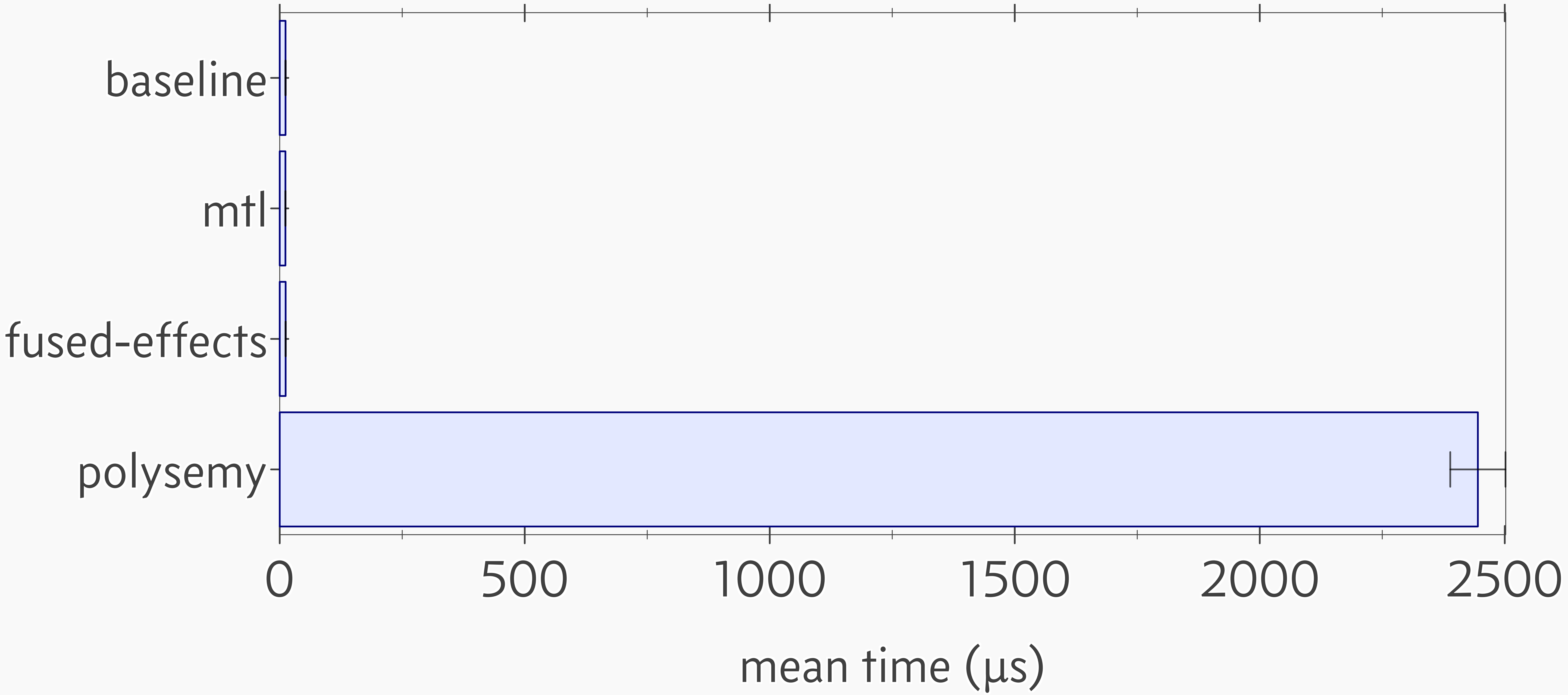
BENCHMARK: COUNTDOWN

BENCHMARK: COUNTDOWN



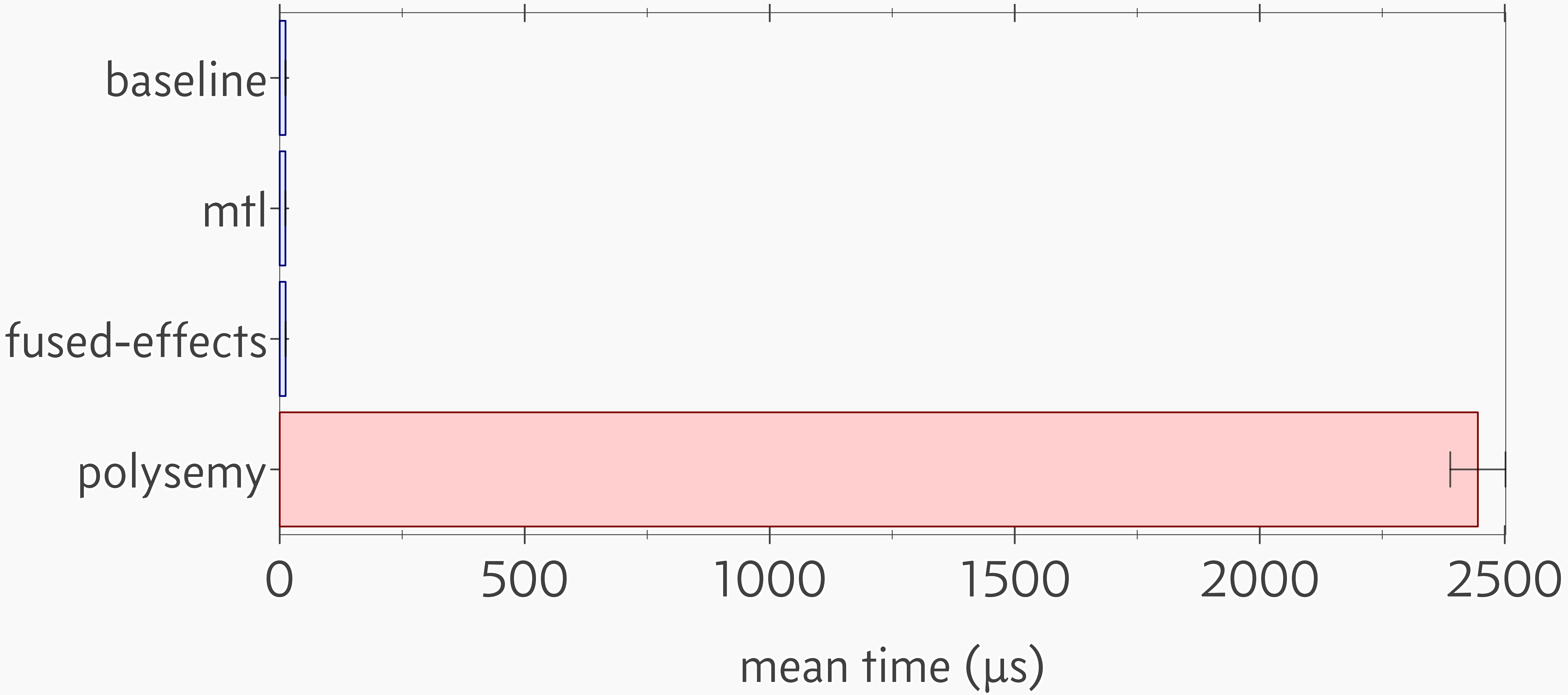
BENCHMARK: COUNTDOWN

(lower is better)



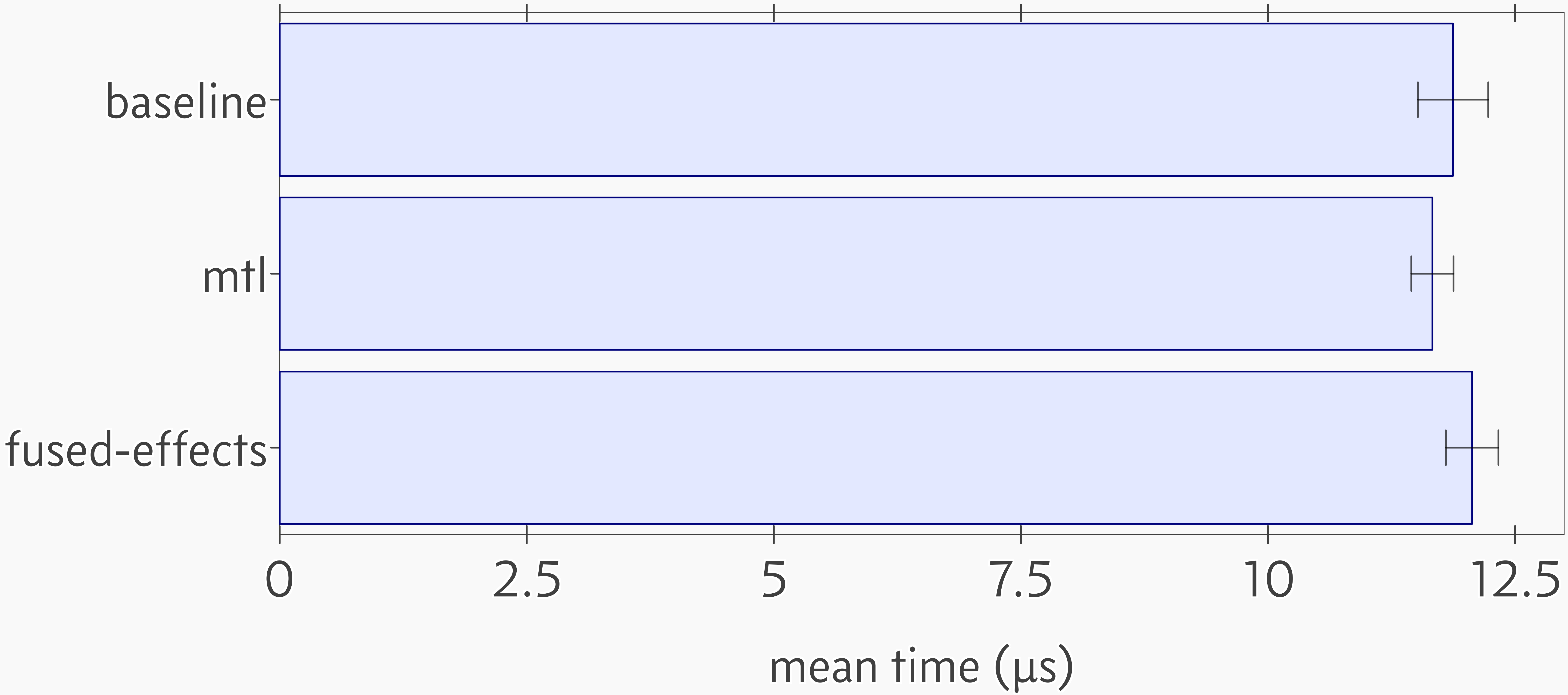
BENCHMARK: COUNTDOWN

(lower is better)



BENCHMARK: COUNTDOWN

(lower is better)



Can we afford to use an effect system?

Can we afford to use an effect system?

Answer: yes.

Just don't pick polysemy.

It’s never that simple!

QUESTIONS

QUESTIONS

1. What *is* the countdown benchmark?

QUESTIONS

1. What *is* the countdown benchmark?
2. What is the “baseline” implementation?

QUESTIONS

1. What *is* the countdown benchmark?
2. What is the “baseline” implementation?
3. Are these differences even meaningful?

QUESTIONS

1. What *is* the countdown benchmark?
2. What is the “baseline” implementation?
3. Are these differences even meaningful?
4. Why does polysemy do *that* much worse?

THE COUNTDOWN MICROBENCHMARK

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

THE COUNTDOWN MICROBENCHMARK

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

COUNTDOWN MICROBENCHMARK: BASELINE

```
countDown :: Int → (Int, Int)
countDown = program

program :: Int → (Int, Int)
program n = if n ⩽ 0 then (n, n)
            else program (n - 1)
```

COUNTDOWN MICROBENCHMARK: BASELINE

```
countDown :: Int → (Int, Int)
countDown = program

program :: Int → (Int, Int)
program n = if n ≤ 0 then (n, n)
                     else program (n - 1)
```

COUNTDOWN MICROBENCHMARK: BASELINE

```
countDown :: Int → (Int, Int)
countDown = program

program :: Int → (Int, Int)
program n = if n ≤ 0 then (n, n)
            else program (n - 1)
```

This is incredibly synthetic!

This is incredibly synthetic!

Is countdown a bad benchmark?

IN DEFENSE OF MICROBENCHMARKS

IN DEFENSE OF MICROBENCHMARKS

REAL-WORLD BENCHMARKS

PROS

IN DEFENSE OF MICROBENCHMARKS

REAL-WORLD BENCHMARKS

PROS

→ Probably representative of *something*.

IN DEFENSE OF MICROBENCHMARKS

REAL-WORLD BENCHMARKS

PROS

→ Probably representative of *something*.
→ Unlikely to be a fluke/flaw of the benchmarking process.

IN DEFENSE OF MICROBENCHMARKS

REAL-WORLD BENCHMARKS

PROS

→ Probably representative of *something*.
→ Unlikely to be a fluke/flaw of the benchmarking process.

CONS

IN DEFENSE OF MICROBENCHMARKS

REAL-WORLD BENCHMARKS

PROS

- Probably representative of *something*.
- Unlikely to be a fluke/flaw of the benchmarking process.

CONS

- Really big!

IN DEFENSE OF MICROBENCHMARKS

REAL-WORLD BENCHMARKS

PROS

- → Probably representative of *something*.
- → Unlikely to be a fluke/flaw of the benchmarking process.

CONS

- → Really big!
- → Difficult to isolate costs.

IN DEFENSE OF MICROBENCHMARKS

REAL-WORLD BENCHMARKS

PROS

- Probably representative of *something*.
- Unlikely to be a fluke/flaw of the benchmarking process.

CONS

- Really big!
- Difficult to isolate costs.
- May be challenging to extrapolate results.

IN DEFENSE OF MICROBENCHMARKS

	REAL-WORLD BENCHMARKS	MICROBENCHMARKS
PROS	→ Probably representative of *something*. → Unlikely to be a fluke/flaw of the benchmarking process.	
CONS	→ Really big! → Difficult to isolate costs. → May be challenging to extrapolate results.	

IN DEFENSE OF MICROBENCHMARKS

	REAL-WORLD BENCHMARKS	MICROBENCHMARKS
PROS	→ Probably representative of *something*. → Unlikely to be a fluke/flaw of the benchmarking process.	→ Easy to isolate costs!
CONS	→ Really big! → Difficult to isolate costs. → May be challenging to extrapolate results.	

IN DEFENSE OF MICROBENCHMARKS

	REAL-WORLD BENCHMARKS	MICROBENCHMARKS
PROS	→ Probably representative of *something*. → Unlikely to be a fluke/flaw of the benchmarking process.	→ Easy to isolate costs! → Small enough to *completely* understand.
CONS	→ Really big! → Difficult to isolate costs. → May be challenging to extrapolate results.	

IN DEFENSE OF MICROBENCHMARKS

	REAL-WORLD BENCHMARKS	MICROBENCHMARKS
PROS	→ Probably representative of *something*. → Unlikely to be a fluke/flaw of the benchmarking process.	→ Easy to isolate costs! → Small enough to *completely* understand. → If thoroughly understood, can provide a useful cost model.
CONS	→ Really big! → Difficult to isolate costs. → May be challenging to extrapolate results.	

IN DEFENSE OF MICROBENCHMARKS

	REAL-WORLD BENCHMARKS	MICROBENCHMARKS
PROS	→ Probably representative of *something*. → Unlikely to be a fluke/flaw of the benchmarking process.	→ Easy to isolate costs! → Small enough to *completely* understand. → If thoroughly understood, can provide a useful cost model.
CONS	→ Really big! → Difficult to isolate costs. → May be challenging to extrapolate results.	→ Easy to measure the wrong thing!

IN DEFENSE OF MICROBENCHMARKS

	REAL-WORLD BENCHMARKS	MICROBENCHMARKS
PROS	→ Probably representative of *something*. → Unlikely to be a fluke/flaw of the benchmarking process.	→ Easy to isolate costs! → Small enough to *completely* understand. → If thoroughly understood, can provide a useful cost model.
CONS	→ Really big! → Difficult to isolate costs. → May be challenging to extrapolate results.	→ Easy to measure the wrong thing! → *Crucial* to understand the scope of results.

IN DEFENSE OF MICROBENCHMARKS

	REAL-WORLD BENCHMARKS	MICROBENCHMARKS
PROS	→ Probably representative of *something*. → Unlikely to be a fluke/flaw of the benchmarking process.	→ Easy to isolate costs! → Small enough to *completely* understand. → If thoroughly understood, can provide a useful cost model.
CONS	→ Really big! → Difficult to isolate costs. → May be challenging to extrapolate results.	→ Easy to measure the wrong thing! → *Crucial* to understand the scope of results. → Costs are not considered in context.

Effects are *particularly* hard to measure with real-world benchmarks!

Effects are *particularly* hard to measure with real-world benchmarks!

Effects are *particularly* hard to measure with real-world benchmarks!

Effects are *particularly* hard to measure with real-world benchmarks!

Effects are *particularly* hard to measure with real-world benchmarks!

Effects are *particularly* hard to measure with real-world benchmarks!

Effects are *particularly* hard to measure with real-world benchmarks!

What does countdown measure?

