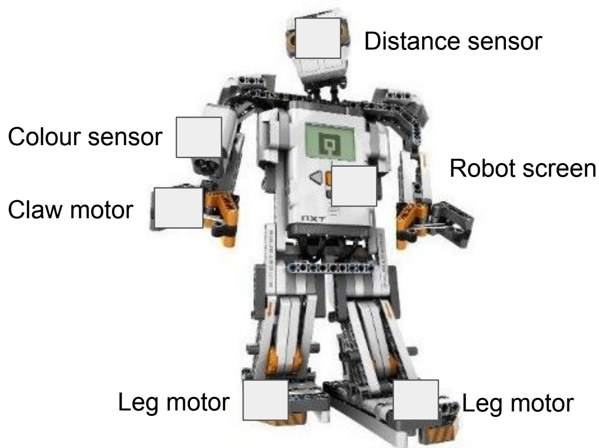
Resolving data flow dependencies of concurrent programs with a graded monad

Finnbar Keating
University of Warwick

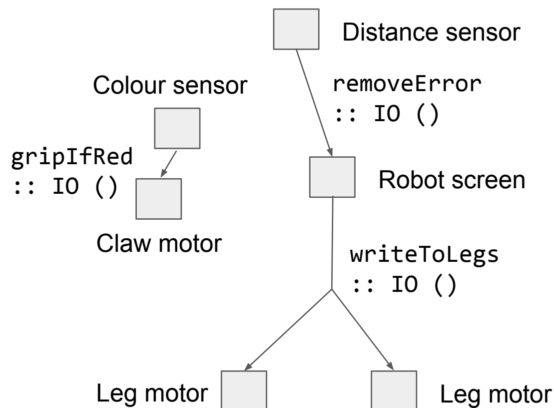
Hi, I'm Finnbar, and today I'm gonna be talking about this scarily-titled thing. We'll talk about what these words mean throughout the presentation, but the key thing to think about for now is that we're going to be looking at programs where functions have dependencies between each other (because an earlier one needs to compute a result used by a later one, for example) and their types don't really reflect that. But first, an example to clarify this...

A robot friend



Here's a robot! We're going to consider its sensors and actuators like memory cells - to get an input from some sensor you read the cell, and then to control some actuator (like motor speed), you write to its cell. We're also going to consider cells which are both read from and written to. In this example, we consider the robot screen to be like this - you can write values to the cell to display them to the screen but also use it as a place to read from to do later calculations. Treat it like a location for storing intermediate values.

A robot friend



- We have dependencies (one computation writes to a memory cell read by another computation) between computations which require correct ordering.
- We could also run parts of the robot in parallel because of a lack of dependencies.
- However, the types give us no information about this!

We then define functions between these to define the workings of our robot. In this example, we have two separate processes going on. We have the left tree, which links our colour sensor to a claw motor - gripIfRed does some calculation using the inputs of the colour sensor to determine whether an object is red and if so sets the motor speed to grip it (reads to the colour sensor, writes to the claw motor - hence the directed edge between the two). Meanwhile, the right tree takes the value of the distance sensor, removes some kind of error from it and writes that value to the robot screen (as an intermediate calculated value), then takes that intermediate value and writes it to the leg motors with writeToLegs.

We can see a few things about this:

- We have a dependency between `removeError` and `writeToLegs` - `writeToLegs` requires the intermediate value (output from `removeError`) to be computed before it can be run.
- We have two separate trees that are unaffected by each other, so we could run them concurrently.
- The types of each function are just `IO ()`, which means the compiler cannot check whether either of the above things have been done safely by the programmer! We could easily run `removeError` after `writeToLegs`, or even run them in parallel so have no guarantee that the inputs will be correct.

In the rest of this talk, we'll be fixing these things via better types.

Talk Structure

- How does Haskell’s IO work?
- How can we improve this using graded monads?
- Let’s use type-level programming to resolve dependencies

This talk is split into three parts.

How does Haskell’s IO work?

For this section we’re gonna briefly look at some Haskell you need to know for understanding the rest. This talk is designed with minimal Haskell knowledge required, but it still requires some, so here goes.

