Linear Haskell: some hows and whys

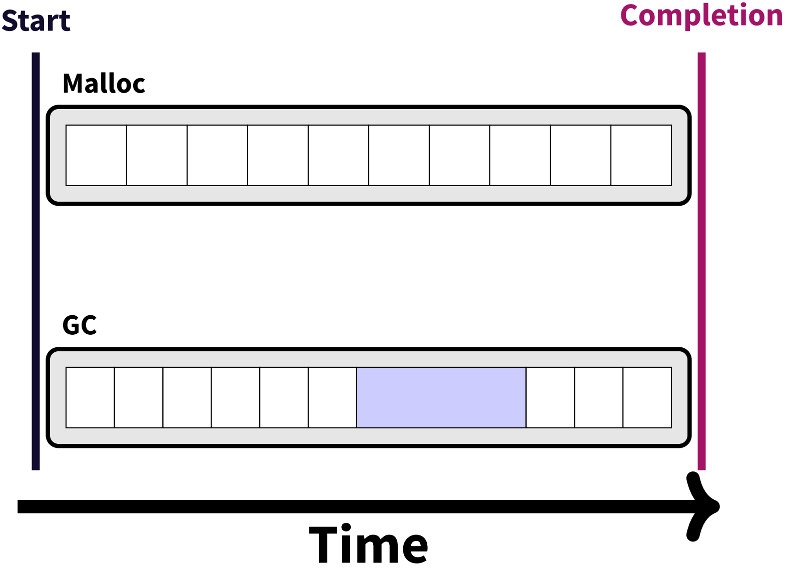
Arnaud Spiwack

Eutypes 2018

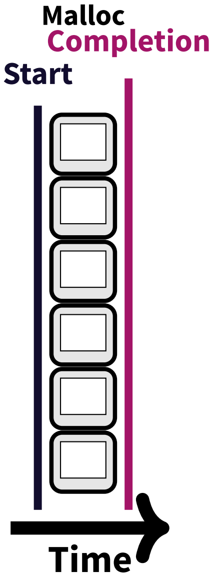
tweag.io

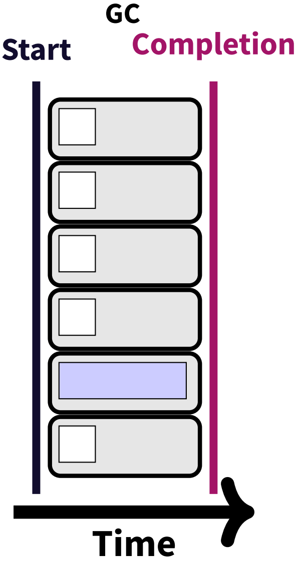
Idea ➡ Joke ➡ Let's do it!

Let’s talk about garbage collection



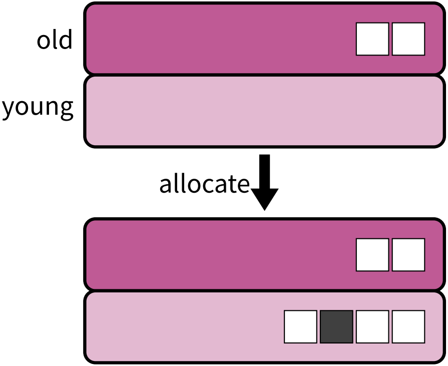
Garbage collection, latency, synchronisation



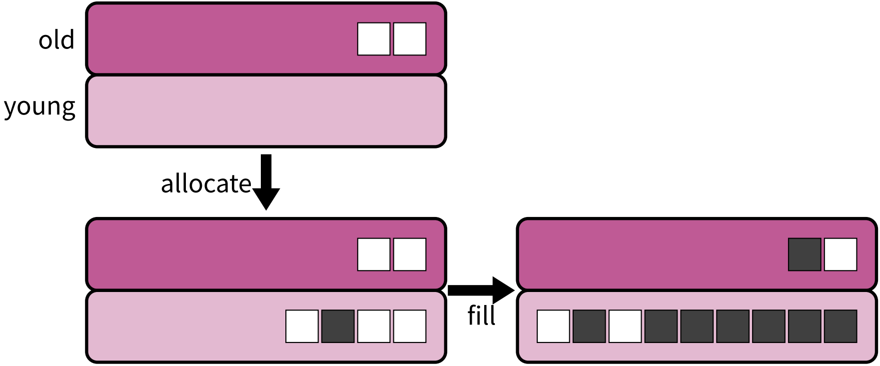


Generational hypothesis

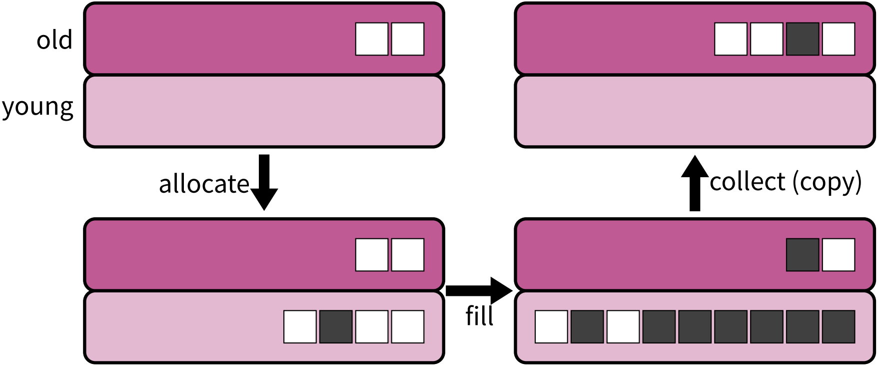
Generational hypothesis



Generational hypothesis



Generational hypothesis



Manual memory management

```
data Se82

do let (x :: Se82) = init
   doSomething x
```