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

What does countdown measure?

```
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

```haskell
reallyExpensive :: MonadExpensive m ⇒ m Blah
reallyExpensive = colorGraph >> mineBitcoin >> compileHaskell
```

```haskell
reallyExpensive :: MonadExpensive m ⇒ m Blah
reallyExpensive = colorGraph >> mineBitcoin >> compileHaskell
```

call `reallyExpensive`

```haskell
reallyExpensive :: MonadExpensive m ⇒ m Blah
reallyExpensive = colorGraph >> mineBitcoin >> compileHaskell
```

call `reallyExpensive`

`colorGraph`
`mineBitcoin`
`compileHaskell`

```haskell
reallyExpensive :: MonadExpensive m ⇒ m Blah
reallyExpensive = colorGraph >> mineBitcoin >> compileHaskell
```

call `reallyExpensive`

`colorGraph`

`mineBitcoin`

`compileHaskell`

return to caller

```haskell
reallyExpensive :: MonadExpensive m ⇒ m Blah
reallyExpensive = colorGraph >> mineBitcoin >> compileHaskell
```

call `reallyExpensive`

```
colorGraph
mineBitcoin
compileHaskell
```

return to caller

```haskell
reallyExpensive :: MonadExpensive m ⇒ m Blah
reallyExpensive = colorGraph >> mineBitcoin >> compileHaskell
```

call reallyExpensive

colorGraph

mineBitcoin

compileHaskell

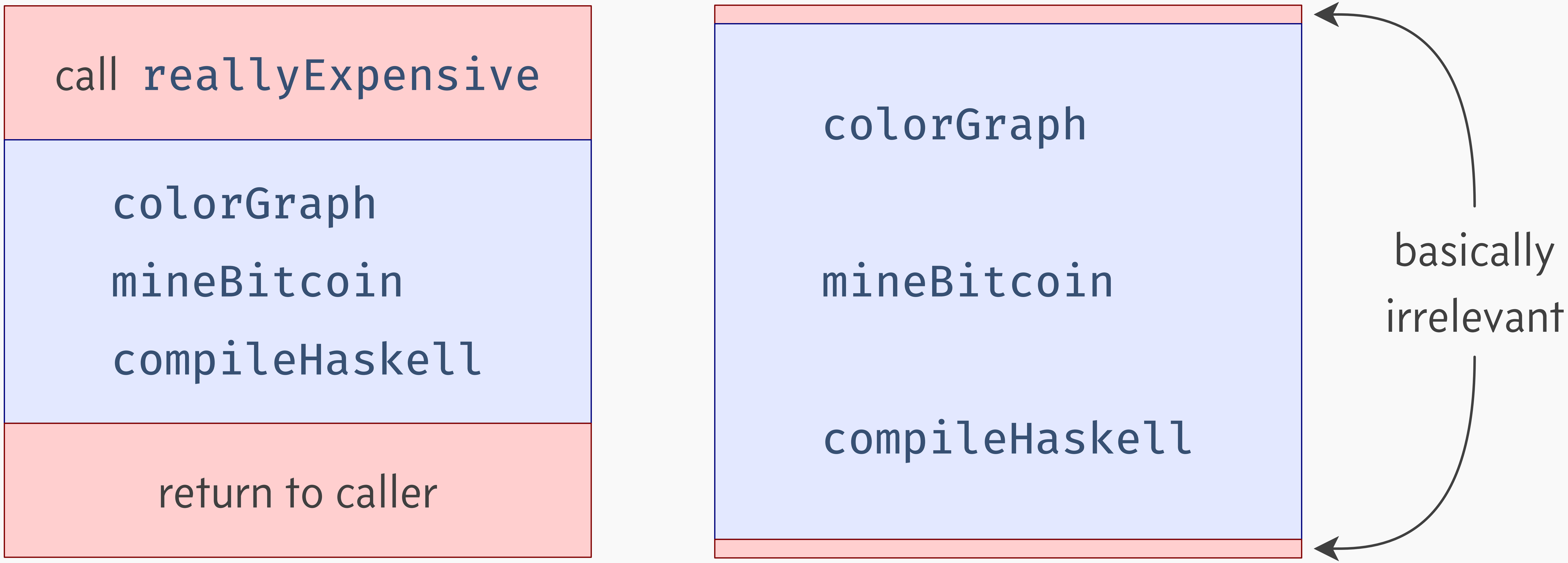
return to caller

colorGraph

mineBitcoin

compileHaskell

```haskell
reallyExpensive :: MonadExpensive m ⇒ m Blah
reallyExpensive = colorGraph >> mineBitcoin >> compileHaskell
```



call `get`

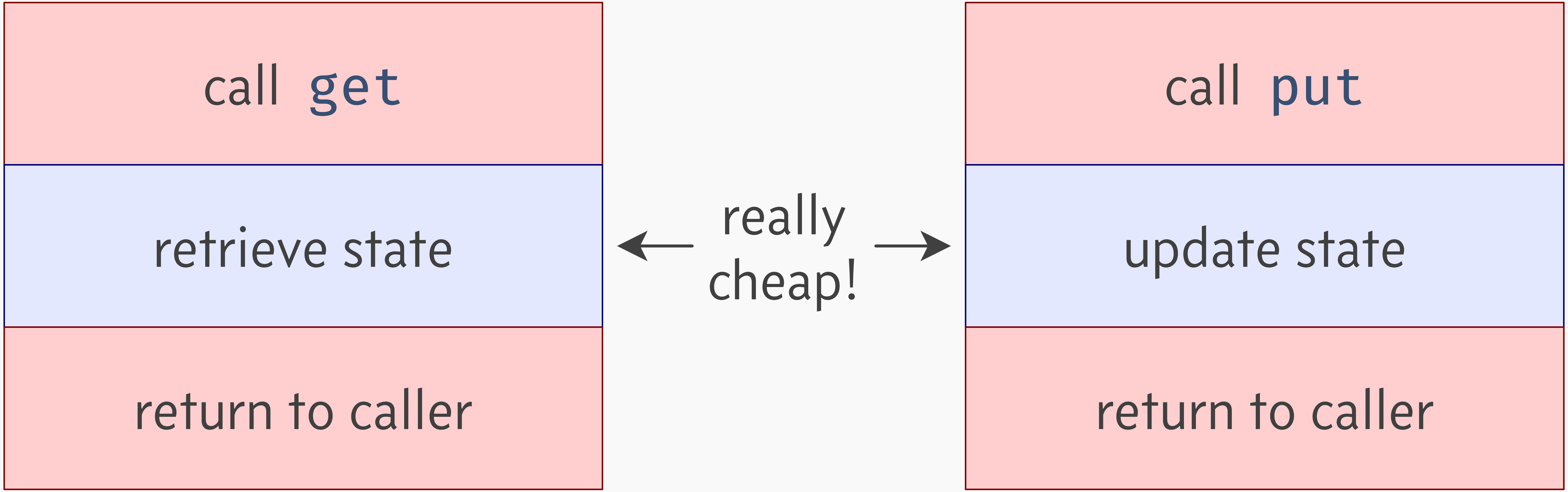
retrieve state

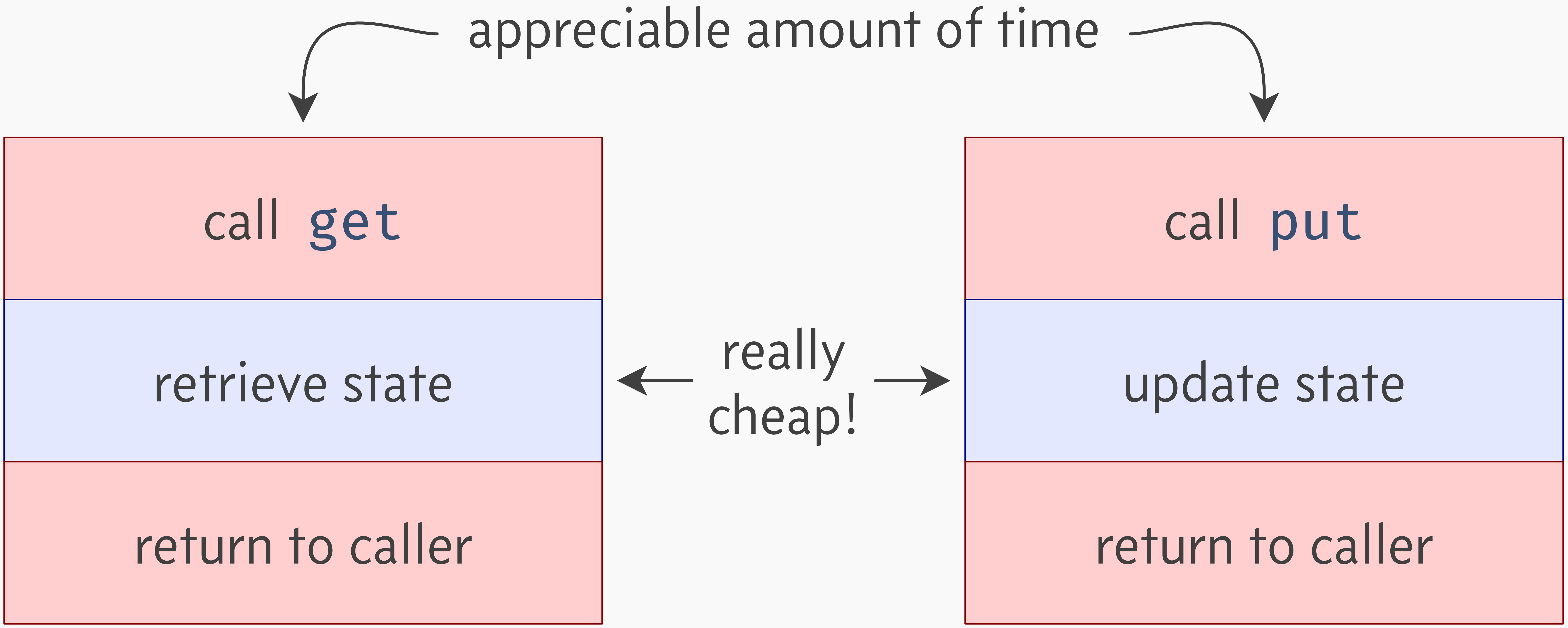
return to caller

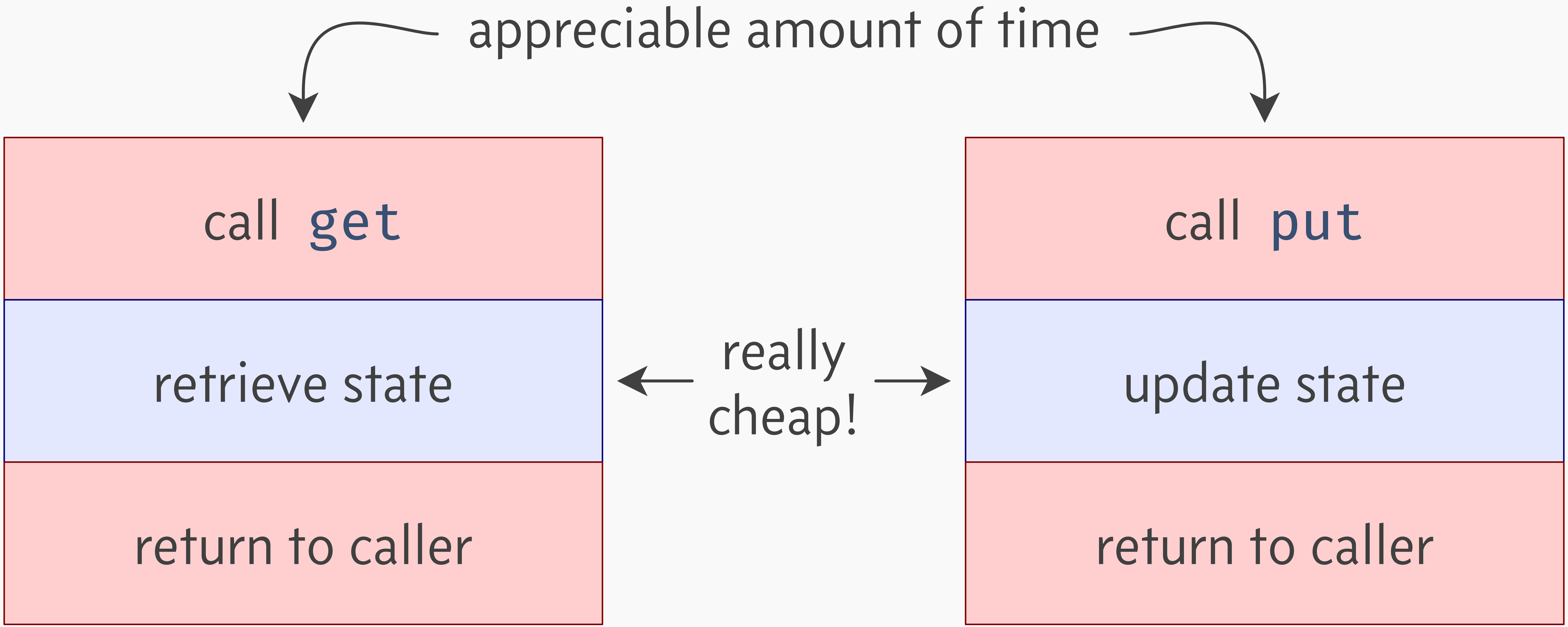
call `put`

update state

return to caller


call `get`
retrieve state
return to caller
really cheap!
call `put`
update state
return to caller


appreciable amount of time
call `get`
retrieve state
return to caller
really cheap!
call `put`
update state
return to caller



Countdown benchmarks *effect dispatch*.

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ≤ 0
               then return n
               else put (n - 1) >> program
```

```haskell
countDown :: Int -> (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m => m Int
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

```haskell
countDown :: Int → (Int, Int)
countDown initial = runState program initial

program :: MonadState Int m ⇒ m Int
program = get >>= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

`>>=` is an exceptionally common operation.

$\succ\!\!=$ is an exceptionally common operation.

```
do a ← f x
   b ← g y
   h a b
```

>>= is an exceptionally common operation.

```
do a ← f x
   b ← g y
   h a b
```

```
Foo <$> getX
    <*> getY
    <*> getZ
```

`>>=` is an exceptionally common operation.

```haskell
do a <- f x
   b <- g y
   h a b
```

```haskell
Foo <$> getX
    <*> getY
    <*> getZ
```

```haskell
mapM
traverse
sequence
```

>>= is an exceptionally common operation.

```
do a ← f x
   b ← g y
   h a b
```

```
Foo <$> getX
    <*> getY
    <*> getZ
```

`mapM`
`traverse`
`sequence`

`when`
`unless`
`replicateM`

RECAP

RECAP

1. Countdown is a microbenchmark.

RECAP

1. Countdown is a microbenchmark.
2. Theoretically, it's a valuable benchmark.

RECAP

1. Countdown is a microbenchmark.
2. Theoretically, it's a valuable benchmark.
3. It measures two things:

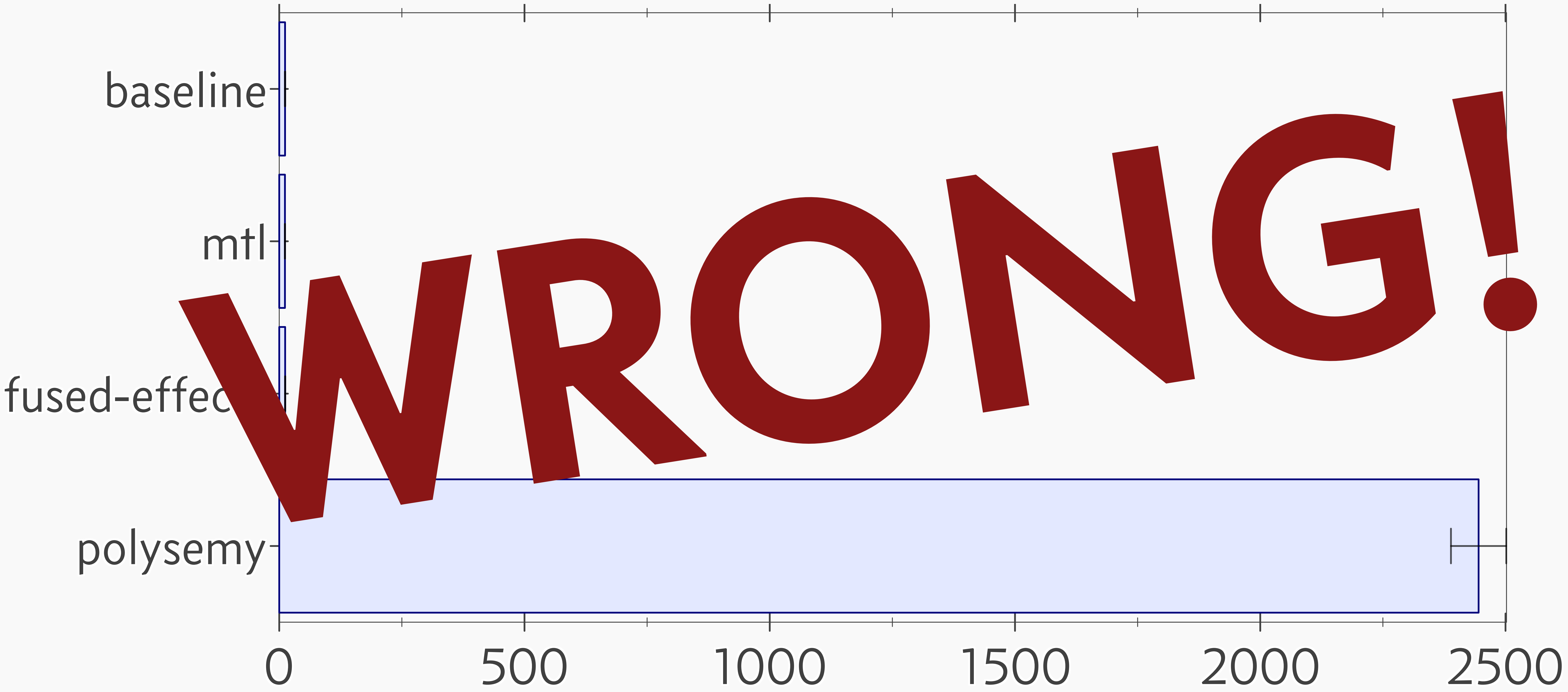
RECAP

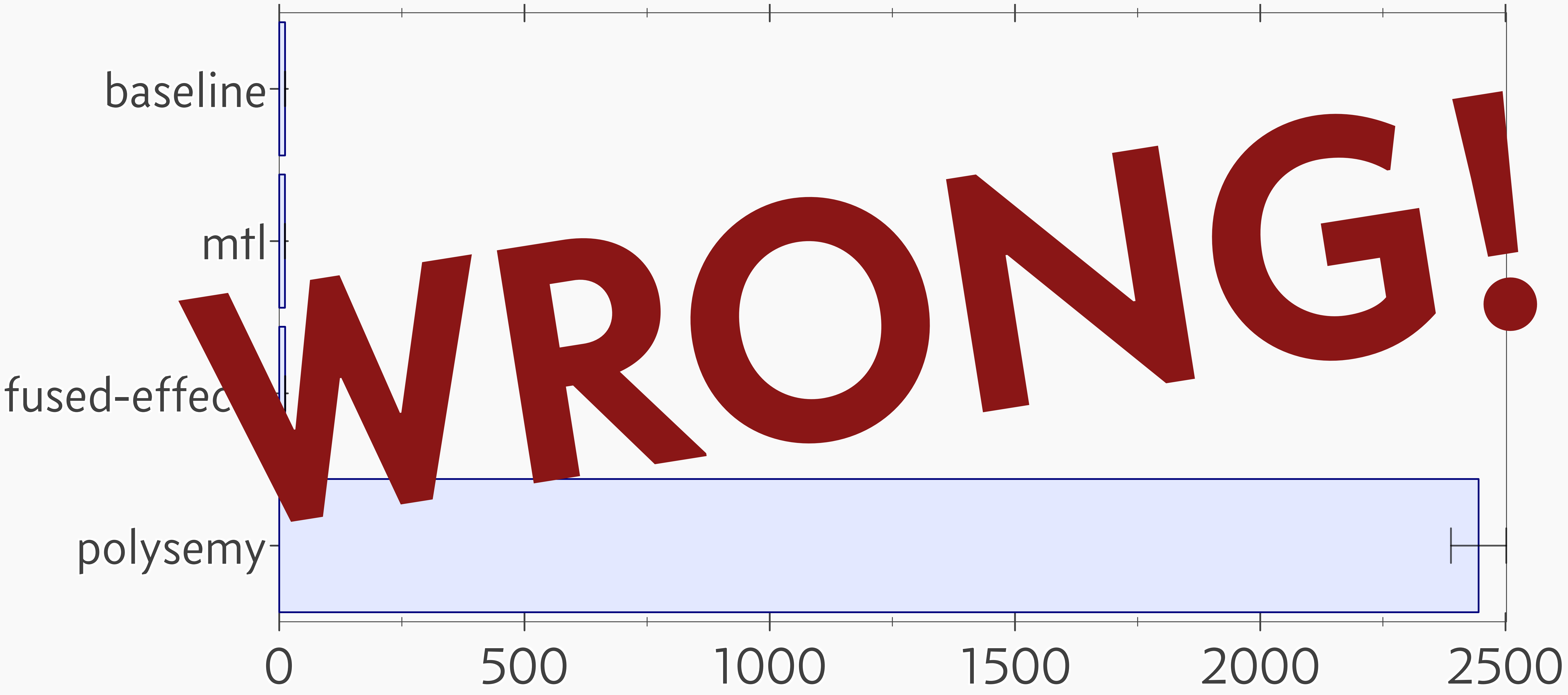
1. Countdown is a microbenchmark.
2. Theoretically, it's a valuable benchmark.
3. It measures two things:
    → …the cost of effect dispatch.