Pure functions in Haskell

- Haskell is based on lambda calculus, so has variables (`n`), abstractions (`\n -> e`) and applications (in this case, infix `+`).

```
plusThree :: Int -> Int
plusThree = \n -> n + 3
```

- The execution order of pure functions doesn't matter.

Haskell is basically very fancy lambda calculus. We have definitions with types (::) and values (=). We have functions with \x -> e.
Notably, the execution order is implicit because with pure functions, you will get the same result whatever the execution order. However, in IO, this isn't the case...

IO in Haskell

- In order to work with things outside of pure functions, we consider IO computations - values of type `IO t` that output a value of type `t` when “run”.
- To build larger IO computations, we combine smaller ones with `>>=`.
- We can also build computations our of pure functions with `return`.

$$return :: a \to IO\ a$$
$$(>>=) :: IO\ a \to (a \to IO\ b) \to IO\ b$$

In general, these sequenceable computations are called **monads**, where you can replace IO with another one of these.

In IO, we’re considering writing to files, reading from memory etc. and as such the order matters. Therefore, we introduce the idea of “IO computations”, which are things that can be run and output a value. We join together smaller computations into larger ones with >>=, and have return (not really used in this talk, mainly there for the full definition).

This generalises to monads, which we’ll generally not talk about.

Working with memory in IO

```
cell1 :: Ptr Int
cell2 :: Ptr Int

peek :: Ptr a -> IO a
poke :: Ptr a -> a -> IO ()

writePlusTwo :: IO ()
writePlusTwo = peek cell1 >>= \inp -> poke cell2 (inp + 2)
                    ↑                        ↖
                 IO Int                Int -> IO ()
```

We could swap the memory cells, or even add more memory operations and **it wouldn't change the type**.

We define two memory cells, cell1 and cell2 (note that the strings "input" and "output" are actually part of the type here - consider it like some kind of identifier of that memory cell. (We'll use this properly later.) We define readCell and writeCell which read an integer from the given memory cell (giving a computation of type `IO Int`) and write an integer to a given memory cell (giving a computation of type `IO ()`, where `()` is the unit type*).

Finally, we define writePlusTwo, which sequences a computation of type `IO Int` (reading cell1) with a function that given the result of the first computation (of type `Int`) returns a computation of type `IO ()` (reading cell2). This has type `IO ()`. Our motivation is as follows: we can see by eye that this function defines dataflow between cell1 and cell2 - the value of cell1 is used to compute the value of cell2. It would be useful for the compiler to have this information, however, it only has the type `IO ()` to work with. We'll motivate this in the next few slides.

* The unit type is the type with only one value, called unit. This is useful for when there's nothing that really needs to be returned, but you need a return value anyway because it's a function.

Two key problems to solve

1. The types of our memory locations do not uniquely identify them.
   - We redefine our pointers with an additional type-level identifier:
     `newtype MemoryCell (s :: Symbol) t = Ptr t`
   - We redefine the memory functions to work with this:
     `readCell :: MemoryCell s t -> IO t`
     `writeCell :: MemoryCell s t -> t -> IO ()`
2. We need `>>=` to track the identifiers read to and written from.
   - “If `cx` reads from x, and `cy` writes to y, `cx >>= cy` reads from x and writes to y”

There are two steps we need to take in order to better track the effects of some computation.

How can we improve this using graded monads?

[Katsuma 2014; Orchard and Petricek, 2014]

This section is basically a short explanation of the first part of the paper Embedding Effect Systems in Haskell (https://kar.kent.ac.uk/57487/1/haskell19f-revised.pdf).

Regardless, in this section we'll introduce the general idea of graded monads and how we're going to apply them to our problem.

The problem with normal monads

In our definition of bind, the monad (m) remains the same. This means we can't differentiate between different uses of it.

$$return :: a \to m\,a$$

$$(>>=) :: m\,a \to (a \to m\,b) \to m\,b$$

"IO that writes to a cell called x"

"IO that reads from a cell called y"

"IO that writes to x and reads from y"

But this doesn't work, since the monad has to stay the same!

