A Brief Look at Linear Types in GHC 8.4

Erik de Castro Lopo

July 28, 2017

State of Play

- GHC 8.2.1 was released this past weekend.
- GHC devs aim to release GHC 8.4 Q1 2018.
- 8.4 will have a new feature, Linear Types.

Work by

- Arnaud Spiwack (Tweag I/O).
- Jean-Philippe Bernary (U of Gothenburg).
- Edvard Hübinette (U of Gothenburg / GSoC / Tweag).

Current Status

- Working prototype at

  `https://github.com/tweag/linear-types/releases` (paper/tarball)

  `https://github.com/tweag/ghc` (linear-types branch).

- According to **`diffstat`**:

  122 files changed, 1242 insertions(+), 813 deletions(-)

Don’t be Alarmed!

- Existing programs continue to typecheck.
- Existing data types can be used as-is.
- Linear types are opt-in.

Fundamental Idea

- A value with a Linear Type **must** be used **exactly** once.
- Not zero times, not more than once.

Naive Approach

```
newtype Linear a = Linear a
```

- Still needs a compiler hack to bake in linearity.
- Doesn’t compose well.
- Would be painful to use.

A better approach

Linear ~~Types~~ Function Arrows

```
func ::  a ⊸ b
```

- ⊸ is a sub class of the existing **->**.
- Linearity-on-the-arrow supports linearity polymorphism.
- Functions can be written to work uniformly in both linear and non-linear code.

Unicode character U+22B8

Operationally

The Linear arrow guarantees that if

```
f x
```

is consumed exactly once, then the argument

```
x
```

is consumed exactly once.

Consume exactly once:

- To consume a value of atomic base type (like Int or Ptr) exactly once, just evaluate it.
- To consume a function exactly once, apply it to one argument, and consume its result exactly once.
- To consume a pair exactly once, pattern-match on it, and consume each component exactly once.
- In general, to consume a value of an algebraic data type exactly once, pattern-match on it, and consume all its linear components exactly once.

Developing Intuitions

Standard Haskell defines a function that returns the first element of a pair:

```
fst ::  (a, b) ⊸ a
fst (a, _) = a
```

Developing Intuitions

Standard Haskell defines a function that duplicates a value to returns a pair:

```
dup ::  a ⊸ (a, a)
dup a = (a, a)
```

Linearity Polymorphism

List append example.

What does this buy us?

- Avoid memory allocation by updating in place.
- Resource management.
- Enforcing invariants in protocols.
- Safe replacement for `unsafeFreeze`.
- Safe mutable arrays.

Example: Linear map function

The good old **`map`** function but with the linear arrow.

```
lmap ::  (a ⊸ b) -> [a] ⊸ [b]
lmap _ [] = []
lmap f (x:xs) = f x :  lmap f xs
```

Example: Resource Management

When we close a file handle, we want to do it **once** and then not use the handle again.

```
closeHandle ::  Handle ⊸ IO ()
closeHandle hdl = ...
```

Example: Protocols

Example in the paper.

Example: Safe Vector freeze

Let's look at the example of reversing a vector.

```
reverse ::  Vector a -> IO (Vector a)
```

- Why isn't this pure?

Example: New Vector API

```
newVector ::  Int -> (MVector a ⊸ Vector b) ⊸ Vector b

write ::  MVector a ⊸ Int ⊸ a -> MVector a

read ::  MVector a ⊸ Int -> (MVector a, a)

freeze ::  MVector a ⊸ Vector b
```

Further Work

Let's define a **`Functor`** with a linear arrow:

```
class LFunctor a where
    fmap ::  (a ⊸ b) -> [a] ⊸ [b]
```

Does replacing the unrestricted arrow with a linear arrow make sense?