RECAP

1. Countdown is a microbenchmark.
2. Theoretically, it’s a valuable benchmark.
3. It measures two things:
    → …the cost of effect dispatch.
    → …the cost of `>>=`.

Why am I belaboring this point?


WRONG!
baseline
mtl
fused-effec
polysemy
0
500
1000
1500
2000
2500



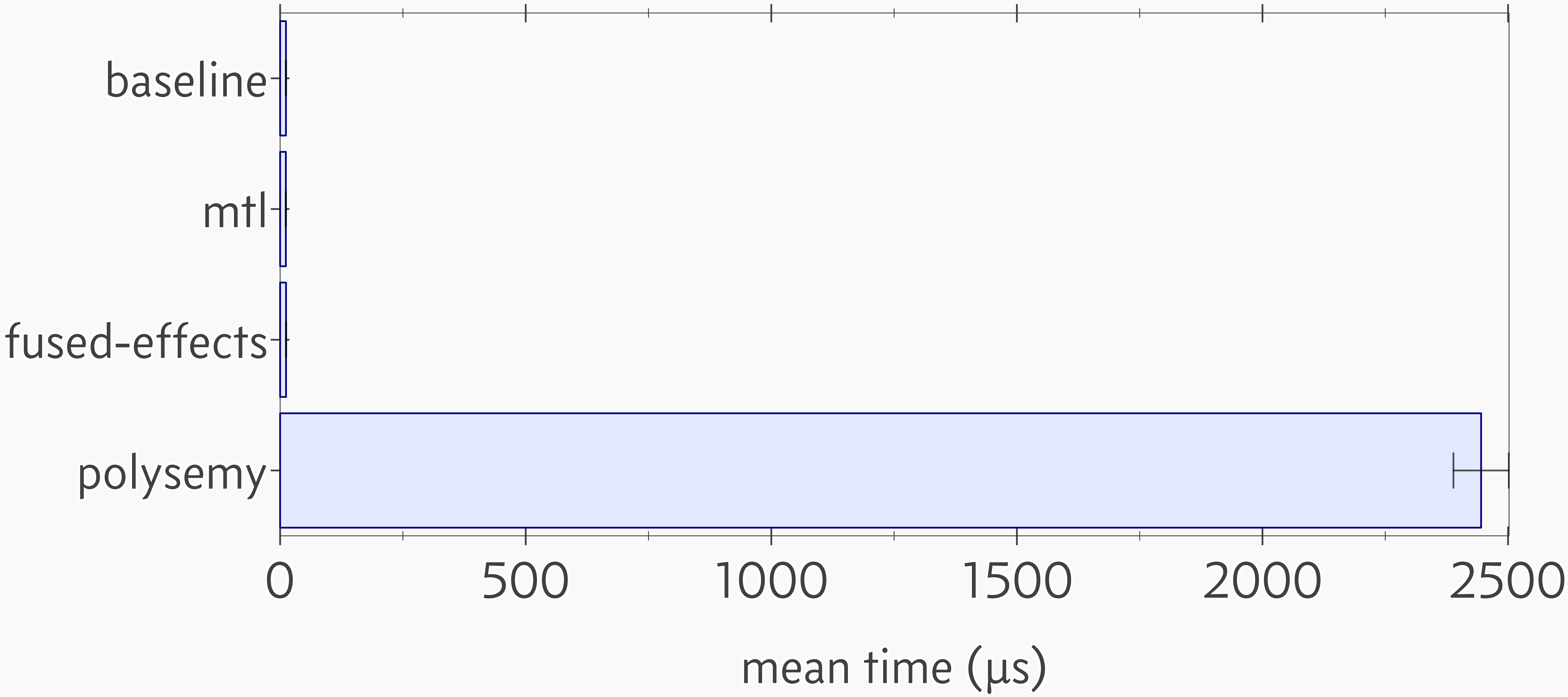
…or at least highly misleading.

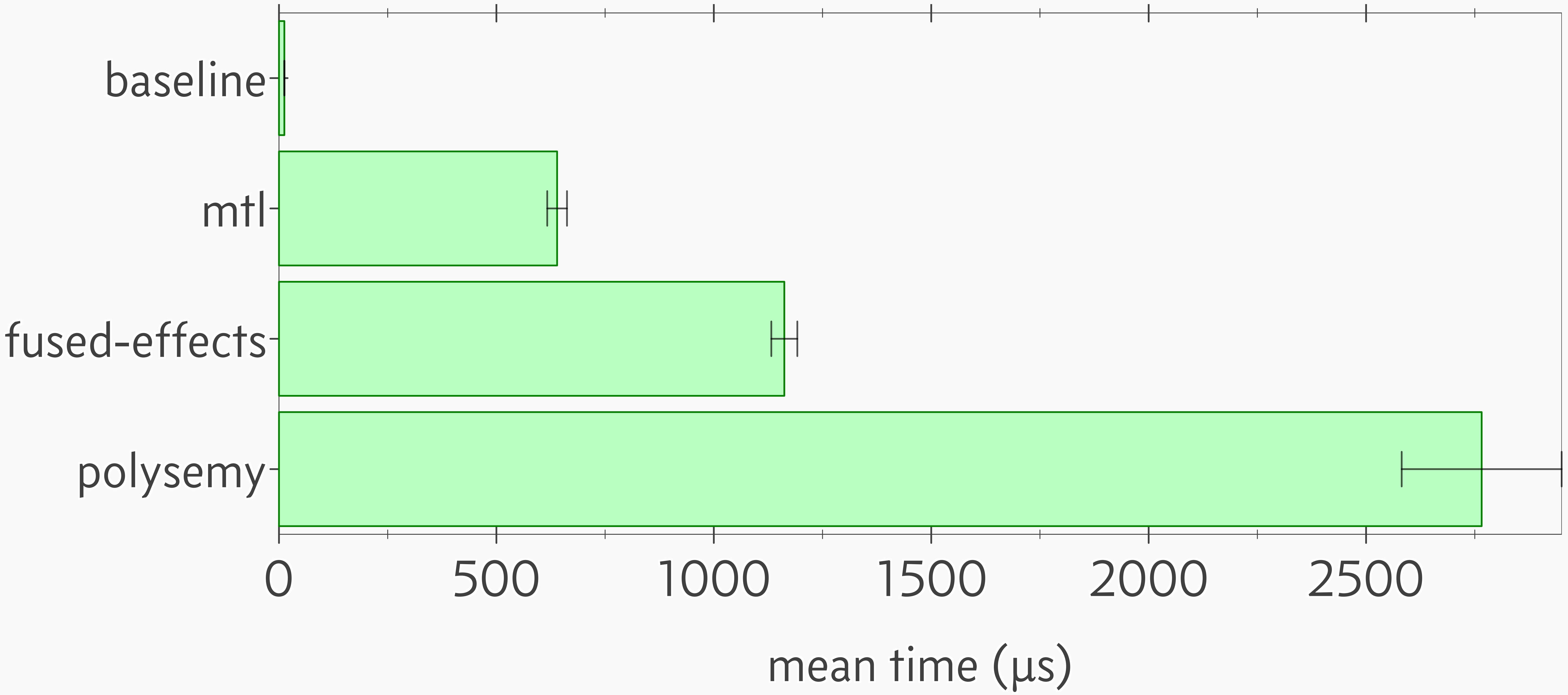
```haskell
module CountDown where
  countDown :: Int → (Int, Int)
  countDown initial = runState program initial

  program :: MonadState Int m ⇒ m Int
  program = do n ← get
               if n ≤ 0
                 then return n
                 else put (n - 1) >> program
```

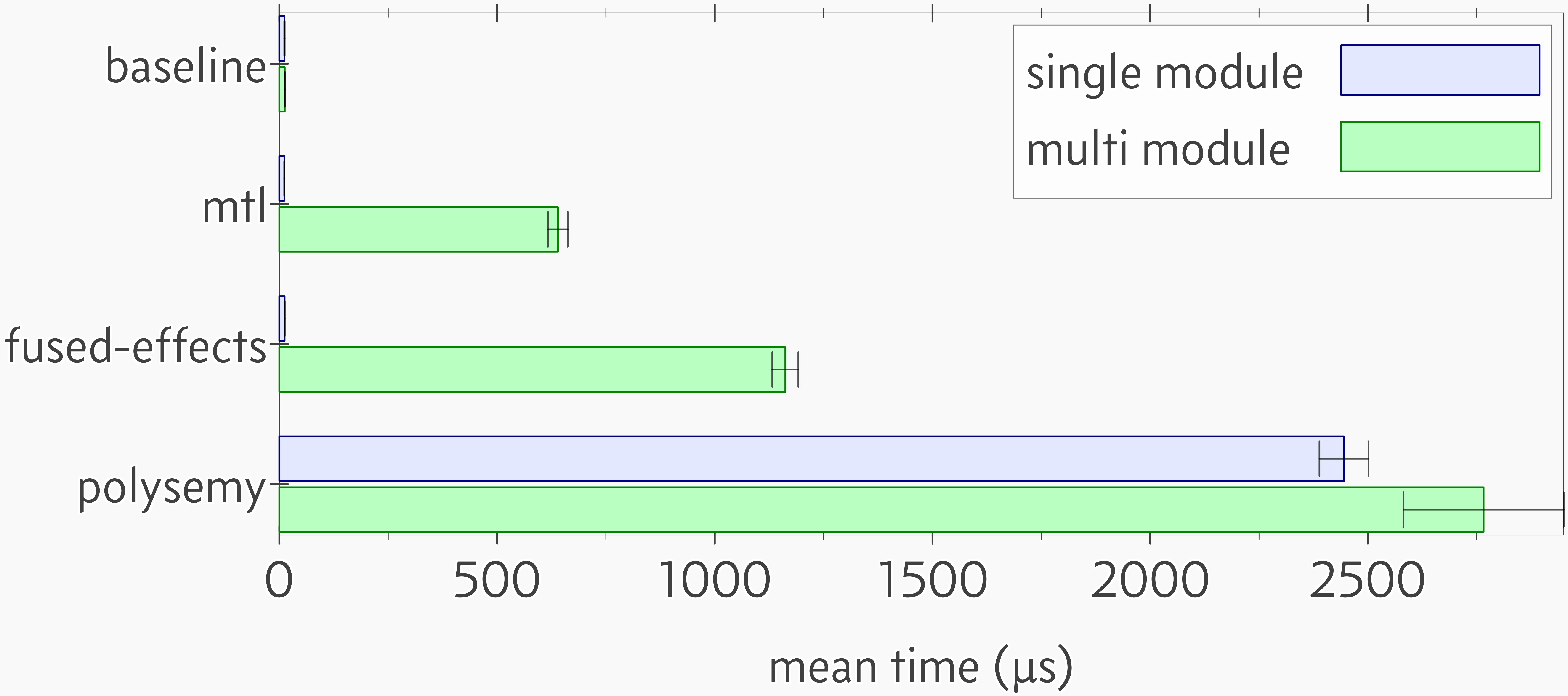
```haskell
module CountDown where
  import Program
  countDown :: Int → (Int, Int)
  countDown initial = runState program initial

module Program where
  program :: MonadState Int m ⇒ m Int
  program = do n ← get
               if n ≤ 0
                 then return n
                 else put (n - 1) >> program
```

```haskell
module CountDown where
  import Program
  countDown :: Int → (Int, Int)
  countDown initial = runState program initial

module Program where
  program :: MonadState Int m ⇒ m Int
  program = do n ← get
               if n ⩽ 0
                 then return n
                 else put (n - 1) >> program
```

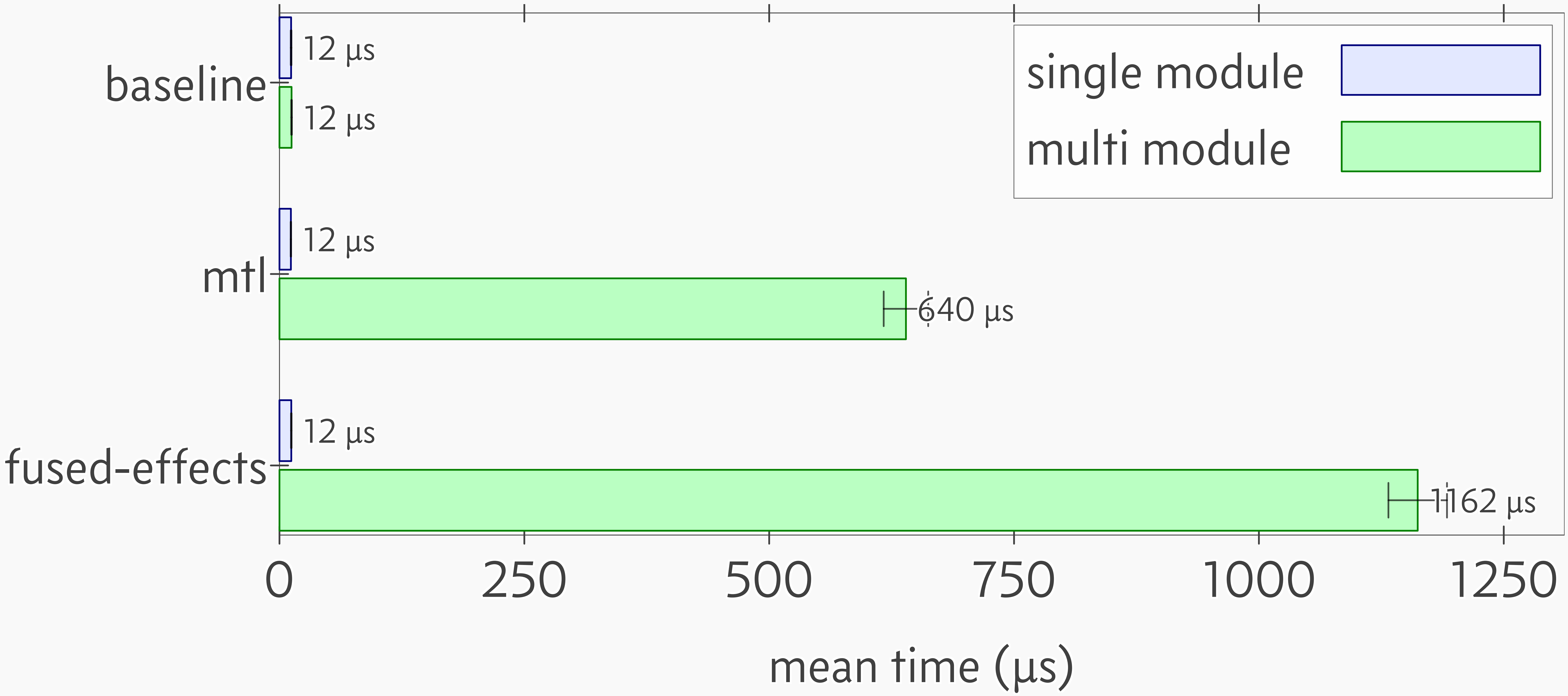
Surely this shouldn't change anything?


baseline
mtl
fused-effects
polysemy
0
500
1000
1500
2000
2500
mean time (μs)

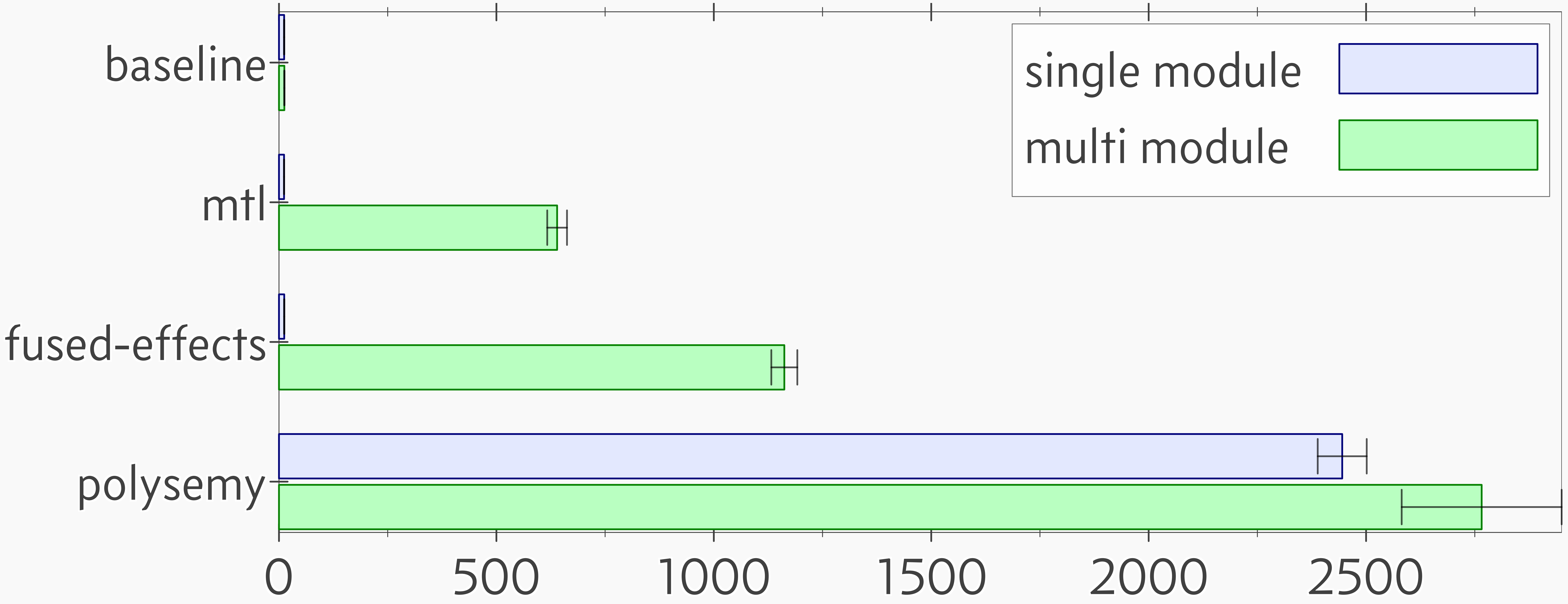

baseline
mtl
fused-effects
polysemy
0
500
1000
1500
2000
2500
mean time (μs)