Here's the monad definition from before. One way we might go about solving our problem is to change the type of the computations to better reflect what information they use - so have (for some memory cell c, integer n) `readInt c :: IOReads c Int` and `writeInt c n :: IOWrites c Int` or something like that. The problem there is that we can't then sequence these computations, since the type of the computation (the m in the above definitions) must remain the same - so you couldn't write something like

```
readCell c >>= \input -> writeCell c’ input
```
since `IOReads c ~ IOWrites c` is impossible (you can't unify things which are guaranteed to be different)

This is a dead end, then. Fortunately, we have graded monads to solve this problem for us!

Graded Monads

We use a different form of >>= which parameterises the monad!

$$\text{Given monoid}\,(S,\ \cdot,\ I):$$

$$return\ ::\ a\ \rightarrow\ m_I\, a$$

$$(>>=)\ ::\ m_F\, a\ \rightarrow\ (a\ \rightarrow\ m_G\, b)\ \rightarrow\ m_{F \cdot G}\, b$$

"IO that writes to a cell called x"

"IO that reads from a cell called y"

"IO that writes to x and reads from y"

So we just need a monoid that defines this composition.

These graded monads parameterise the type of the computation with a monoid! (Reminder, because I'm very good at forgetting this: a monoid for some set S is a composition operator (.) and identity (I) such that (.) is associative and I is an identity on that operator.) For the purposes of this talk, you can basically think of these monoids as sets of things that our computations do outside of giving back some result.

To this end, we redefine return and (>>=). To bring a pure value into a computation (return), we just parameterise that computation type with the identity, since the computation doesn't do anything outside of returning the pure value. To sequence two computations, we combine their individual parameters to get F.G - the combined computation does both of the things that the individual combinations do, so we combine them with the composition operator.

The Memory Monad

We define a monoid to represent the read and write effects of a given piece of code as a tuple (rs, ws).

For set of memory cells M, we define monoid $(S, \cdot, I)$:
$S = P(M) \times P(M)$
$(rs, ws) \cdot (rs', ws') = (rs \cup rs', ws \cup ws')$
$I = (\emptyset, \emptyset)$

This gives us a replacement to IO in this case: **`Memory (rs, ws) a`**, which represents a computation that reads from all memory cells in rs, writes to all memory cells in ws, and returns a value of type a.

We now present a new graded monad called Memory. It's parameterised by a tuple of sets - the first entry representing the memory cells we read from and the second representing the memory cells we write to. The set that our monoid is defined on (where M is the set of all memory cells), our composition operator and identity are given on the slide.

Applying this to Memory Cells

In our earlier IO example, we had:

```
readCell :: MemoryCell s t -> IO t
writeCell :: MemoryCell s t -> t -> IO ()
```

We can make a few changes to this to apply graded monads with our new Memory monad:

```
readCell :: MemoryCell s t -> Memory ([MemoryCell s t], []) t
writeCell :: MemoryCell s t -> t
            -> Memory ([], [MemoryCell s t]) ()
```

Let's actually apply this to memory cells now! We redefine `readCell` and `writeCell` to return computations in Memory rather than in IO now. readCell now has the effect of reading from the input memory cell, so has the tuple `([cell], [])`. writeCell has the effect of writing to the output memory cell `([], [cell])`.

Note that these [] in the type represent type level lists - we treat them as sets (under the hood, our compose operator sorts the lists and removes the duplicates), so just think of them as sets of types. We only keep track of the identifier (s) of the cell in our sets for clarity. (In the real implementation we track more, but the identifier is sufficient here.)

Applying this to Memory Cells

This means that when we compose writes and reads, the type fully explains what effects a given function has:

```
cell1 :: MemoryCell "input" Int
cell2 :: MemoryCell "output" Int

writePlusTwo :: Memory (["input"], ["output"]) ()
writePlusTwo =
    readCell cell1 >>= \inp -> writeCell cell2 (inp + 2)
```

Memory (["input"], []) Int   Int -> Memory ([], ["output"]) ()

We now see this in action with our example from before. The left half of (>>=) reads from input, and the right half writes to output, so they're combined into the correct type. This means that we have these more specific types now!