⇒

```
data Se82

do (x :: Se82) ← malloc
   doSomething x
   free x
   doMore x
```

Manual memory management

```
data Se82

doSomething x
```

```
data Se82

do (x :: Se82) ← malloc
    doSomething x
    free x
    doMore x
```

Don't forget to free!

Don't use x anymore!

Again in the fields of x!

Linear functions

A linear function

$\mathsf{succ} :: \mathsf{Int} \multimap \mathsf{Int}$
$\mathsf{succ}\ i = i + 1$

$\mathsf{repeat} :: \mathsf{Int} \to [\mathsf{Int}]$
$\mathsf{repeat}\ n = n : \mathsf{repeat}\ n$

$\mathsf{constFalse} :: \mathsf{Int} \to \mathsf{Bool}$
$\mathsf{constFalse}\ n = \mathsf{False}$

Not linear functions

Linear functions

Linear functions

$f :: \mathsf{Bool} \multimap \mathsf{Int} \multimap \mathsf{Int}$
$f\ b\ i = \mathbf{if}\ b\ \mathbf{then}\ i + 1\ \mathbf{else}\ i + 2$

$k :: \mathsf{Int} \multimap (\mathsf{Int}, \mathsf{Bool})$
$k\ i = (i, \mathsf{True})$

$g :: \mathsf{Int} \to [\mathsf{Int}]$
$g\ n = 42 : \mathrm{repeat}\ n$

$h :: \mathsf{Bool} \multimap \mathsf{Int} \to \mathsf{Bool}$
$h\ b\ i = \mathbf{if}\ b\ \mathbf{then}\ i + 1\ \mathbf{else}\ 57$

Not linear functions

Malloc: revisited

$\textbf{data}\ \mathrm{IO}_1\ a$

$$
\begin{aligned}
(\ggg\!=) &:: \mathrm{IO}_1\ a \multimap (a \multimap \mathrm{IO}_1\ b) \multimap \mathrm{IO}_1\ b \quad \text{— must be linear}\\
\mathit{malloc} &:: \mathrm{Storable}\ a \Rightarrow \mathrm{IO}_1\ (\mathrm{Ptr}\ a)\\
\mathit{free} &:: \mathrm{Ptr}\ a \multimap ()
\end{aligned}
$$

Malloc: revisited

data IO$_1$ a

$(\gg=)$ :: IO$_1$ a ⊸ (a ⊸ IO$_1$ b) ⊸ IO$_1$ b — **must be linear**
malloc :: Storable a ⇒ IO$_1$ (Ptr a)
free :: Ptr a ⊸ ()

do (x ::$_1$ Se82) ← malloc
x′ ← doSomething x — doSomething **doesn't free** x
free x′ — **we can't forget** free
doMore — x **isn't available in** doMore

Malloc: re^{2}visited

$$
\begin{aligned}
&\mathsf{malloc} :: \mathsf{Storable}\ \mathsf{a} \Rightarrow \mathsf{a} \to (\mathsf{Ptr}\ \mathsf{a} \multimap \mathbf{U}\ \mathsf{a}) \multimap \mathbf{U}\ \mathsf{a} \\
&\mathsf{free} \quad\ \ :: \mathsf{Ptr}\ \mathsf{a} \multimap ()
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{malloc}\ \mathsf{init}\ \$\ \lambda(\mathsf{x} ::_1\ \mathsf{Se82}) \to \\
&\quad \mathbf{let}\ \mathsf{x}' = \mathsf{doSomething}\ \mathsf{x}\ \mathbf{in} \\
&\qquad \mathsf{free}\ \mathsf{x}'\ \ `\mathsf{seq}`\ \ \mathsf{doMore}
\end{aligned}
$$

What we've learnt so far

- GC pauses are costly in distributed settings
- Generational hypothesis: younger data dies sooner
  - Old heap collection causes the pauses
- Long-lived data contribute to pauses: we can manage them manually
  - Can be done safely with linear types

What we’ve learnt so far (and one thing we haven’t)

- GC pauses are costly in distributed settings
- Generational hypothesis: younger data dies sooner
  - Old heap collection causes the pauses
- Long-lived data contribute to pauses: we can manage them manually
  - Can be done safely with linear types

A new idea

- Prevent data from moving to the old heap
  - Allocate fewer short-lived data too

Fusion

GHC: rules

$$\text{map f (map g) l} \quad \rightsquigarrow \quad \text{map (f} \circ \text{g) l}$$

$$\text{foldr k z (build g)} \quad \rightsquigarrow \quad \text{g k z}$$

Vector: stream fusion (mostly inlining)

$$\begin{array}{lll} \text{stream (unstream s)} & \rightsquigarrow & \text{s} \\ \text{unstream (stream v)} & \rightsquigarrow & \text{clone v} \end{array}$$

Should I allocate?