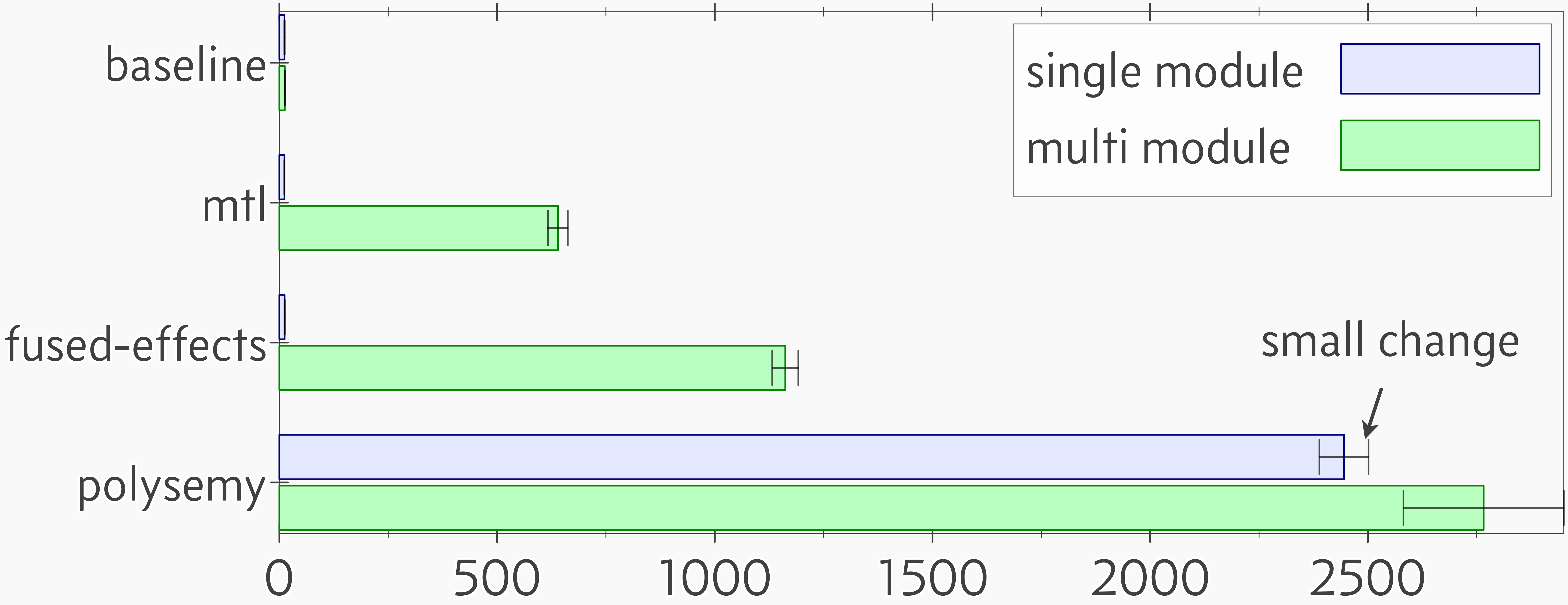


baseline
mtl
fused-effects
polysemy
single module
multi module
0
500
1000
1500
2000
2500
mean time (μs)

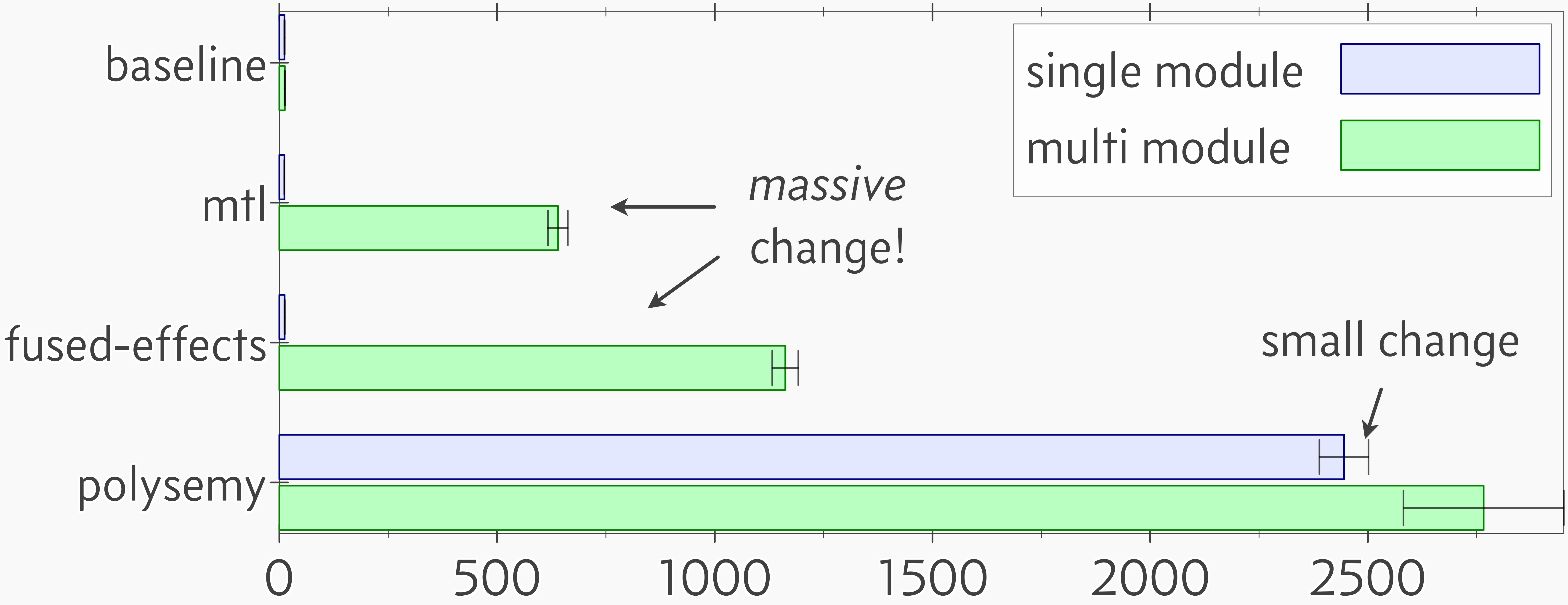

single module
multi module
baseline
mtl
fused-effects
12 μs
12 μs
12 μs
640 μs
12 μs
1162 μs
0
250
500
750
1000
1250
mean time (μs)

What happened?

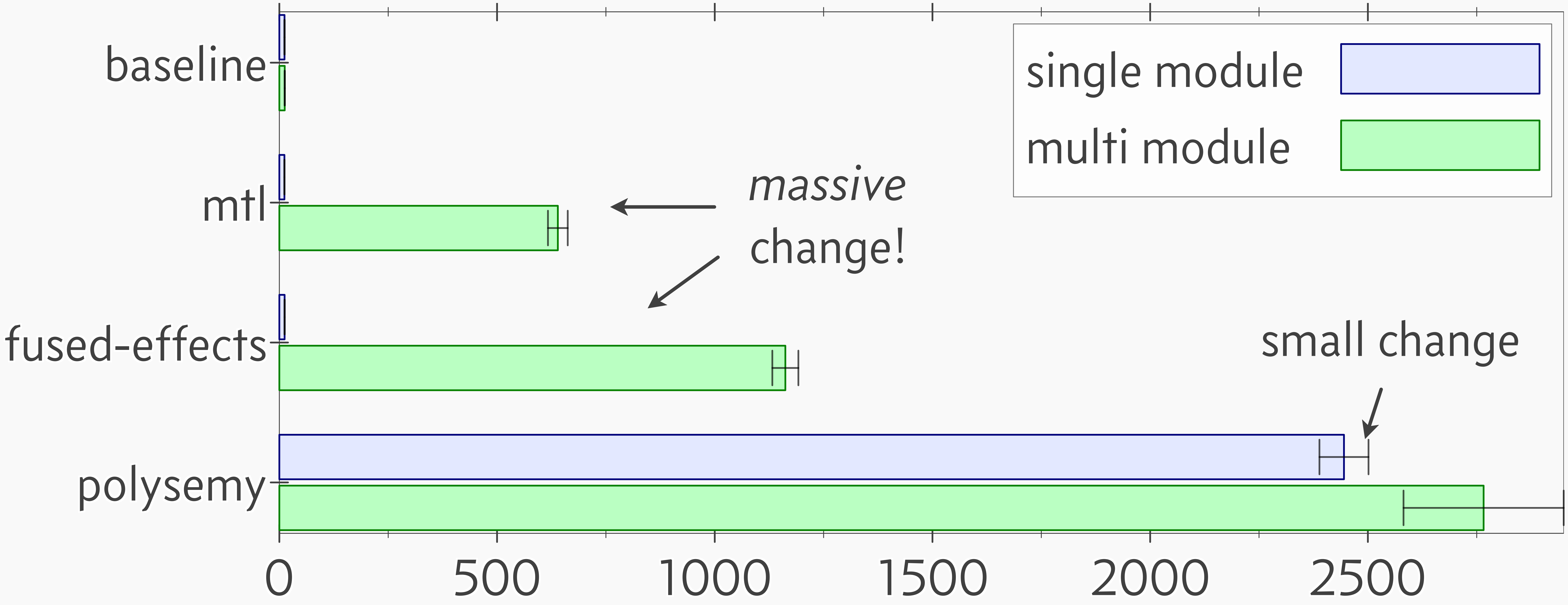

single module
multi module
baseline
mtl
fused-effects
polysemy
0
500
1000
1500
2000
2500



single module
multi module
baseline
mtl
fused-effects
polysemy
small change
0
500
1000
1500
2000
2500



baseline
mtl
fused-effects
polysemy
single module
multi module
massive
change!
small change
0
500
1000
1500
2000
2500



mtl and fused-effects are victims of the optimizer.

KEY TAKEAWAYS

KEY TAKEAWAYS

1. mtl and fused-effects *are* faster than polysemy…

KEY TAKEAWAYS

1. mtl and fused-effects *are* faster than polysemy…
2. …but are more reliant on compiler optimizations for best-case perf.

KEY TAKEAWAYS

1. mtl and fused-effects *are* faster than polysemy…
2. …but are more reliant on compiler optimizations for best-case perf.
3. Tiny program changes can have huge perf diffs!

① Monad transformers + typeclasses.

① Monad transformers + typeclasses.

→ Tried and true; believed to be performant.

① Monad transformers + typeclasses.

- → Tried and true; believed to be performant.
- → Can require complex boilerplate.

① Monad transformers + typeclasses.

- → Tried and true; believed to be performant.
- → Can require complex boilerplate.

Examples: `mtl`, `fused-effects`

① Monad transformers + typeclasses.

- → Tried and true; believed to be performant.
- → Can require complex boilerplate.

Examples: `mtl`, `fused-effects`

② Free-like monads.

① Monad transformers + typeclasses.

- → Tried and true; believed to be performant.
- → Can require complex boilerplate.

Examples: `mtl`, `fused-effects`

② Free-like monads.

- → Highly flexible, can be simpler to use.

① Monad transformers + typeclasses.

- → Tried and true; believed to be performant.
- → Can require complex boilerplate.

Examples: `mtl`, `fused-effects`

② Free-like monads.

- → Highly flexible, can be simpler to use.
- → Performance is a known limitation.

① Monad transformers + typeclasses.

- → Tried and true; believed to be performant.
- → Can require complex boilerplate.

Examples: `mtl`, `fused-effects`

② Free-like monads.

- → Highly flexible, can be simpler to use.
- → Performance is a known limitation.

Examples: `freer-simple`, `polysemy`

① Monad transformers + typeclasses.

- → Tried and true; believed to be performant.
- → Can require complex boilerplate.

Examples: `mtl`, `fused-effects`

② Free-like monads.

- → Highly flexible, can be simpler to use.
- → Performance is a known limitation.

Examples: `freer-simple`, `polysemy`

```
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

```
program = get ⋙= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

▼

```
program :: Eff (State Int) Int
program = Get `Then` \n →
           if n ⩽ 0
             then Return n
             else Put (n - 1) `Then` \_ → program
```

```haskell
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n ->
           if n <= 0
             then Return n
             else Put (n - 1) `Then` \_ -> program
```

```
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

```
program :: Eff (State Int) Int
program = Get `Then` \n ->
           if n <= 0
             then Return n
             else Put (n - 1) `Then` \_ -> program
```

```haskell
data Eff f a where
  Return :: a → Eff a
  Then   :: f a → (a → Eff b) → Eff b
```

```haskell
data Eff f a where
  Return :: a → Eff a
  Then   :: f a → (a → Eff b) → Eff b

data State s a where
  Get    :: State s s
  Put    :: s → State ()
```

```haskell
data Eff f a where
  Return :: a → Eff a
  Then   :: f a → (a → Eff b) → Eff b

data State s a where
  Get    :: State s s
  Put    :: s → State ()

runState :: s → Eff (State s) a → (s, a)
runState s (Return x)         = (s, x)
runState s (Get   `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

PROS

PROS

→ Beautifully simple.

PROS

- → Beautifully simple.
- → Extremely flexible.

PROS

→ Beautifully simple.

→ Extremely flexible.

CONS

PROS

- → Beautifully simple.
- → Extremely flexible.

CONS

- → Reifies the entire program as a tree.

PROS

- → Beautifully simple.
- → Extremely flexible.

CONS

- → Reifies the entire program as a tree.
- → Obscures structure to the optimizer.

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ⩽ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

```haskell
newtype State s a = State { runState :: s → (s, a) }
```

```haskell
newtype State s a = State { runState :: s → (s, a) }

instance Monad (State s) where
  return x = State $ \s → (s, x)
  m ≫= f = State $ \s → case runState m s of
    (s', a) → runState (f a) s'
```

```haskell
newtype State s a = State { runState :: s → (s, a) }

instance Monad (State s) where
  return x = State $ \s → (s, x)
  m >>= f = State $ \s → case runState m s of
    (s', a) → runState (f a) s'

get :: State s s
get = State $ \s → (s, s)

put :: s → State s ()
put s = State $ \_ → (s, ())
```

```haskell
program :: State Int Int
program = get >>= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: State Int Int
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: State Int Int
program = State $ \s1 → runState get s1 of
  (s2, n) → if n ≤ 0
               then runState (return n) s2
               else case runState (put (n - 1)) s2 of
                 (s3, _) → runState program s3
```

```haskell
program :: State Int Int
program = State $ \s1 → runState get s1 of
  (s2, n) → if n ≤ 0
                then runState (return n) s2
                else case runState (put (n - 1)) s2 of
                  (s3, _) → runState program s3
```

```haskell
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ⩽ 0
              then return n s2
              else case put (n - 1) s2 of
                     (s3, _) → program s3
```

```haskell
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ≤ 0
              then return n s2
              else case put (n - 1) s2 of
                     (s3, _) → program s3
```

```
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ≤ 0
              then (s2, n)
              else case put (n - 1) s2 of
                     (s3, _) → program s3
```

```haskell
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ⩽ 0
              then (s2, n)
              else case put (n - 1) s2 of
                     (s3, _) → program s3
```

```haskell
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ≤ 0
              then (s2, n)
              else case put (n - 1) s2 of
                     (s3, _) → program s3
```

```haskell
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ⩽ 0
              then (s2, n)
              else case (n - 1, ()) of
                     (s3, _) → program s3
```

```
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ⩽ 0
              then (s2, n)
              else case (n - 1, ()) of
                     (s3, _) → program s3
```

```
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ⩽ 0
                then (s2, n)
                else case (n - 1, ()) of
                       (s3, _) → program s3
```

```
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ≤ 0
              then (s2, n)
              else program (n - 1)
```

```haskell
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ≤ 0
               then (s2, n)
               else program (n - 1)
```

```
program :: Int → (Int, Int)
program s1 = case get s1 of
  (s2, n) → if n ≤ 0
               then (s2, n)
               else program (n - 1)
```

```haskell
program :: Int → (Int, Int)
program s1 = case (s1, s1) of
  (s2, n) → if n ≤ 0
              then (s2, n)
              else program (n - 1)
```

```haskell
program :: Int → (Int, Int)
program s1 = case (s1, s1) of
  (s2, n) → if n ⩽ 0
              then (s2, n)
              else program (n - 1)
```

```
program :: Int → (Int, Int)
program s1 = case (s1, s1) of
  (s2, n) → if n ≤ 0
              then (s2, n)
              else program (n - 1)
```

```haskell
program :: Int → (Int, Int)
program s1 = if s1 ⩽ 0
               then (s1, s1)
               else program (s1 - 1)
```

```haskell
program :: Int → (Int, Int)
program s1 = if s1 ⩽ 0
                then (s1, s1)
                else program (s1 - 1)
```

```haskell
program :: Int → (Int, Int)
program s1 = if s1 ⩽ 0
               then (s1, s1)
               else program (s1 - 1)

program :: Int → (Int, Int)
program n = if n ⩽ 0
              then (n, n)
              else program (n - 1)
```

```
program :: Int → (Int, Int)
program s1 = if s1 ⩽ 0
               then (s1, s1)
               else program (s1 - 1)
```

```
program :: Int → (Int, Int)
program n = if n ⩽ 0
              then (n, n)
              else program (n - 1)
```

This is great! But it’s not an effect system.

```
program :: State Int Int
program = get >>= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

This is great! But it's not an effect system.

```
program :: State Int Int
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

coupled to implementation

Can we get the flexibility of free monads
with the performance of monad transformers?

Similar to the list fusion problem.

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

producer

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

consumer producer

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

```
foldr (+) 0 (1:2:3:4:5:[])
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

▼

```
foldr (+) 0 (1:2:3:4:5:[])
```

```
1 : 2 : 3 : 4 : 5 : []
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

```
foldr (+) 0 (1:2:3:4:5:[])
```

```
1 : 2 : 3 : 4 : 5 : []
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

```
foldr (+) 0 (1:2:3:4:5:[])
```

```
1 + 2 + 3 + 4 + 5 + []
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

$\blacktriangledown$

```
foldr (+) 0 (1:2:3:4:5:[])
```

```
1 + 2 + 3 + 4 + 5 + []
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

$$\Downarrow$$

```
foldr (+) 0 (1:2:3:4:5:[])
```

```
1 + 2 + 3 + 4 + 5 + []
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

```
foldr (+) 0 (1:2:3:4:5:[])
```

```
1 + 2 + 3 + 4 + 5 + 0
```

Similar to the list fusion problem.

```
foldr (+) 0 [1..5]
```

▼

```
foldr (+) 0 (1:2:3:4:5:[])
```

```
1 + 2 + 3 + 4 + 5 + 0
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ⩽ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

```haskell
runState :: s → Eff (State s) a → (s, a)
runState s (Return x)           = (s, x)
runState s (Get      `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ≤ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

```haskell
runState :: s → Eff (State s) a → (s, a)
runState s (Return x)          = (s, x)
runState s (Get     `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ⩽ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

producer

```haskell
runState :: s → Eff (State s) a → (s, a)
runState s (Return x)          = (s, x)
runState s (Get    `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

```
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ≤ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

producer

```
runState :: s → Eff (State s) a → (s, a)
runState s (Return x)         = (s, x)
runState s (Get    `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ≤ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

producer

consumer

```haskell
runState :: s → Eff (State s) a → (s, a)
runState s (Return x)       = (s, x)
runState s (Get   `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
          if n ⩽ 0
            then Return n
            else Put (n - 1) `Then` \_ → program
```

producer consumer

```haskell
runState :: s → Eff (State s) a → (s, a)
runState s (Return x)        = (s, x)
runState s (Get   `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ⩽ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

producer

consumer

```haskell
runState :: s → Eff (State s) a → (s, a)
runState s (Return x)       = (s, x)
runState s (Get   `Then` k) = runState s (k s)
runState _ (Put s `Then` k) = runState s (k ())
```

Can we get GHC to do this?

Can we get GHC to do this?

Might `mtl` do better?

```haskell
program :: MonadState Int m => m Int
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

How is this compiled?

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

How is this compiled?

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

How is this compiled?

How are typeclasses compiled?