Let’s use type-level programming to resolve dependencies

Aim: resolve our dependencies at type-level, and then bring them back to the value level.

Finally, we’re gonna use these more specific types to perform useful analysis of them (looking at the stuff from earlier about functions needing to be run in order), and then talk about how we actually use these in a program.

Working with Types

- We can apply functions (type families) to types in order to perform analyses at compile time.
- We can also have type-level lists and tuples for grouping types together so that relationships between the types can be analysed.

```
type family Dependency x y :: Bool where
     Dependency (Memory ‘(rs, ws) a)
        (Memory ‘(rs’, ws’) b)
         = NonEmptyIntersect ws rs’

mems :: HList ‘[Memory ([“a”], [“b”]) (),
                Memory ([“c”], [“d”]) (),
                Memory ([“b”], [“e”, ”f”]) ()]
mems = removeError :+:
       gripIfRed :+:
       writeToLegs :+: HNil
```

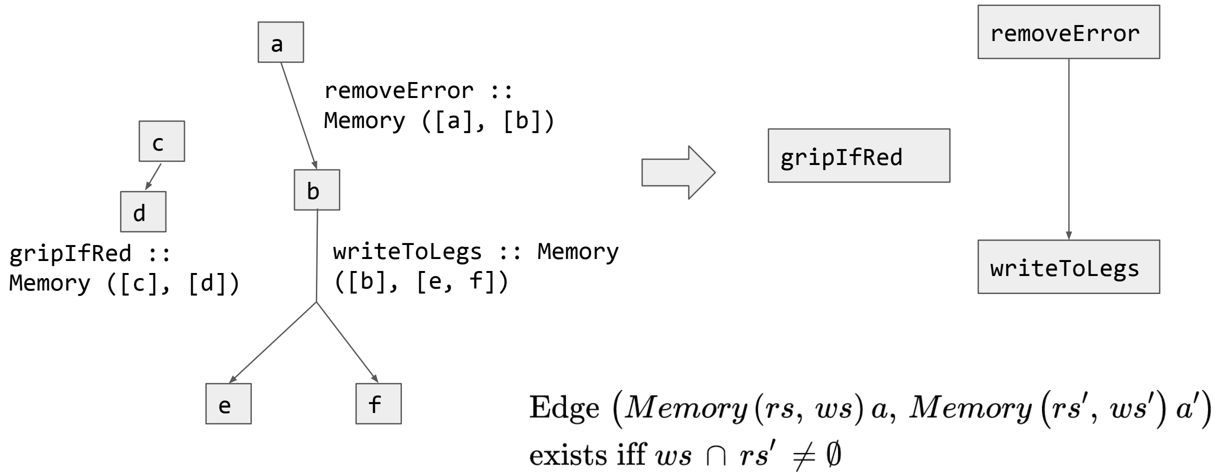
We need to introduce a few quick things about type level programming.

- We can apply functions to types! These are called type families.
- We can have more complex structures of types - we had tuples earlier on with our Memory monad, but we can also have lists.

What can we work out about a list of computations?

- We now have a list of types corresponding to a list of computations that we might want to run!
```
mems :: HList ‘[Memory ([“a”], [“b”]) (), Memory ([“c”], [“d”]) (), Memory ([“b”], [“e”, ”f”]) ()]
```
- We can perform some simple analyses on these.
  - We can check that for possible race conditions by looking at the write sets of the input types.
  - If we know that only some memory cells have been modified, we can determine whether a computation will have any effect (by looking at the read sets).
- We can also look at more complex analyses regarding dependencies between computations.

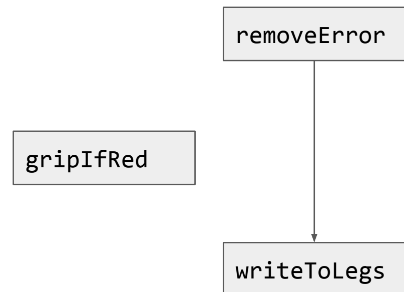
Given mems from the previous slide, what sort of analyses can we do?
We can do some simple analyses that don’t require more complex data structures - such as checking for race conditions and dealing with partial updates.
But also, we can perform more complex analyses by considering the relationships between the types (dependencies).