$\text{replicate} :: \text{Int} \to a \to \text{Vector}\ a$

— Relies on stream fusion

Should I allocate?

What if you're writing to a file (memory map)?

What if the size is known up the pipeline?

replicate :: Int → a → Vector a
— Relies on stream fusion

What if the consumer is a foreign function?

What if you're writing to another computer (RDMA)?

What if the consumer isn't inlined?

Destination-passing style

$\text{replicate} :: \text{Int} \to \text{a} \to \text{MVector a} \to ()$

Destination-passing style

$$\text{replicate} :: \text{Int} \to \text{a} \to \text{MVector a} \to ()$$

I should use ST or IO

Destination-passing style

```
replicate    :: (MonadPrim m)
             ⇒ a → MVector (PrimBase m) a ⊸ m ()

mmap         :: FilePath → IO (MVector Word8)
unsafeFreeze :: (MonadPrim m)
             ⇒ MVector (PrimBase m) a → m (Vector a)
```

Destination-passing style

$$
\begin{aligned}
\text{replicate} &:: a \to \text{DVector}\ a \multimap () \\
\text{split} &:: \text{Int} \to \text{DVector}\ a \multimap (\text{DVector}\ a, \text{DVector}\ a) \\
\\
\text{mmap} &:: \text{FilePath} \to \text{IO}_1\ (\text{DVector}\ a) \\
\text{alloc} &:: (\text{DVector}\ a \multimap ()) \multimap \text{Vector}\ a
\end{aligned}
$$

Reversed constructors

```
data Tree where
  Node :: (Tree, Tree) ⊸ Tree
  Leaf :: Int          ⊸ Tree

nodeD :: D Tree ⊸ (D Tree, D Tree)
leadD :: D Tree ⊸ D Int
```

```
data [a] where
  (:) :: (a, [a]) ⊸ [a]
  []  :: ()       ⊸ [a]

consD :: D [a] ⊸ (D a, D [a])
nilD  :: D [a] ⊸ ()
```

A tale of recursive maps

```
fill   :: a ⊸ D a ⊸ ()
alloc  :: (D [a] ⊸ ()) ⊸ [a]

consD :: D [a] ⊸ (D a, D [a])
nilD  :: D [a] ⊸ ()
```

```
map :: (a → b) → [a] → [b]
map f l = alloc $ λd → loop l d
  where
    loop :: [a] → D [b] ⊸ ()
    loop []      d  = nilD d
    loop (a : l) d  = fill (f a) db `seq` loop l dr
      where (db, dr) = consD d
```

A tale of recursive maps

```
fill   :: a ⊸ D a ⊸ ()
alloc  :: (D [a] ⊸ ()) ⊸ [a]

consD  :: D [a] ⊸ (D a, D [a])
nilD   :: D [a] ⊸ ()
```

```
map :: (a → b) → [a] → [b]
map f l = alloc $ λd → nilD (foldl′ step d l)
  where
    step :: D [b] ⊸ a ⊸ D [b]
    step    d a = fill (f a) db `seq` dr
      where (db, dr) = consD d
```

Representation

$$\textbf{data}\ [a] = \text{Slice}\ \{\text{base} :: \text{MArray}\ a, \text{offset} :: \text{Int}, \text{last} :: \text{IORef Int}\}$$

$$\begin{aligned}
&\text{nilD}\quad\ l = \text{unsafePerformIO}\ \$\ \text{writeIORef}\ (\text{last}\ l)\ (\text{offset}\ l)\\
&\text{consD}\ l = ((\text{base}\ l, \text{offset}\ l), l\ \{\text{offset} = \text{offset}\ l + 1\})
\end{aligned}$$

$$\text{fill}\ a\ (l, \text{offset}) = \text{unsafePerformIO}\ \$\ \text{writeArray}\ l\ a$$

$$\begin{aligned}
&\text{listView} :: [a] \rightarrow \text{Maybe}\ (a, [a])\\
&\text{pattern}\ (a : l) \leftarrow \text{Just}\ (a, l) \leftarrow \text{listView}\\
&\text{pattern}\ [\,]\qquad\ \leftarrow \text{Nothing}\quad \leftarrow \text{listView}
\end{aligned}$$

The big idea

Facilities offered by the compiler

- Garbage collection
- Fusion
- Data representation

You don't have to think about them.
But when you need to think about them, the compiler is no help.

- With linear types you can take them safely back in your own hands

> *A programming language is low level when its programs require attention to the irrelevant.*
> *— Alan Perlis*

Follow the adventures of Linear Haskell

- tweag.io/blog
- github.com/tweag/ghc/tree/linear-types
- github.com/tweag/linear-types
- https://github.com/ghc-proposals/ghc-proposals/pull/111
- https://ghc.haskell.org/trac/ghc/wiki/LinearTypes