Non-Solution 1: Type Dispatch

Non-Solution 1: Type Dispatch

```
show x = case typeOf x of
  Bool   → show_Bool x
  Char   → show_Char x
  String → show_String x
  ...
```

Non-Solution 1: Type Dispatch

```
show x = case typeOf x of
  Bool   → show_Bool x
  Char   → show_Char x
  String → show_String x
  …
```

Immediate problem: requires whole-program compilation.

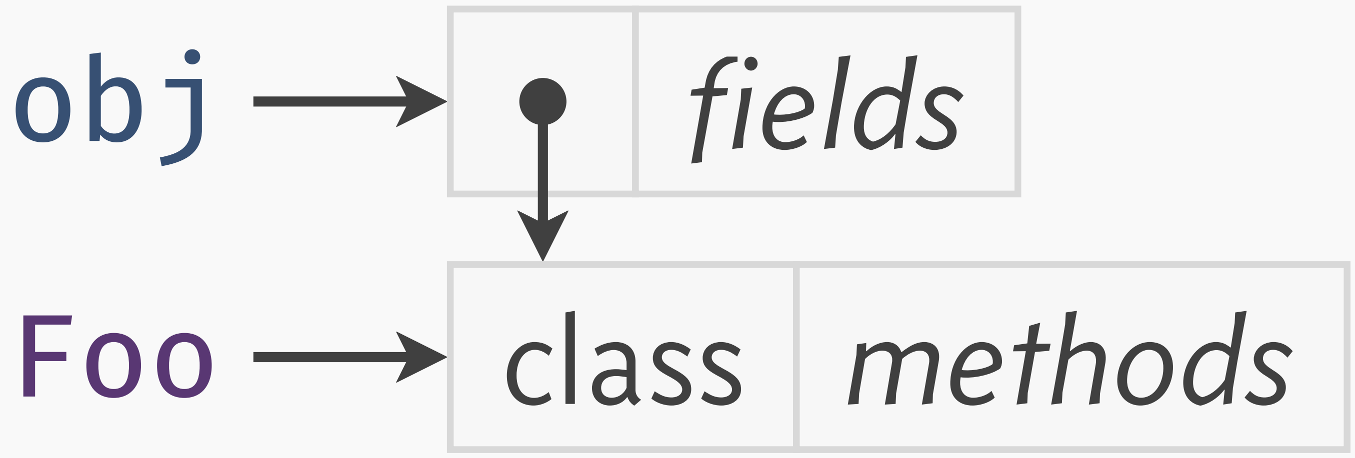
Deeper problem: full type erasure.

Deeper problem: full type erasure.

Java: `if (obj instanceof Foo) { … }`

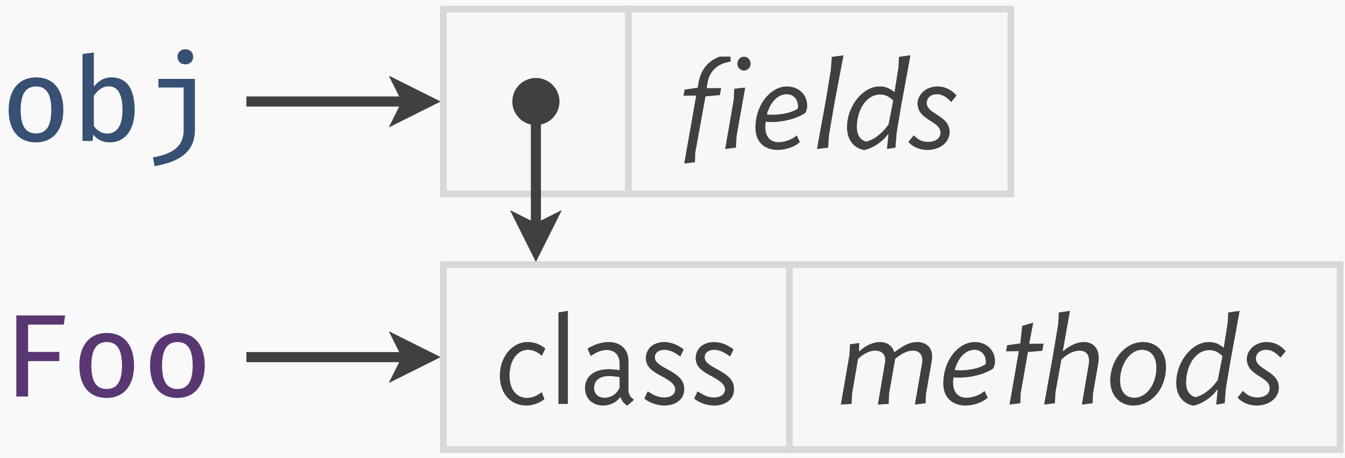
Deeper problem: full type erasure.

Java: `if (obj instanceof Foo) { ... }`



Deeper problem: full type erasure.

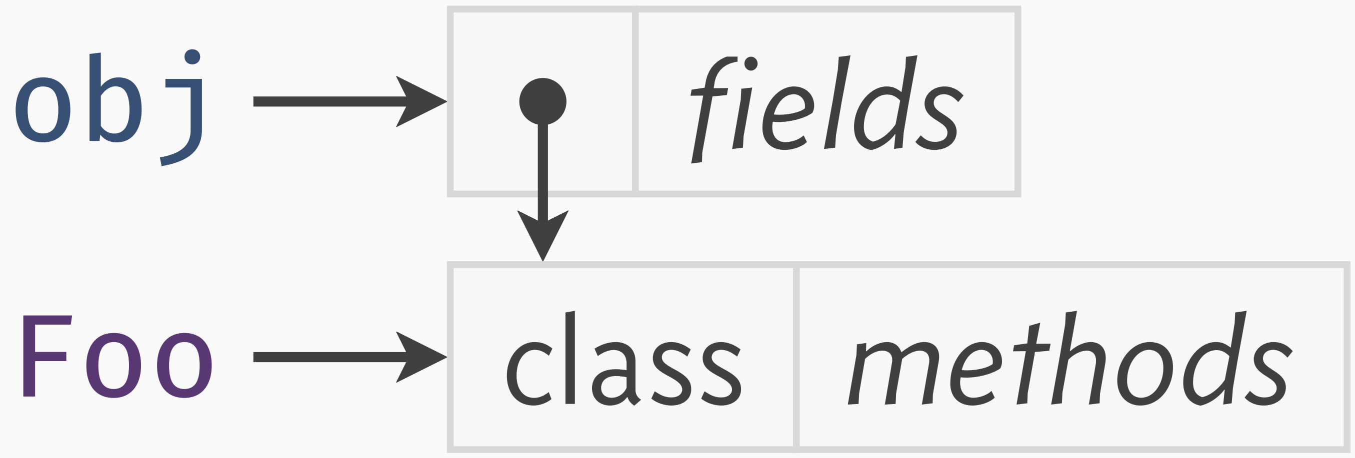
Java: `if (obj instanceof Foo) { ... }`



Haskell: `data Foo = MkFoo Int String`

Deeper problem: full type erasure.

Java: `if (obj instanceof Foo) { … }`

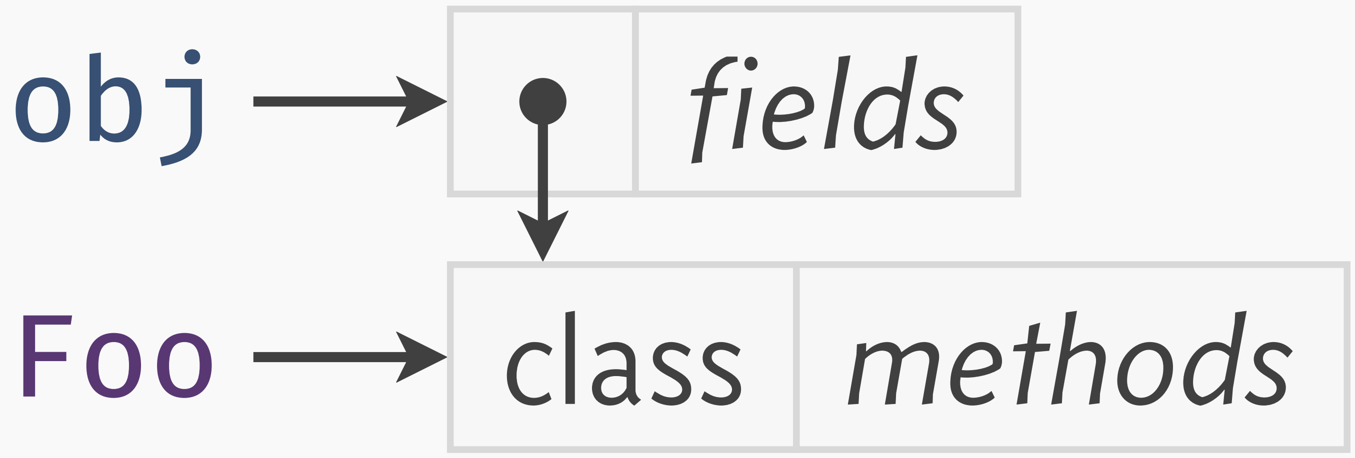


Haskell: `data Foo = MkFoo Int String`

```
let val = MkFoo 42 "hello"
```

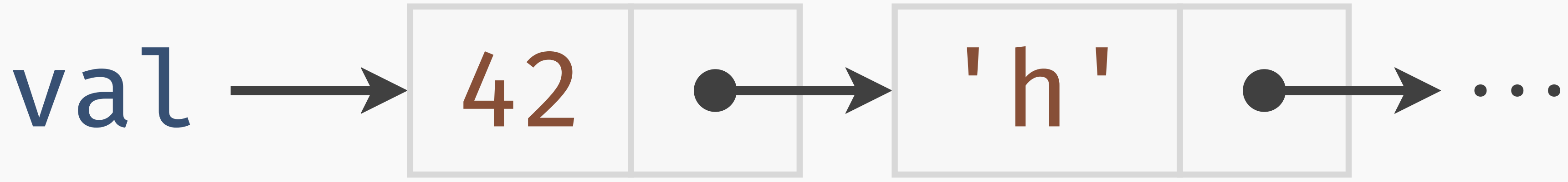
Deeper problem: full type erasure.

Java: `if (obj instanceof Foo) { … }`



Haskell: `data Foo = MkFoo Int String`

```
let val = MkFoo 42 "hello"
```



Non-Solution 2: Monomorphization

Non-Solution 2: Monomorphization

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

Non-Solution 2: Monomorphization

```
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

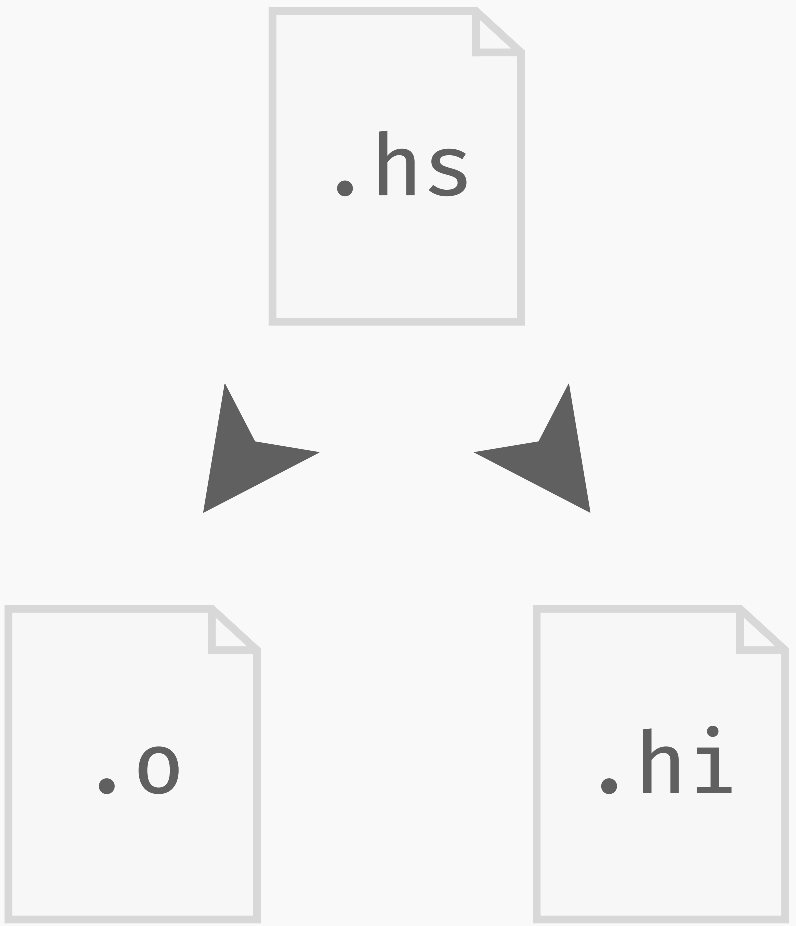
1. Generate *no code* for `exclaim`.

Non-Solution 2: Monomorphization

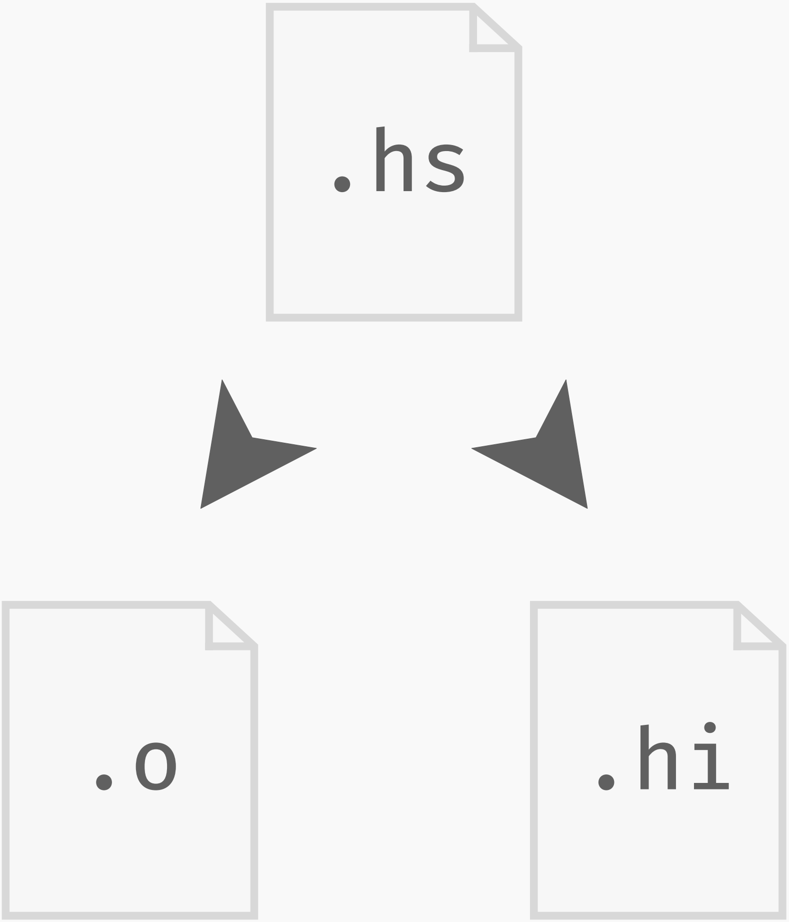
```
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

1. Generate *no code* for `exclaim`.
2. Record `exclaim`'s definition in the interface file.


.hs

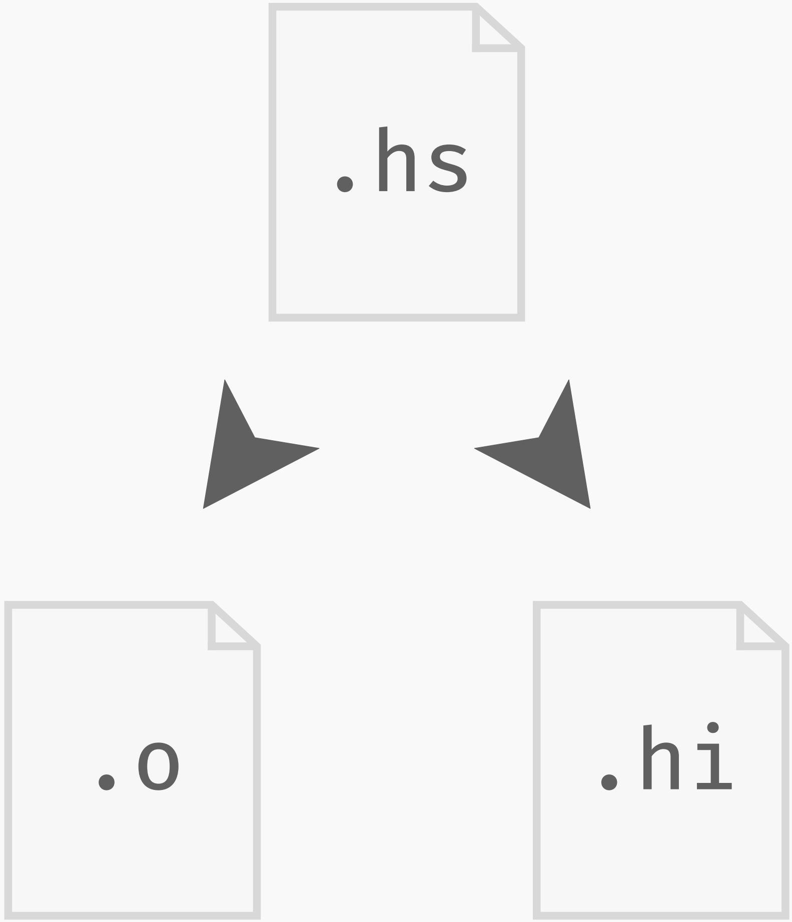

.hs
.o
.hi

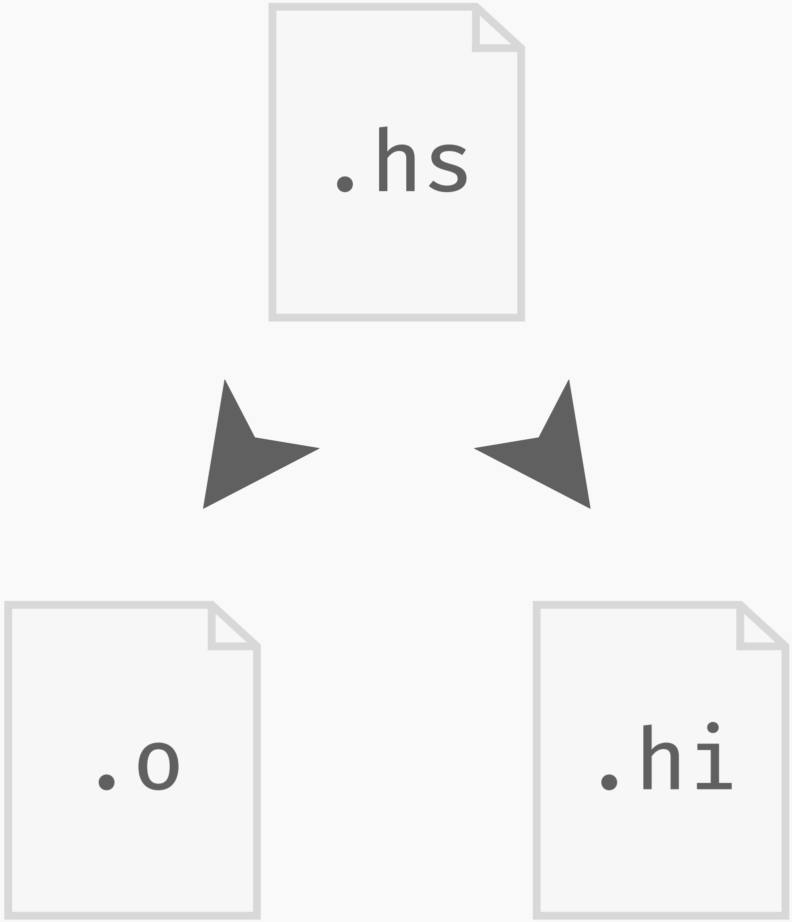
`.hi` — “Haskell interface”