With this definition, we can use our computations from before and their types to introduce a dependency graph - `removeError` and `writeToLegs` have the cell b shared between them, so there's a dependency.

Working with Dependency Graphs

- Ordering our computations such that no dependencies are violated is just a topological sort!
- Splitting our computations into components that can be run in parallel is just a connected components search!



This dictates components that can be run in parallel (connected components*) and orderings (topological sorts of those components).

* explicitly *not* strongly connected components

The Challenge

- We’ve provided ways to make these analyses happen at the type level (compile time). However, we need a way to make them available at the value level (runtime).
- There are normally two ways of doing this:
  - If the value can be uniquely defined by the type, we can just use that value. (For example, type-level natural numbers have that number as the only value that type can take.)
  - Otherwise, you can write an equivalent value-level algorithm to perform the same process as the type-level algorithm.
- Ideally, we’d like to avoid rewriting all of our analyses.

```
ordered :: xs’ ~ OrderMems xs => HList xs -> HList xs’
ordered xs = ???
```

We have type level functions for doing our analyses! However, we’d like to get the results back to value-level (available past compile time) so that we don’t have to do a topsort again, for example.
Two ways to do this: neither are great here because `Memory (rs, ws) a` doesn’t define a single function. Our type signature there has a constraint - given that some `OrderMems xs` produces `xs’`, the function is of type `HList xs -> HList xs’`.

Rearrangements

- Currently, we have a value-level list of Memory functions with some type xs, and we'd like our list to instead have type xs'.
- Fortunately, since this is a sort, the types contained within xs are the same as within xs' - so we can just lookup values in the input list based on their type and put them in the right place of our output list.

```
ordered :: (xs' ~ OrderMems xs, RearrangeList xs xs') =>
    HList xs -> HList xs'
ordered = rearrange
```

We now have some list of types (our input list), values of those types, and an output type that we'd like the list to have. We're looking for a way to transform the input list to the output list type. Fortunately, because topological sort is a sort, we know that the types contained in the input will be the same as those contained in the output - so we can just take elements from the input list and rearrange them to suit the type of the output list.

We say that `RearrangeList xs xs'` holds iff there is some way to define `rearrange :: HList xs -> HList xs'`. Let's define this for ourselves!

Rearrangements

```
class RearrangeList old new where
    rearrange :: HList old -> HList new

instance RearrangeList '[] '[] where
    rearrange _ = HNil

instance (GetHListElem n old old',
          RearrangeList old' ns)
         => RearrangeList old (n ': ns) where
    rearrange li = elem :+: rearrange li'
        where (elem, li') = getHListElem li
```


This is ugly but I'll try my best to explain it. There are three bits of code here, which essentially illustrate an inductive proof.

- The class defines a typeclass, which is a group of types that we can define these functions for. In this case, it defines `rearrange :: HList old -> HList new`.
- The next statement is our base case. If our two types are the empty list, then we define a trivial rearrangement from the empty `HList` to the empty `HList`.
- The final statement is an inductive case, allowing us to rearrange the old list (of type `old`) into the new list (of type `new`, where the first element is `n` and the rest of the elements are `ns`) if we're able to rearrange a smaller old list (`old` without the element `n`, called `old'`) into a smaller new list (`ns`).

Conclusions

- We talked about a program which does not have enough information in its type in order to reason well about it.
- We added information to the type using a graded monad.
- We then use type-level functions to reason about those types at compile time.
- Finally, we define a simple rearrangement to the correct type for use at runtime!
- Because that rearrangement only relies on the fact that topological sort is a sort, we can use this to run other analyses at compile time and access them at runtime.

So there we go! We've defined a way to express dependencies of computations that read from and write to memory, and then we have a way of sorting it at compile time so that they will be run in an order that preserves those dependencies.

The cool final thing to note is that our rearrangements idea applies to any type-level analysis that's a sort - so if we want to change our analysis to look for other things, we can do so without changing anything at the value level.

Thank you!

Thank you!