6 Future work

Enlarging the scope

Currently we only look at the *immediate* scope of let bindings. Technically speaking there is nothing preventing our optimisation from working with a more indirect scoping mechanism. Indeed the following should trigger our optimisation

```
defaultVal : MyData
defaultVal = MkDefault 3

update : (1 rec : MyData) -> MyData
update (MkDefault n) = MkDefault (S n)
update (MkOther n) = MkOther (S (S n))

operate : MyData
operate = let 1 def = defaultVal
              1 newVal = update def in
              update newVal
```

But it will not because `defaultVal` is not a constructor, it's a function call that returns itself a constructor.

One implementation strategy would be to wait for the compiler to inline those definitions and then run our optimiser without further changes.

Another optimisation would be to aggressively follow references to see if they result in plain data constructors and replace the entire call chain by the constructor itself, and then run our optimisation.

While both those strategies are valid they incur a cost in terms of complexity and compile time that may not be worth the effort in terms of performance results. They could be hidden behind a -O3 flag, but that kind of effort is probably better spend in making the ergonomics of linear types more streamlined, which would help make those optimisations more commonplace. Which is the topic of the next section

Making linearity easier to use

There are multiple barriers that make linearity harder to use than one might expect. They roughly end up in two buckets:

- I want to use linearity but I cannot
- I have a linear variable and that's actually annoying

Not linear enough

The first one appears when the programmer tries to make thoughtful usage of linear and erased annotation but finds that other parts of existing libraries do not support linearity. Here are a couple of examples

```
operate : (1 n : Nat) -> (1 m : Nat) -> Int
operate n m = n + m
```

gives the error

```
Trying to use linear name n in non-linear context
```

Because the `+` interface is defined as

```
interface Num ty where
    (+) : ty -> ty -> ty
```

Despite addition on `Nat` being defined linearly

```
plus : (1 n : Nat) -> (1 m : Nat) -> Nat
plus Z m = m
plus (S n) m = S (plus n m)
```

A similar problem occurs with interfaces

```
interface Monad (Type -> Type) where
    ...
```

```
data MyData : (0 ty : Type) -> Type where
    ...

instance Monad MyData where
    ...
```

```
Expected Type -> Type
got (0 ty : Type) -> Type
```

One way to solve those issues would be to have linearity polymorphism and be able to abstract over linearity annotations. For example the map function could be written as

```
map : forall l . ((l v : a) -> b) -> (l ls : List a) -> List b
map f [] = []
map f (x :: xs) = f x :: map f xs
```

That is, the list is linearly consumed iff the higher order function is linear. What it means for our interface problem is that it could be rewritten as

```
interface forall l . Functor (m : (l _ : Type) -> Type) where
    ...
interface forall l . Functor {l} m ⇒ Applicative {l} m where
    ...
interface forall l . Applicative {l} m ⇒ Monad {l} m where
    ...
```

A similar solution could be provided for `Num`

```
interface Num ty where
    (+) : forall l. (l n : ty) -> (l m : ty) -> ty
```

So that it can be used with both linear and non-linear variables.

Too linear now

We've already mentionned before how benefitial it would be for our optimisation strategy to be *100%* linear in every aspect. We also mentioned how this is a problem to implement basic functionality like `drop` and `copy`, but those are artifical examples, rarely does a programmer need to call `copy` or `drop` in industrial applications. Therefore I will present a couple of situation where being *entirely* linear results in tricky code or impossible code and then propose a solution.

Logging

This is a common scenario, you're trying to debug effectful code, and for this you're spreading around log statements hoping that running the program will give you insight into how it's running.

```
do datas <- getData arg1 arg2
   Just success <- trySomething datas (options.memoized)
     | _ ⇒ pure $ returnError "couldn't make it work"
   case !(check_timestamp success) of
      Safe t v ⇒ functionCall t v
      Unsafe t ⇒ trySomethingElse t
      UnSynchronized v ⇒ functionCall 0 v
      Invalid ⇒ pure $ returnError "failed to check"
```

Assuming everything is linear, there is no possible way to add a new print statement without getting a linearity error:

```
do datas <- getData arg1 arg2
   Just success <- trySomething datas (options.memoized)
     | _ ⇒ pure $ returnError "couldn't make it work"
   putStrLn $ show success -- <- one use here
   --
   -- And one use there ----|
   --                       v
   case !(check_timestamp success) of
      Safe t v ⇒ functionCall t v
      Unsafe t ⇒ trySomethingElse t
      UnSynchronized v ⇒ functionCall 0 v
      Invalid ⇒ pure $ returnError "failed to check"
```

Multiplication

We've seen how to add two Nat linearly, but what about multiplication and exponentiation?

```
mult : (1 n : Nat) -> (
```