`.hi` — "Haskell interface"

→ type/class/instance declarations

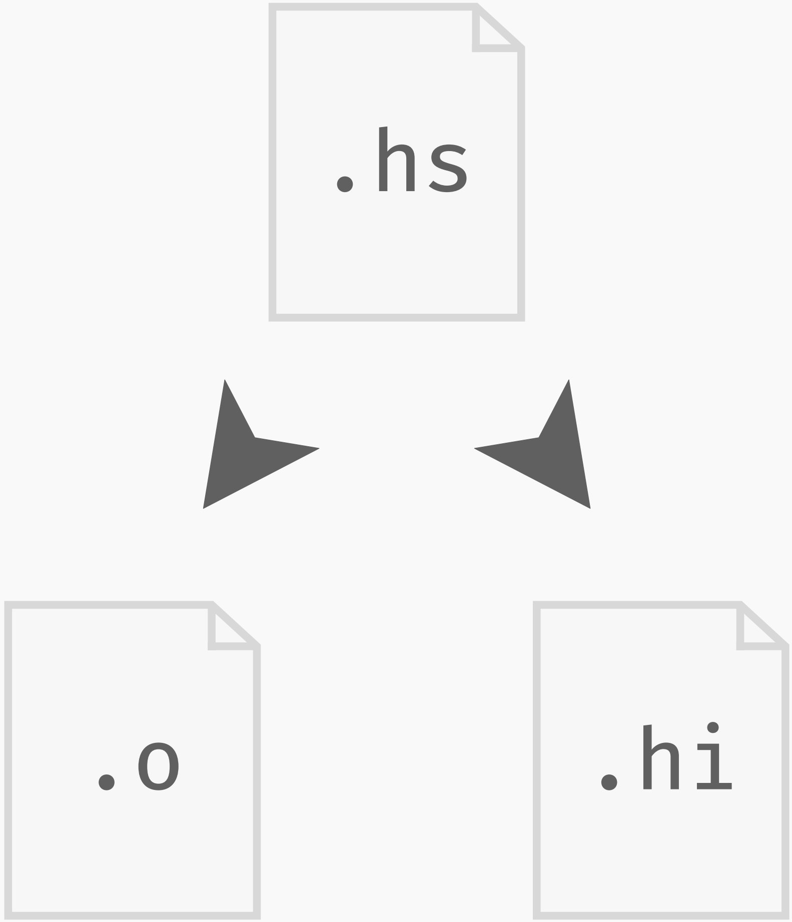




`.hi` — “Haskell interface”

→ type/class/instance declarations

→ types of exported bindings

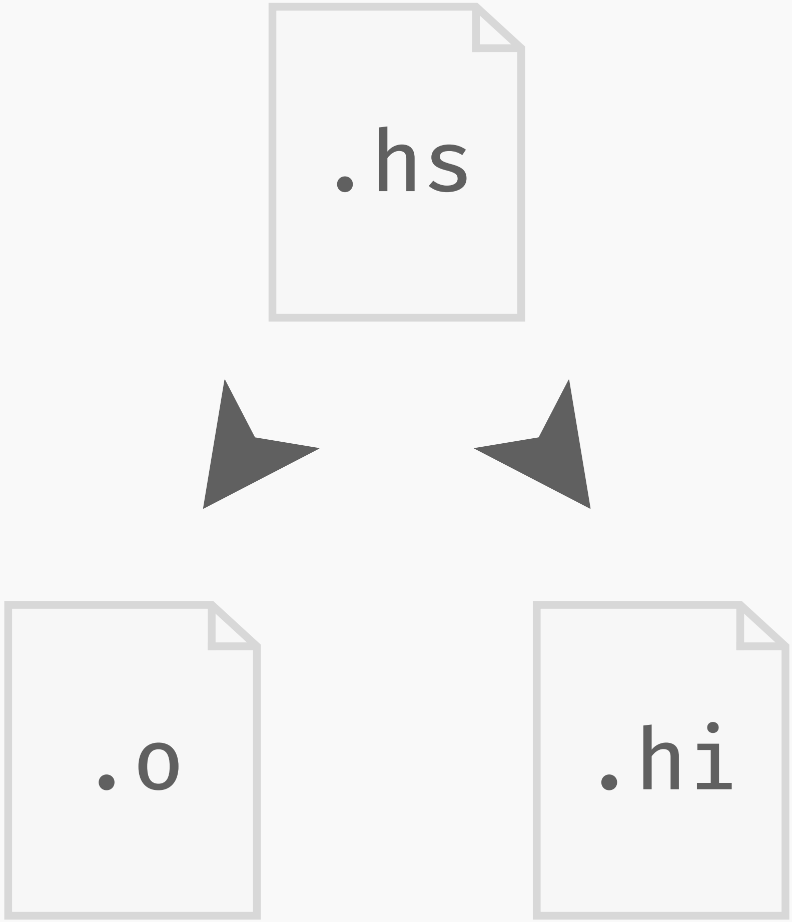


`.hi` — “Haskell interface”

→ type/class/instance declarations

→ types of exported bindings

→ source code of small bindings



`.hi` — "Haskell interface"

- → type/class/instance declarations
- → types of exported bindings
- → source code of small bindings
- → for monomorphization: source code of *all* overloaded bindings

```
exclaim True
```

```
exclaim True
```

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

```
exclaim True
```

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

```haskell
exclaim_Bool :: Bool → String
exclaim_Bool x = show_Bool x ++ "!"
```

```haskell
exclaim True
```

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

```haskell
exclaim_Bool :: Bool → String
exclaim_Bool x = show_Bool x ++ "!"
```

```haskell
exclaim True ➤ exclaim_Bool True
```

exclaim True

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

```haskell
exclaim_Bool :: Bool → String
exclaim_Bool x = show_Bool x ++ "!"
```

exclaim True ➤ exclaim_Bool True

Overloading has no runtime cost!

…but it can create a lot of bloat.

…but it can create a lot of bloat.

```
exclaim True    exclaim 42    exclaim "hello"
```

…but it can create a lot of bloat.

```
exclaim True    exclaim 42    exclaim "hello"
```

▼

```
exclaim_Bool :: Bool → String
exclaim_Bool x = show_Bool x ++ "!"

exclaim_Int :: Int → String
exclaim_Int x = show_Int x ++ "!"

exclaim_String :: String → String
exclaim_String x = show_String x ++ "!"
```

```
reallyBig :: (Foo a, Bar b, Baz c) ⇒ …
reallyBig = < really big RHS >
```

```
reallyBig :: (Foo a, Bar b, Baz c) ⇒ …
reallyBig = < really big RHS >
```

⮟

```
reallyBig @Bool   @Bool   @Bool     reallyBig @Bool   @Bool   @Int     reallyBig @Bool   @Bool   @String
reallyBig @Bool   @Int    @Bool     reallyBig @Bool   @Int    @Int     reallyBig @Bool   @Int    @String
reallyBig @Bool   @String @Bool     reallyBig @Bool   @String @Int     reallyBig @Bool   @String @String
reallyBig @Int    @Bool   @Bool     reallyBig @Int    @Bool   @Int     reallyBig @Int    @Bool   @String
reallyBig @Int    @Int    @Bool     reallyBig @Int    @Int    @Int     reallyBig @Int    @Int    @String
reallyBig @Int    @String @Bool     reallyBig @Int    @String @Int     reallyBig @Int    @String @String
reallyBig @String @Bool   @Bool     reallyBig @String @Bool   @Int     reallyBig @String @Bool   @String
reallyBig @String @Int    @Bool     reallyBig @String @Int    @Int     reallyBig @String @Int    @String
reallyBig @String @String @Bool     reallyBig @String @String @Int     reallyBig @String @String @String
```

Monomorphization

Monomorphization

→ Can be good for runtime performance.

Monomorphization

→ Can be good for runtime performance.

→ Can be very bad for code size & compile times.

Monomorphization

- → Can be good for runtime performance.
- → Can be very bad for code size & compile times.
- → C++/Rust programmers have to worry about this!

Monomorphization

- → Can be good for runtime performance.
- → Can be very bad for code size & compile times.
- → C++/Rust programmers have to worry about this!
- → Haskell programmers generally do not.

```haskell
exclaim :: Show a => a -> String
exclaim x = show x ++ "!"
```

```
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

▼

```
exclaim :: (a → String) → a → String
exclaim show_a x = show_a x ++ "!"
```

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

▼

```haskell
exclaim :: (a → String) → a → String
exclaim show_a x = show_a x ++ "!"
```

```haskell
exclaim True
```

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

▼

```haskell
exclaim :: (a → String) → a → String
exclaim show_a x = show_a x ++ "!"
```

```haskell
exclaim True ➤ exclaim show_Bool True
```

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

▼

```haskell
exclaim :: (a → String) → a → String
exclaim show_a x = show_a x ++ "!"
```

```haskell
exclaim True ➤ exclaim show_Bool True

exclaim 42
```

```haskell
exclaim :: Show a ⇒ a → String
exclaim x = show x ++ "!"
```

▼

```haskell
exclaim :: (a → String) → a → String
exclaim show_a x = show_a x ++ "!"
```

```haskell
exclaim True ➤ exclaim show_Bool True

exclaim 42 ➤ exclaim show_Int 42
```

```haskell
class Show a where
  show      :: a → String
  showsPrec :: Int → a → ShowS
  showList  :: [a] → ShowS
```

```haskell
class Show a where
  show      :: a → String
  showsPrec :: Int → a → ShowS
  showList  :: [a] → ShowS

data Show a = ShowDict
  { show      :: a → String
  , showsPrec :: Int → a → ShowS
  , showList  :: [a] → ShowS }
```

```haskell
class Show a where
  show      :: a → String
  showsPrec :: Int → a → ShowS
  showList  :: [a] → ShowS

data Show a = ShowDict
  { show      :: a → String
  , showsPrec :: Int → a → ShowS
  , showList  :: [a] → ShowS }

exclaim :: Show a → a → String
exclaim dict x = show dict x ++ "!"
```

This is *dictionary passing*.

This is *dictionary passing*.

→ Elegantly simple.

This is *dictionary passing*.

- → Elegantly simple.
- → Cheap to compile.

This is *dictionary passing*.

→ Elegantly simple.

→ Cheap to compile.

→ Does it have a runtime cost?

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get >>= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (≫=) monadDict
       (get stateDict)
       (\n → if n ⩽ 0
          then return monadDict n
          else (>>) monadDict
                    (put stateDict (n - 1))
                    (program stateDict)
```

```
program :: MonadState Int m ⇒ m Int
program = get >>= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

```
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (>>=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get >>= \n →
              if n ⩽ 0
                then return n
                else put (n - 1) >> program
```

▼

```haskell
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (>>=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                      (put stateDict (n - 1))
                      (program stateDict)
```

```
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

▼

```
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (≫=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (≫=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                      (put stateDict (n - 1))
                      (program stateDict)
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

▼

```haskell
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (≫=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

▼

```haskell
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (≫=) monadDict
        (get stateDict)
        (\n → if n ≤ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

Is this performant?

```
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (>>=) monadDict
        (get stateDict)
        (\n → if n ≤ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

Is this performant?

Nope: we can’t inline anything.

```
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (>>=) monadDict
        (get stateDict)
        (\n → if n ≤ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

Is this performant?

Nope: we can't inline anything.

```
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (>>=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

Is this performant?

Nope: we can't inline anything.

```
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (>>=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

Most calls are *known* calls.

Most calls are *known* calls.

```
fst ("hello", "world")
```

Most calls are *known* calls.

```
fst ("hello", "world")
```

```
fst :: (a, b) → a
fst (x, _) = x
```

Most calls are *known* calls.

```
fst ("hello", "world")
```

```
(\(x, _) → x) ("hello", "world")
```

Most calls are *known* calls.

```
fst ("hello", "world")
```

▼

```
(\(x, _) → x) ("hello", "world")
```

▼

```
"hello"
```

Higher-order functions make *unknown* calls.

Higher-order functions make *unknown* calls.

```
foldr :: (a → b → b) → b → [a] → b
foldr _ v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Higher-order functions make *unknown* calls.

```
foldr :: (a → b → b) → b → [a] → b
foldr _ v []       = v
foldr f v (x:xs) = f x (foldr f v xs)
```

Higher-order functions make *unknown* calls.

```
foldr :: (a → b → b) → b → [a] → b
foldr _ v []      = v
foldr f v (x:xs)  = f x (foldr f v xs)
```

???

Higher-order functions make *unknown* calls.

```
foldr :: (a → b → b) → b → [a] → b
foldr _ v []         = v
foldr f v (x:xs) = f x (foldr f v xs)
```

???

Higher-order functions make *unknown* calls.

```
foldr :: (a → b → b) → b → [a] → b
foldr _ v []     = v
foldr f v (x:xs) = f x (foldr f v xs)
```

???

Unknown calls are *hard stops* for the optimizer.

KNOWN CALL

UNKNOWN CALL

KNOWN CALL

→ Compiled to direct jump.

UNKNOWN CALL

KNOWN CALL

- → Compiled to direct jump.

UNKNOWN CALL

- → Compiled to indirect jump.

KNOWN CALL

- → Compiled to direct jump.
- → Exposes strictness info.

UNKNOWN CALL

- → Compiled to indirect jump.

KNOWN CALL

→ Compiled to direct jump.

→ Exposes strictness info.

UNKNOWN CALL

→ Compiled to indirect jump.

→ Assumed lazy in all args.

KNOWN CALL

- → Compiled to direct jump.
- → Exposes strictness info.
- → Args can be unboxed.

UNKNOWN CALL

- → Compiled to indirect jump.
- → Assumed lazy in all args.

KNOWN CALL

- → Compiled to direct jump.
- → Exposes strictness info.
- → Args can be unboxed.

UNKNOWN CALL

- → Compiled to indirect jump.
- → Assumed lazy in all args.
- → Args must have declared types.

KNOWN CALL

- → Compiled to direct jump.
- → Exposes strictness info.
- → Args can be unboxed.
- → Can have rewrite RULES.

UNKNOWN CALL

- → Compiled to indirect jump.
- → Assumed lazy in all args.
- → Args must have declared types.

KNOWN CALL

→ Compiled to direct jump.

→ Exposes strictness info.

→ Args can be unboxed.

→ Can have rewrite RULES.

UNKNOWN CALL

→ Compiled to indirect jump.

→ Assumed lazy in all args.

→ Args must have declared types.

→ Opaque to RULES.

KNOWN CALL

- → Compiled to direct jump.
- → Exposes strictness info.
- → Args can be unboxed.
- → Can have rewrite RULES.
- → Inlined if small.

UNKNOWN CALL

- → Compiled to indirect jump.
- → Assumed lazy in all args.
- → Args must have declared types.
- → Opaque to RULES.

KNOWN CALL	UNKNOWN CALL
→ Compiled to direct jump.	→ Compiled to indirect jump.
→ Exposes strictness info.	→ Assumed lazy in all args.
→ Args can be unboxed.	→ Args must have declared types.
→ Can have rewrite RULES.	→ Opaque to RULES.
→ Inlined if small.	→ Never inlined.

KNOWN CALL

- → Compiled to direct jump.
- → Exposes strictness info.
- → Args can be unboxed.
- → Can have rewrite RULES.
- → Inlined if small.

UNKNOWN CALL

- → Compiled to indirect jump.
- → Assumed lazy in all args.
- → Args must have declared types.
- → Opaque to RULES.
- → Never inlined.

This is the cost of AOT compilation!

Typeclass overloading creates unknown calls.

Typeclass overloading creates unknown calls.

Overloading is **not** free!

Unknown calls are not a death sentence.

Unknown calls are not a death sentence.

```
sumIndicies :: Eq a ⇒ a → [a] → [Int]
sumIndicies v xs = zip [1..] xs
                 & filter ((== v) . snd)
                 & map fst
                 & sum
```

Unknown calls are not a death sentence.

```
sumIndicies :: Eq a ⇒ a → [a] → [Int]
sumIndicies v xs = zip [1..] xs
                 & filter ((== v) . snd)
                 & map fst
                 & sum
```

Unknown calls are not a death sentence.

```
sumIndicies :: Eq a ⇒ a → [a] → [Int]
sumIndicies v xs = zip [1..] xs
                 & filter ((== v) . snd)
                 & map fst
                 & sum
```

Unknown calls are not a death sentence.

```
sumIndicies :: Eq a ⇒ a → [a] → [Int]
sumIndicies v xs = zip [1..] xs
                 & filter ((== v) . snd)
                 & map fst
                 & sum
```

List fusion can still happen!

Unknown calls to $\gg=$ are a problem.

Unknown calls to ⤜ are a problem.

```
foo :: k → Map k Int → Either String Int
foo key vals = do
  nums ← case Map.lookup key vals of
           Nothing  → Left "not found"
           Just val → Right [1..val]
  Right $ sum nums
```

Unknown calls to ⤜ are a problem.

```haskell
foo :: k → Map k Int → Either String Int
foo key vals = do
  nums ← case Map.lookup key vals of
              Nothing  → Left "not found"
              Just val → Right [1..val]
  Right $ sum nums
```

Unknown calls to `>>=` are a problem.

```haskell
foo :: k -> Map k Int -> Either String Int
foo key vals = do
  nums <- case Map.lookup key vals of
            Nothing  -> Left "not found"
            Just val -> Right [1..val]
  Right $ sum nums
```

Unknown calls to >>= are a problem.

```
foo :: k → Map k Int → Either String Int
foo key vals = do
  nums ← case Map.lookup key vals of
           Nothing  → Left "not found"
           Just val → Right [1..val]
  Right $ sum nums
```

Unknown calls to ≫= are a problem.

```
foo :: k → Map k Int → Either String Int
foo key vals = do
  nums ← case Map.lookup key vals of
             Nothing  → Left "not found"
             Just val → Right [1..val]
  Right $ sum nums
```

Unknown calls to >>= are a problem.

```
foo :: MonadError String m ⇒ k → Map k Int → m Int
foo key vals = do
  nums ← case Map.lookup key vals of
           Nothing  → throwError "not found"
           Just val → return [1..val]
  return $ sum nums
```

Unknown calls to ≫= are a problem.

```
foo :: MonadError String m ⇒ k → Map k Int → m Int
foo key vals = do
  nums ← case Map.lookup key vals of
           Nothing  → throwError "not found"
           Just val → return [1..val]
  return $ sum nums
```

Unknown calls to $\gg=$ are a problem.

```
foo :: MonadError String m ⇒ k → Map k Int → m Int
foo key vals = do
  nums ← case Map.lookup key vals of
           Nothing  → throwError "not found"
           Just val → return [1..val]
  return $ sum nums
```

$\gg=$ is *glue*.

