Linear Types in Haskell

David Overton

Melbourne Haskell Users Group - July 2020

Linear types can change the world!

— Philip Wadler, 1990

Outline

- Motivation
- What are linear types?
- Proposed GHC extension
- Linear base library
- Trying it out
- Examples
- Related work
- Time for questions / discussion / hacking

Motivation

Linear types make using resources safer and more predictable:

— Safe `malloc`/`free` memory management
— Safe usage of file handles - closed exactly once, not used after close
— Safe mutable arrays - destructive update
— Safer inlining and guaranteed fusion
— Session types

What are Linear Types?

Linearity

— A function `f` is *linear* when
if `f u` is *consumed exactly once*
then `u` is *consumed exactly once*

— *Consuming* a **value** of a data type *exactly once* means evaluating it to head normal form exactly once, discriminating on its tag any number of times, then consuming its fields exactly once.

— *Consuming* a **function** *exactly once* means applying it and conuming its result exactly once.

Proposed GHC extension

```
A #-> B
```

is the type of linear functions from A to B.

— In papers often written $A \multimap B$ (from Linear Logic).
— Alternative (rejected) proposals:
  — `A -o B`
  — `A ->. B`

Simple examples

```
-- Valid:
const :: a -> b -> a
const a _ = a

-- Valid:
const :: a #-> b -> a

-- Not valid:
const :: a -> b #-> a

-- Not valid:
dup :: a #-> (a, a)
dup x = (x, x)
```

Most "normal" data types will have linear constructors. E.g.

```
data Maybe a
     = Nothing
     | Just a
```

is equivalent to

```
data Maybe a where
     Nothing :: Maybe a
     Just :: a #-> Maybe a
```

Explicitly *unrestricted* or non-linear constructor using GADT syntax:

```
data Unrestricted a where
  Unrestricted :: a -> Unrestricted a
```

Two possible types of `map`:

```
map :: (a -> b) -> [a] -> [b]
map :: (a #-> b) -> [a] #-> [b]
```

— The `map` function preserves the *multiplicity* of its function argument

— But we don't want to have to define it twice

To avoid code duplication, functions can have *multipicity polymorphism*.

```
map :: (a #p-> b) -> [a] #p-> [b]
```

where `p` is a multiplicity parameter.

— Linear functions have multiplicity 1 or `One`

— Unrestricted functions have multiplicity ω or `Many`

Multiple multiplicities

```
(.) :: (b #p-> c) -> (a #q-> b) -> a #(p ':* q)-> c
```

would require 4 different types without polymorphism:

```
(.) :: (b ->  c) -> (a ->  b) -> a ->  c
(.) :: (b #-> c) -> (a ->  b) -> a ->  c
(.) :: (b ->  c) -> (a #-> b) -> a ->  c
(.) :: (b #-> c) -> (a #-> b) -> a #-> c
```

— `(p ':* q)` multiplies the multiplicies p and q:

:*	1	ω
1	1	ω
ω	ω	ω

Details

```
data Multiplicity = One | Many

-- Type families
type family (:+) (p :: Multiplicity) (q :: Multiplicity) :: Multiplicity
type family (:*) (p :: Multiplicity) (q :: Multiplicity) :: Multiplicity

-- New type constructor
FUN :: Multiplicity -> forall (r1 r2 :: RuntimeRep). TYPE r1 -> TYPE r2

-- FUN p a b ~ a #p-> b

type (->) = FUN 'Many
type (#->) = FUN 'One
```

Linear base library

Linear base library

https://github.com/tweag/linear-base

Needed to write anything useful with linear types

Provides:

— Linear Prelude
— Linear arrays
— Linear I/O
— …

Trying it out

Trying it out

- GHC 8.11 has a partial implementation of `-XLinearTypes`
  - Missing support for `where` and `let`
  - Missing support for multiplicity polymorphism
  - Probably missing other stuff too
- GHC 9.0.1 will have more complete support, but still considered "experimental"
  - Due for release in September 2020
- `linear-base` library has instructions for setting up a Stack project with GHC 8.11 and `linear-base` using `nix`

Examples

I/O

```
-- Ensures I/O state is linearly threaded, meaning it is safe to
-- expose the IO constructor
newtype IO a = IO (State# RealWorld #-> (# State# RealWorld, a #))

-- Resource-aware IO monad (equivalent of ResourceT IO)
newtype RIO a = RIO (IORef ReleaseMap -> Linear.IO a)

-- Note: non-linear
openFile :: FilePath -> System.IOMode -> RIO Handle

-- hClose consumes the handle so it can't be used again after it's closed.
hClose :: Handle #-> RIO ()

-- Other I/O operations must also be linear with respect to handle
-- meaning each operation needs to return a new handle.
hPutChar :: Handle #-> Char -> RIO Handle
hGetChar :: Handle #-> RIO (Unrestricted Char, Handle)
```

I/O

```
linearGetFirstLine :: FilePath -> RIO (Unrestricted Text)
linearGetFirstLine fp = do
  handle <- Linear.openFile fp System.ReadMode
  (t, handle') <- Linear.hGetLine handle
  Linear.hClose handle'
  return t
  where
    Control.Builder {..} = Control.monadBuilder

linearPrintFirstLine :: FilePath -> System.IO ()
linearPrintFirstLine fp = do
  text <- Linear.run (linearGetFirstLine fp)
  System.putStrLn (unpack text)
```

Excercise:

— What happens if we forget to close the file handle?

— Use the handle more than once?

— Use after close?

Mutable arrays

```
fromList :: [a] -> (Array a #-> Unrestricted b) -> Unrestricted b
toList :: Array a #-> (Array a, [a])
length :: Array a #-> (Array a, Int)
write :: Array a #-> Int -> a -> Array a
read :: Array a #-> Int -> (Array a, Unrestricted a)
```

— Note use of continuation passing style for `fromList`

— Required to ensure that the array is always used linearly

— Operations return a "new" array

Excercise:

— Write a function to swap two elements in an array.

```
void swap(int a[], int i, int j) {
  int t = a[i];
  int u = a[j]
  a[i] = u;
  a[j] = t;
}
```

```
swap :: Array a #-> Int -> Int -> Array a
swap a i j =
  Array.read a i & \case
    (a', Unrestricted t) -> Array.read a' j & \case
      (a'', Unrestricted u) -> Array.write a'' i u &
        \a''' -> Array.write a''' j t
```

Related Work

- *Linear Types Can Change the World!*, Philip Wadler, 1990
  - Based on Linear Logic
  - Linearity on types rather than functions
- Clean
  - Uniqueness types - "I have the only unique reference to this object"
  - Used for I/O state
- Mercury
  - Uniqueness using modes
- Rust
  - Ownership and borrowing have similarites to uniqueness and linearity, respectively

The Linearity Design Space[1]

	Linearity on the arrows	**Linearity on the kinds**
Linear types	Linear Haskell	Rust (borrowing)
Uniqueness types		Rust (ownership), Clean, Mercury (modes)

[1] Taken from SPJ's talk

References

— *Linear Haskell - Practical Linearity in a Higher-Order Polymorphic Langauge*, Bernardy et al, Nov 2017.

— GHC Proposal

— Linear Base Library

— Examples

— Simon Peyton Jones - Linear Haskell: practical linearity in a higher-order polymorphic language

— *Linear types can change the world!* - Philip Wadler, 1990