Conclusion: not surprising at all that `mtl` has a cost!

Conclusion: not surprising at all that `mtl` has a cost!

But why is it *sometimes* fast?

SPECIALIZATION

GHC monomorphizes in limited circumstances.

GHC monomorphizes in limited circumstances.

1. GHC looks for calls to overloaded functions at known, concrete types.

GHC monomorphizes in limited circumstances.

1. GHC looks for calls to overloaded functions at known, concrete types.
2. The function must satisfy *at least one* of the following:

GHC monomorphizes in limited circumstances.

1. GHC looks for calls to overloaded functions at known, concrete types.
2. The function must satisfy *at least one* of the following:
    - → It is defined in the current module.

GHC monomorphizes in limited circumstances.

1. GHC looks for calls to overloaded functions at known, concrete types.
2. The function must satisfy *at least one* of the following:
   - → It is defined in the current module.
   - → It was declared with an `INLINABLE` pragma.

GHC monomorphizes in limited circumstances.

1. GHC looks for calls to overloaded functions at known, concrete types.
2. The function must satisfy *at least one* of the following:
   - → It is defined in the current module.
   - → It was declared with an `INLINABLE` pragma.
   - → It is a class method *and* its unfolding (source code) is in the interface file.

GHC monomorphizes in limited circumstances.

1. GHC looks for calls to overloaded functions at known, concrete types.
2. The function must satisfy *at least one* of the following:
    - → It is defined in the current module.
    - → It was declared with an `INLINABLE` pragma.
    - → It is a class method *and* its unfolding (source code) is in the interface file.

GHC monomorphizes in limited circumstances.

1. GHC looks for calls to overloaded functions at known, concrete types.
2. The function must satisfy *at least one* of the following:
   - → It is defined in the current module.
   - → It was declared with an `INLINABLE` pragma.
   - → It is a class method *and* its unfolding (source code) is in the interface file.

```haskell
program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

▼

```haskell
program :: State Int Int
program = get ⩾= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = do n ← get
             if n ⩽ 0
               then return n
               else put (n - 1) >> program
```

▼

```haskell
program :: MonadState Int m → m Int
program stateDict@(MonadStateDict monadDict _ _) =
  (>>=) monadDict
        (get stateDict)
        (\n → if n ⩽ 0
           then return monadDict n
           else (>>) monadDict
                     (put stateDict (n - 1))
                     (program stateDict)
```

Why bother explaining all of this?

Why bother explaining all of this?

Because we're not done.

Can we avoid the performance regression?

Can we avoid the performance regression?

→ It is defined in the current module.

→ It was declared with an `INLINABLE` pragma.

→ It is a class method *and* its unfolding (source code) is in the interface file.

Can we avoid the performance regression?

- → It is defined in the current module.
- → It was declared with an `INLINABLE` pragma.
- → It is a class method *and* its unfolding (source code) is in the interface file.

This approach has many problems.

This approach has many problems.

1. It would be annoying.

This approach has many problems.

1. It would be annoying.
2. It requires *whole-program* specialization!

This approach has many problems.

1. It would be annoying.
2. It requires *whole-program* specialization!
   → Probably recompiles everything in `Main.hs`.

This approach has many problems.

1. It would be annoying.
2. It requires *whole-program* specialization!
   - → Probably recompiles everything in `Main.hs`.
   - → Compilation may not terminate.

This approach has many problems.

1. It would be annoying.
2. It requires *whole-program* specialization!
   - → Probably recompiles everything in `Main.hs`.
   - → Compilation may not terminate.
   - → This cost is incurred for *each* specialization.

This approach has many problems.

1. It would be annoying.
2. It requires *whole-program* specialization!
   - → Probably recompiles everything in `Main.hs`.
   - → Compilation may not terminate.
   - → This cost is incurred for *each* specialization.
   - → Can be defeated by existential quantification.

Let’s take a step back.

Let’s take a step back.

Why is specialization necessary here?

Proposition 1: effect systems are *fundamentally* about dynamic dispatch.

Proposition 1: effect systems are *fundamentally* about dynamic dispatch.

```
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

Proposition 1: effect systems are *fundamentally* about dynamic dispatch.

```
program = get >>= \n ->
            if n <= 0
              then return n
              else put (n - 1) >> program
```

Proposition 1: effect systems are *fundamentally* about dynamic dispatch.

```
program = get >>= \n ->
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

interpretation not yet decided!

Proposition 2: effect dispatch can be made perfectly affordable. The real problem is $\ggg=$.

```
program = get >>= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

Proposition 2: effect dispatch can be made perfectly affordable. The real problem is $\gg=$.

```
program = get >>= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

Passing `>>=` via dictionary creates problems!

Passing $\gg=$ via dictionary creates problems!

1. $\gg=$ gets called a *lot*.
2. $\gg=$ is glue; it needs to be inlined to expose further optimizations.
3. Unknown calls to $\gg=$ balloon closure allocation.

Passing $\gg=$ via dictionary creates problems!

1. $\gg=$ gets called a *lot*.
2. $\gg=$ is glue; it needs to be inlined to expose further optimizations.
3. Unknown calls to $\gg=$ balloon closure allocation.

```
f x y = foo x >>= \z -> bar (+ y z)
```

```
f x y = foo x >>= \z -> bar (+ y z)
```

▼

```
f x y = case foo x of
  Left e  -> Left e
  Right z -> bar (+ y z)
```

```
f x y = foo x >>= \z → bar (+ y z)
```

▼

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

```
f x y = foo x >>= \z → bar (+ y z)
```

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

used under a lambda

```
f x y = foo x >>= \z → bar (+ y z)
```

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

used under a lambda

```
f x y = foo x >>= \z → bar (+ y z)
```

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

used under a lambda

```
f x y = foo x >>= \z → bar (+ y z)
```

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

used in a case RHS

used under a lambda

```
f x y = foo x >>= \z → bar (+ y z)
```

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

used in a case RHS

captured in closure

used under a lambda

```
f x y = foo x >>= \z -> bar (+ y z)
```

```
f x y = case foo x of
  Left e  -> Left e
  Right z -> bar (+ y z)
```

used in a case RHS

captured in closure

used under a lambda

```
f x y = foo x >>= \z → bar (+ y z)
```

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

used in a case RHS

captured in closure

used under a lambda

```
f x y = foo x >>= \z → bar (+ y z)
```

stored on stack

```
f x y = case foo x of
  Left e  → Left e
  Right z → bar (+ y z)
```

used in a case RHS

captured in closure

used under a lambda

```haskell
f x y = foo x >>= \z -> bar (+ y z)
```

stored on stack

```haskell
f x y = case foo x of
  Left e  -> Left e
  Right z -> bar (+ y z)
```

used in a case RHS

If $\gg=$ is an unknown call, the caller must allocate a closure for the continuation.

If `>>=` is an unknown call, the caller must allocate a closure for the continuation.

When `>>=` is passed via dictionary, the program is *reified as a tree of lambdas!*

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ⩽ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ⟫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ≤ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ≤ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ⩽ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

```haskell
program :: MonadState Int m ⇒ m Int
program = get ≫= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

```haskell
program :: Eff (State Int) Int
program = Get `Then` \n →
            if n ⩽ 0
              then Return n
              else Put (n - 1) `Then` \_ → program
```

Without specialization, these *aren’t that different!*

```haskell
program :: MonadState Int m ⇒ m Int
program = get ⋙= \n →
            if n ⩽ 0
              then return n
              else put (n - 1) >> program
```

How do we escape?

ESCAPE PLAN

ESCAPE PLAN

① We need a monad with an inlinable $\gg\!=$.

ESCAPE PLAN

① We need a monad with an inlinable `>>=`.

② `>>=` itself must not allocate.

ESCAPE PLAN

① We need a monad with an inlinable `>>=`.

② `>>=` itself must not allocate.

③ It must be able to handle all algebraic effects.

ESCAPE PLAN

① We need a monad with an inlinable `>>=`.

② `>>=` itself must not allocate.

③ It must be able to handle all algebraic effects.

④ Effect dispatch can be dynamic (but must be fast).

ESCAPE PLAN

① We need a monad with an inlinable `>>=`.

② `>>=` itself must not allocate.

③ It must be able to handle all algebraic effects.

④ Effect dispatch can be dynamic (but must be fast).

This is not a small order.

ESCAPE PLAN

① We need a monad with an inlinable $\gg=$.

② $\gg=$ itself must not allocate.

③ It must be able to handle all algebraic effects.

④ Effect dispatch can be dynamic (but must be fast).

This is not a small order.

Requirements ② and ③ are especially hard.

Monad transformers are out of the question.

Monad transformers are out of the question.

(Effect behavior lives in $\succcurlyeq$.)

Monad transformers are out of the question.

(Effect behavior lives in $\succ\!\!=$.)

Free-like monads are also disqualified.

Monad transformers are out of the question.

(Effect behavior lives in `>>=`.)

Free-like monads are also disqualified.

(`>>=` must allocate to construct the tree.)

Monad transformers are out of the question.

(Effect behavior lives in $\gg=$.)

Free-like monads are also disqualified.

($\gg=$ must allocate to construct the tree.)

What’s left?

DELIMITED CONTINUATIONS

DELIMITED CONTINUATIONS

There is a deep, well-known connection between delimited continuations and algebraic effects.

DELIMITED CONTINUATIONS

There is a deep, well-known connection between delimited continuations and algebraic effects.

(Well outside the scope of this talk!)

DELIMITED CONTINUATIONS

There is a deep, well-known connection between delimited continuations and algebraic effects.

(Well outside the scope of this talk!)

Big idea: equip a monad with a pair of super powerful control operators, `prompt` and `control`.

DELIMITED CONTINUATIONS

There is a deep, well-known connection between delimited continuations and algebraic effects.

(Well outside the scope of this talk!)

Big idea: equip a monad with a pair of super powerful control operators, `prompt` and `control`.

All effect handlers can be defined in terms of these operators.

`>>=` does not need to know anything about effect behavior.

New problem: the only way to implement delimited continuations in Haskell is CPS.

New problem: the only way to implement delimited continuations in Haskell is CPS.

In CPS, continuation is passed via closure:

```
m >>= f = \k -> m (\x -> f x k)
```

New problem: the only way to implement delimited continuations in Haskell is CPS.

In CPS, continuation is passed via closure:

```
m >>= f = \k -> m (\x -> f x k)
```

All calls to non-inlined monadic functions must allocate.
(This isn’t much better than free monads.)

New problem: the only way to implement delimited continuations in Haskell is CPS.

In CPS, continuation is passed via closure:

```
m >>= f = \k -> m (\x -> f x k)
```

All calls to non-inlined monadic functions must allocate. (This isn't much better than free monads.)

Cost is okay for continuation-happy code, but effects that don't need them (very common!) still must pay for them.

But this is frustrating!

But this is frustrating!

There are well-known, efficient implementation techniques for delimited continuations!

I give up.

I give up. Let’s just patch GHC.

ghc-proposals / **ghc-proposals**

Watch 143

Code | Issues 15 | Pull requests 61 | Actions | Security 0 | Insights

Delimited continuation primops #313

Open **lexi-lambda** wants to merge 4 commits into `ghc-proposals:master` from `lexi-lambda:delimited-continuation-`

Conversation 39 | Commits 4 | Checks 0 | Files changed 2

lexi-lambda commented on Feb 22 • edited

This is a proposal for adding primops for capturing slices of the RTS stack to improve the performance of algebraic effect systems, which require delimited continuations.

Rendered

👍 75 🎉 17 ❤️ 10 🚀 10

EFF

hasura / eff

Unwatch 40 | Star 283 | Fork 7

Code | Issues 3 | Pull requests 0 | Actions | Projects 0 | Wiki | Security 0 | Insights | Settings

🚧 a work in progress effect system for Haskell 🚧

Edit

Manage topics

26 commits | 5 branches | 0 packages | 0 releases | 1 environment | 3 contributors | ISC

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

lexi-lambda Rename `shift` to `control` ... | Latest commit 17cc75d on Mar 3

eff	Rename `shift` to `control`	3 months ago
.gitignore	initial commit	8 months ago
.travis.yml	Update to support continuation primop changes	4 months ago
LICENSE	initial commit	8 months ago
README.md	Rename `swizzle` to `lift`, remove old version of `lift`	3 months ago
cabal.project	Completely rewrite implementation to use delimited continuations	4 months ago
cabal.project.travis	Completely rewrite implementation to use delimited continuations	4 months ago

README.md

`eff` — screaming fast extensible effects for less

build passing | docs 0.0.0.0

🚧 This library is currently under construction. 🚧

`eff` is a work-in-progress implementation of an *extensible effect system* for Haskell, a general-purpose solution for tracking effects at the type level and handling them in flexible ways. Compared to other effect systems currently available, `eff` differentiates itself in the following respects:

EFF: KEY FEATURES

EFF: KEY FEATURES

→ API is similar to free-like approaches (actually simpler!).

EFF: KEY FEATURES

- API is similar to free-like approaches (actually simpler!).
- Provides a single monad, named `Eff`.

EFF: KEY FEATURES

- API is similar to free-like approaches (actually simpler!).
- Provides a single monad, named `Eff`.
- Basically `ReaderT` over `ST` under the hood.

EFF: KEY FEATURES

- API is similar to free-like approaches (actually simpler!).
- Provides a single monad, named `Eff`.
- Basically `ReaderT` over `ST` under the hood.
  - Reader context holds current handlers.

EFF: KEY FEATURES

- API is similar to free-like approaches (actually simpler!).
- Provides a single monad, named `Eff`.
- Basically `ReaderT` over `ST` under the hood.
  - Reader context holds current handlers.
  - Wraps the unsafe primops in a safe API.

EFF: KEY FEATURES

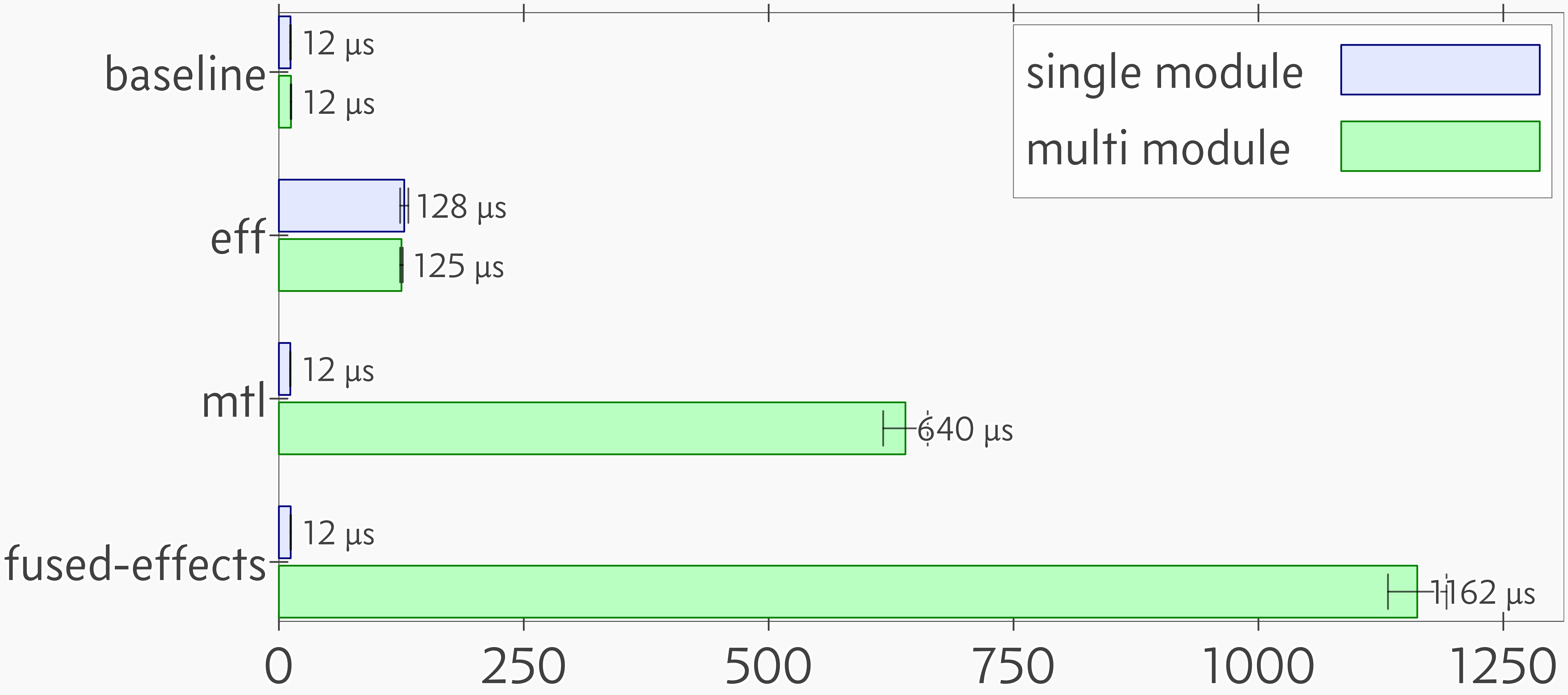
- API is similar to free-like approaches (actually simpler!).
- Provides a single monad, named `Eff`.
- Basically `ReaderT` over `ST` under the hood.
  - Reader context holds current handlers.
  - Wraps the unsafe primops in a safe API.
- Effect dispatch takes constant time (not amortized).

EFF: KEY FEATURES

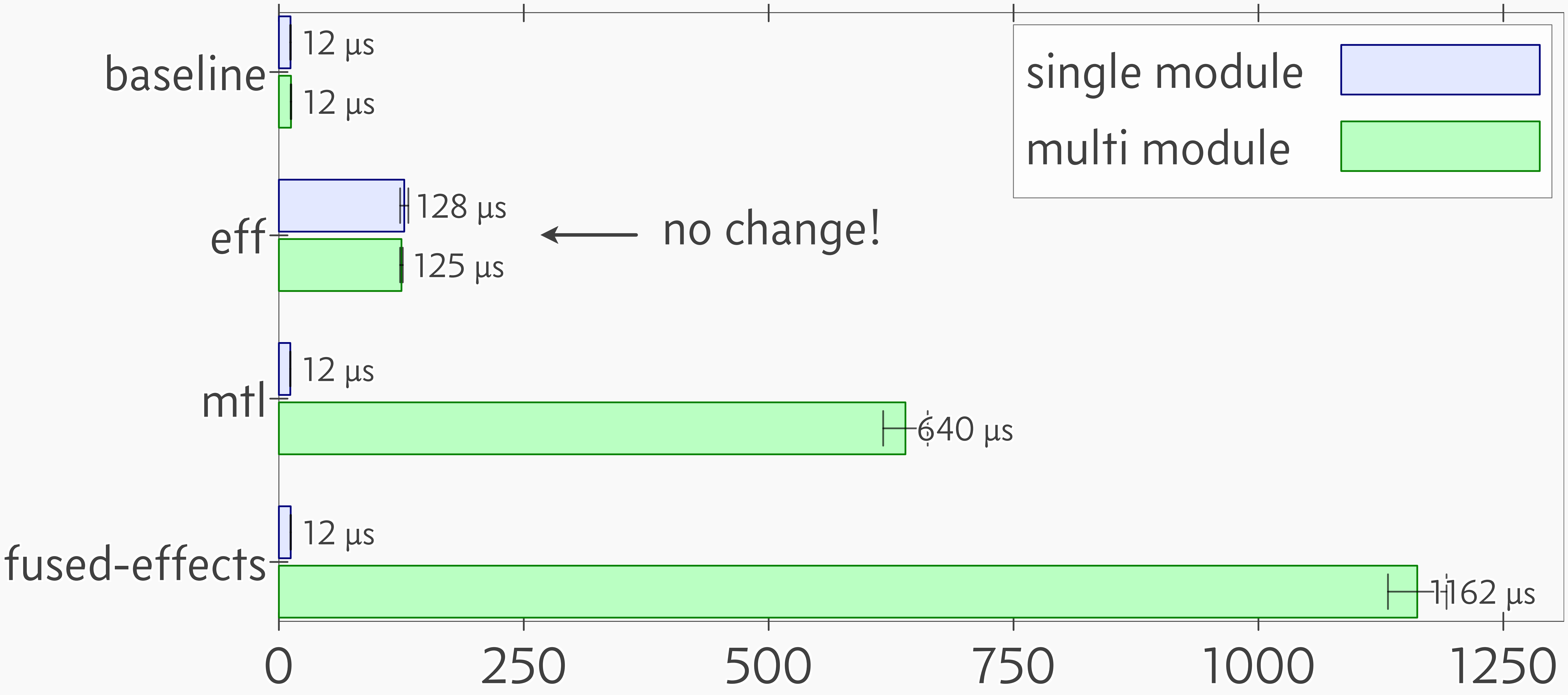
- API is similar to free-like approaches (actually simpler!).
- Provides a single monad, named `Eff`.
- Basically `ReaderT` over `ST` under the hood.
  - Reader context holds current handlers.
  - Wraps the unsafe primops in a safe API.
- Effect dispatch takes constant time (not amortized).
- Faster for most use cases than unspecialized `mtl`.

EFF: THE NUMBERS

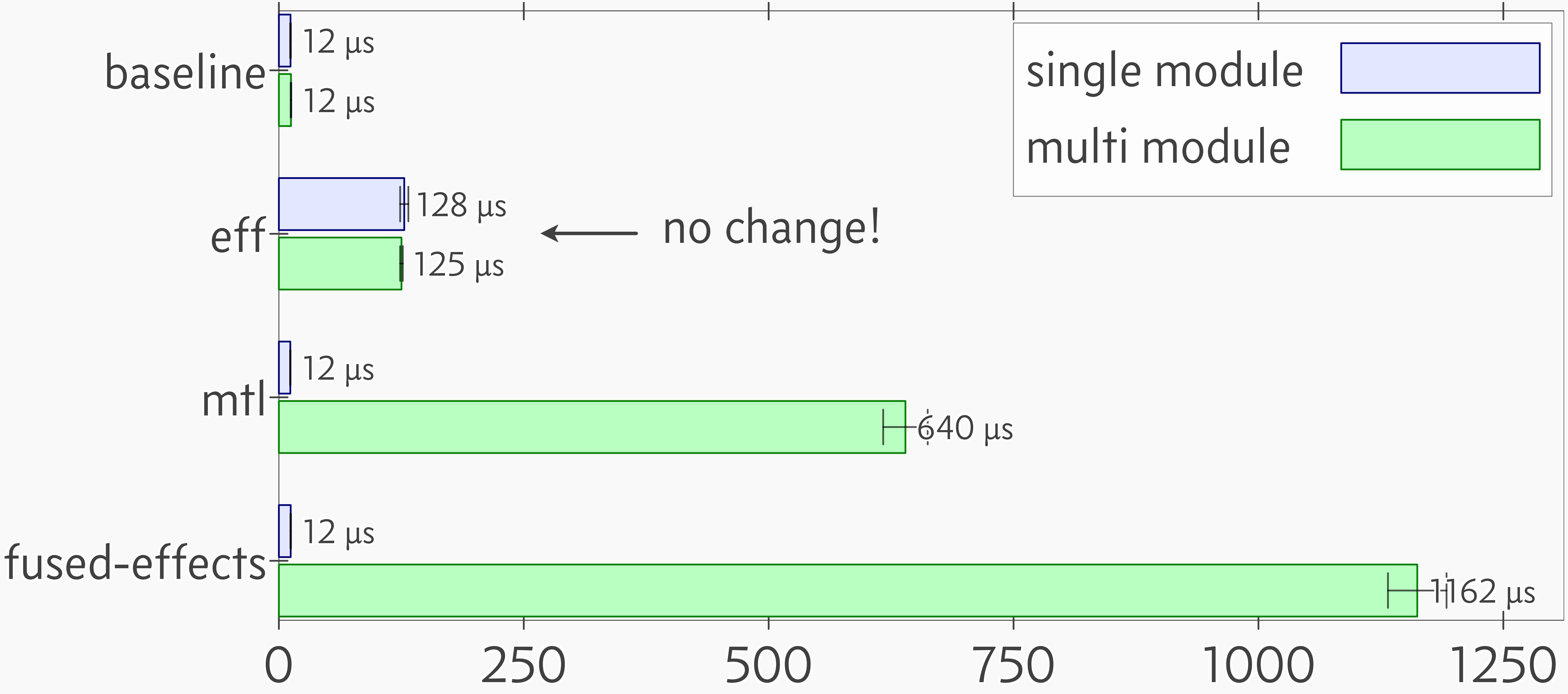
EFF: THE NUMBERS



EFF: THE NUMBERS

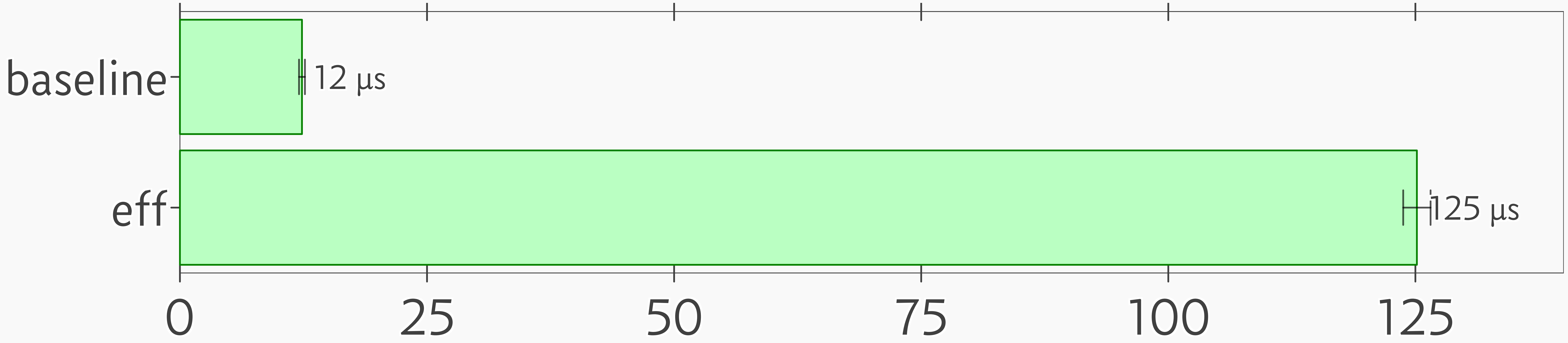


EFF: THE NUMBERS

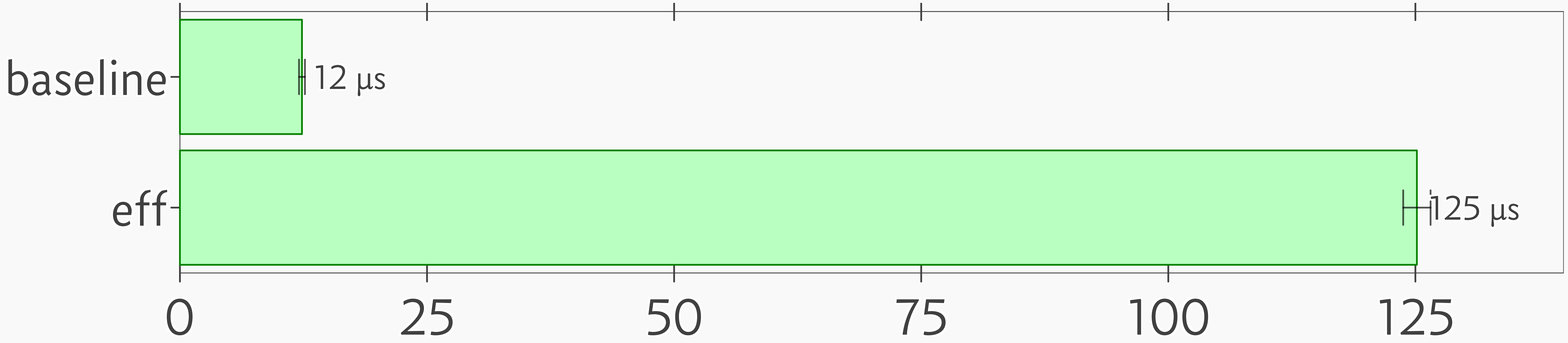


`eff`'s performance does *not* depend on compiler optimizations!

EFF: THE NUMBERS



EFF: THE NUMBERS



Should we be worried?

COUNTDOWN: THE REST OF THE STORY

```
program :: Int → (Int, Int)
program n = if n ≤ 0 then (n, n)
            else program (n - 1)
```

COUNTDOWN: THE REST OF THE STORY

```
program :: Int → (Int, Int)
program n = if n ≤ 0 then (n, n)
            else program (n - 1)
```

Why *so* much faster?

```haskell
program :: Int → (Int, Int)
program n = if n ⩽ 0 then (n, n)
            else program (n - 1)
```

```haskell
program :: Int → (Int, Int)
program n = if n ⩽ 0 then (n, n)
                     else program (n - 1)
```

```haskell
$wprogram :: Int# → (# Int#, Int# #)
$wprogram n = if n ⩽# 0# then (# n, n #)
                         else $wprogram (n -# 1#)
```

```
$wprogram :: Int# → (# Int#, Int# #)
$wprogram n = if n ⩽# 0# then (# n, n #)
                             else $wprogram (n -# 1#)
```

```
$wprogram :: Int# → (# Int#, Int# #)
$wprogram n = if n ⩽# 0# then (# n, n #)
                              else $wprogram (n -# 1#)
```

▼

```
_c4ap: addq $16,%r12
       cmpq 856(%r13),%r12
       ja _c4aB
_c4aA: testq %r14,%r14
       jle _c4aw
_c4av: addq $-16,%r12
       decq %r14
       jmp _c4ap
```

```
$wprogram :: Int# → (# Int#, Int# #)
$wprogram n = if n ≤# 0# then (# n, n #)
                              else $wprogram (n -# 1#)
```

```
_c4ap: addq $16,%r12
       cmpq 856(%r13),%r12
       ja _c4aB
_c4aA: testq %r14,%r14
       jle _c4aw
_c4av: addq $-16,%r12
       decq %r14
       jmp _c4ap
```

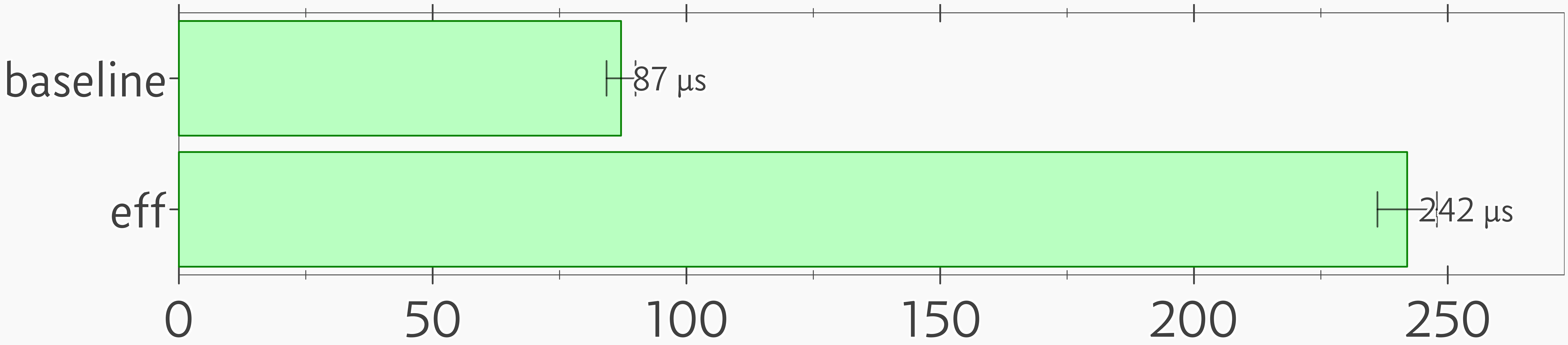
8 instructions!

```
program :: Int → (Int, Int)
program n = if n ⩽ 0 then (n, n)
                  else program (n - 1)
```

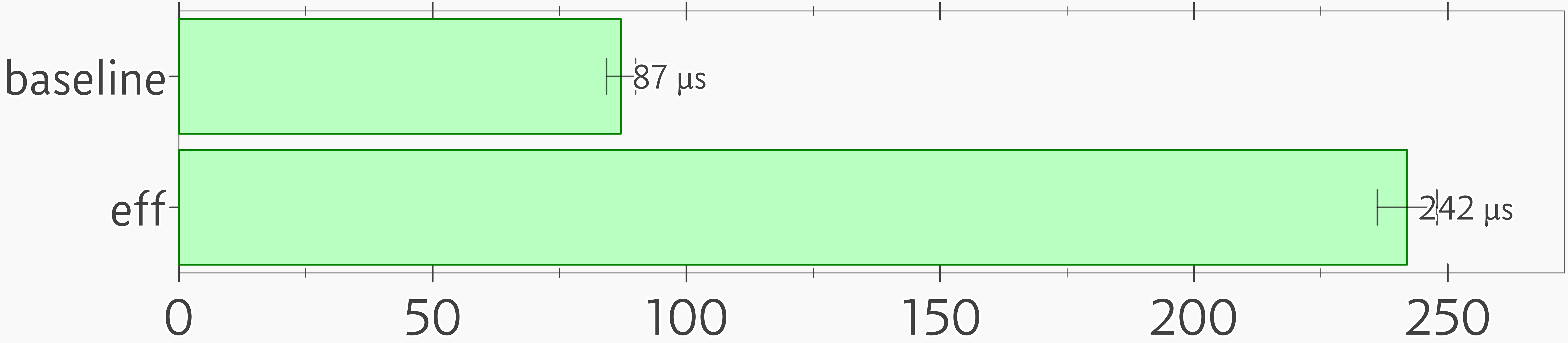
```haskell
program :: Int → (Int, Int)
program n = if n ⩽ 0 then (n, n)
            else program (n - 1)
```

```haskell
program :: Integer → (Integer, Integer)
program n = if n ⩽ 0 then (n, n)
            else program (n - 1)
```

COUNTDOWN: NO UNBOXING



COUNTDOWN: NO UNBOXING



Not quite so bad, after all!

Phew.

RECAP

RECAP

→ Countdown benchmarks and effect dispatch.

RECAP

- → Countdown benchmarks `>>=` and effect dispatch.
- → `mtl` and `fused-effects` rely on compiler optimizations.

RECAP

- Countdown benchmarks `>>=` and effect dispatch.
- `mtl` and `fused-effects` rely on compiler optimizations.
- Those optimizations are very often not viable.

RECAP

- Countdown benchmarks `>>=` and effect dispatch.
- `mtl` and `fused-effects` rely on compiler optimizations.
- Those optimizations are very often not viable.
- `eff` closes the gap by being inherently fast:

RECAP

- Countdown benchmarks `>>=` and effect dispatch.
- `mtl` and `fused-effects` rely on compiler optimizations.
- Those optimizations are very often not viable.
- `eff` closes the gap by being inherently fast:
  - It exposes more local transformations to the optimizer.

RECAP

- Countdown benchmarks `>>=` and effect dispatch.
- `mtl` and `fused-effects` rely on compiler optimizations.
- Those optimizations are very often not viable.
- `eff` closes the gap by being inherently fast:
  - It exposes more local transformations to the optimizer.
  - Delimited continuation primops avoid CPSing.

RECAP

- Countdown benchmarks `>>=` and effect dispatch.
- `mtl` and `fused-effects` rely on compiler optimizations.
- Those optimizations are very often not viable.
- `eff` closes the gap by being inherently fast:
    - It exposes more local transformations to the optimizer.
    - Delimited continuation primops avoid CPSing.
    - Result: `eff` handily wins the benchmark shootout.

RECAP

- Countdown benchmarks `>>=` and effect dispatch.
- `mtl` and `fused-effects` rely on compiler optimizations.
- Those optimizations are very often not viable.
- `eff` closes the gap by being inherently fast:
  - It exposes more local transformations to the optimizer.
  - Delimited continuation primops avoid CPSing.
  - Result: `eff` handily wins the benchmark shootout.
- Numbers are not enough. It is our responsibility to ask why.

CLOSING THOUGHTS

CLOSING THOUGHTS

→ The `eff` story is not over.

CLOSING THOUGHTS

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.

CLOSING THOUGHTS

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.
  - The GHC proposal needs to be accepted.

CLOSING THOUGHTS

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.
  - The GHC proposal needs to be accepted.
- The design of `eff` itself could be its own talk!

CLOSING THOUGHTS

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.
  - The GHC proposal needs to be accepted.
- The design of `eff` itself could be its own talk!
- `mtl`'s performance is misunderstood and overhyped!

CLOSING THOUGHTS

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.
  - The GHC proposal needs to be accepted.
- The design of `eff` itself could be its own talk!
- `mtl`'s performance is misunderstood and overhyped!
- Effect systems are too foundational to ignore perf.

CLOSING THOUGHTS

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.
  - The GHC proposal needs to be accepted.
- The design of `eff` itself could be its own talk!
- `mtl`'s performance is misunderstood and overhyped!
- Effect systems are too foundational to ignore perf.
- We really do need good real-world benchmarks.

CLOSING THOUGHTS

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.
  - The GHC proposal needs to be accepted.
- The design of `eff` itself could be its own talk!
- `mtl`'s performance is misunderstood and overhyped!
- Effect systems are too foundational to ignore perf.
- We really do need good real-world benchmarks.
- Someday: effect specialization?

THE END

- Countdown benchmarks `>>=` and effect dispatch.
- `mtl` and `fused-effects` rely on compiler optimizations.
- Those optimizations are very often not viable.
- `eff` closes the gap by being inherently fast:
  - It exposes more local transformations to the optimizer.
  - Delimited continuation primops avoid CPSing.
  - Result: `eff` handily wins the benchmark shootout.
- Numbers are not enough. It is our responsibility to ask why.

- The `eff` story is not over.
  - Need to plumb in support for `IO` exceptions.
  - The GHC proposal needs to be accepted.
- The design of `eff` itself could be its own talk!
- `mtl`'s performance is misunderstood and overhyped!
- Effect systems are too foundational to ignore perf.
- We really do need good real-world benchmarks.
- Someday: effect specialization?

`eff`: https://github.com/hasura/eff
benchmarks: https://github.com/ocharles/effect-zoo
proposal: https://github.com/ghc-proposals/ghc-proposals/pull/313
Hasura: https://hasura.io
me: https://lexi-lambda.github.io