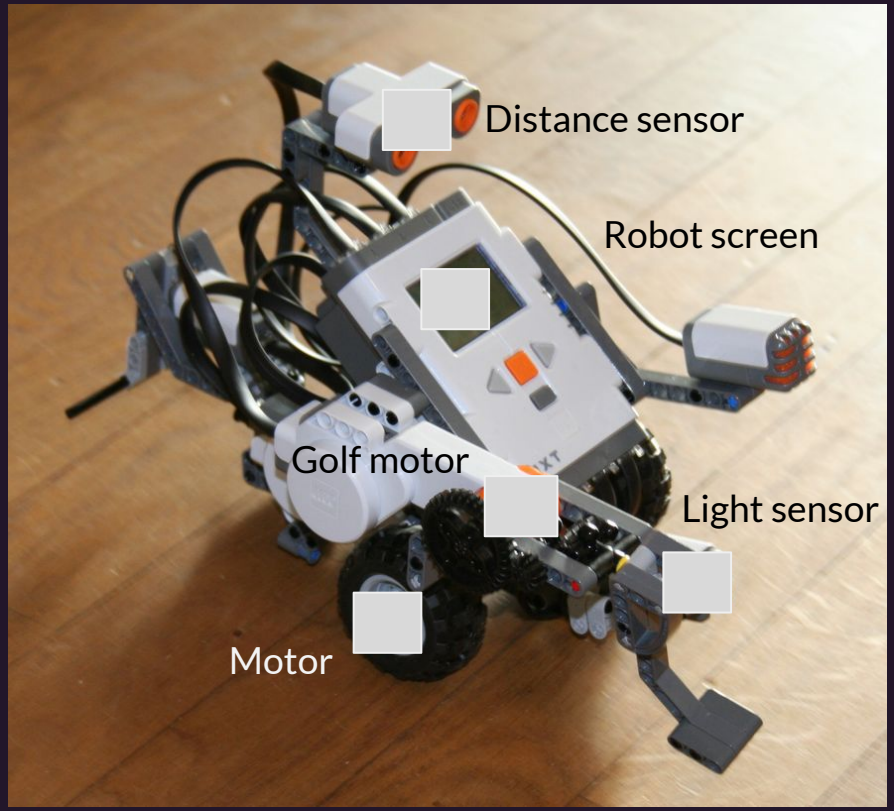
Graded Monads and Type-Level Programming for Dependence Analysis

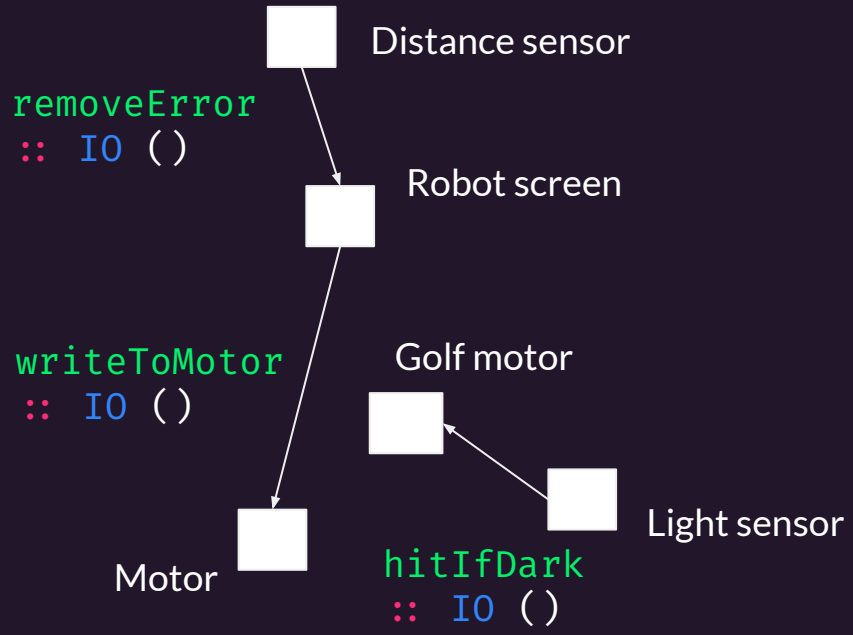
Finnbar Keating
University of Warwick
f.keating@warwick.ac.uk

Michael B. Gale
University of Warwick
m.gale@warwick.ac.uk

A robot friend



A robot friend




```
robot :: IO ()
robot = removeError >>
        writeToMotor >>
        hitIfDark

robot' :: IO ()
robot' = writeToMotor >>
         removeError >>
         hitIfDark
```

We resolve data-flow dependencies in three steps

1. Make computation types more specific.
2. Order computation types to prevent dependency violation.
3. Apply this result to our computations.

1. Make computation types more specific.

Types should encode reads and writes

```haskell
getScreenVal :: IO Int
getScreenVal =
  readIORef screen
```

Should reflect that we read `screen`.

```haskell
putMotor :: Int → IO ()
putMotor = writeIORef motor
```

Should reflect that we wrote to `motor`.

```haskell
writeToMotor :: IO ()
writeToMotor =
  getScreenVal >>= putMotor
```

Should reflect both of the above.

Graded Monads in Haskell [D. Orchard and T. Petricek, 2014]

```haskell
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

```haskell
class GradedMonad (m :: k → * → *) where
  type Unit :: k
  type Plus (f :: k) (g :: k) :: k

  return :: a → m Unit a
  (>>=) :: m f a → (a → m g b)
        → m (Plus f g) b
```

Adds type parameter in terms of a unit type and plus type family (which form a monoid).

Useful in a variety of different contexts.

A graded monad indexed by data-flow dependencies

A parameter representing the memory cells read from and written to

```
newtype MIO (s :: ([*], [*])) a =
    MIO { runMemory :: Set (MemoryUnion s) → IO a }
```

Value-level representations of the cells being used by this computation

This is a graded monad!

```
instance GradedMonad MIO where
    type Unit = '( '[], '[])
    type Plus '(rs, ws) '(rs', ws') =
        '(Union rs rs', Union ws ws')
```

Pair of empty sets

Elementwise union

Sequencing two computations will have the combined reads and writes of both, since we take the union.

We name memory locations

```
data IOCell (s :: Symbol) t where
    IOCell :: IORef t → IOCell s t

input :: IORef Int

inpCell :: IOCell "input" Int
inpCell = IOCell @"input" input

readIOCell :: IOCell s t
           → MIO '( '[IOCell s t], '[]) t
writeIOCell :: IOCell s t → t
            → MIO '( '[], '[IOCell s t]) ()
```

We construct `IOCells`, which give an `IORef` a type-level name.

These allow us to define read and write operations that track what reads and writes have been done.

Rewriting our initial example

```
getScreenVal :: MIO '( '[IOCell "screen" Int], '[]) Int
getScreenVal = readIOCell @"screen"

putMotor :: Int → MIO '( '[], '[IOCell "motor" Int]) ()
putMotor = writeIOCell @"motor"

writeToMotor :: MIO '( '[IOCell "screen" Int]
                     , '[IOCell "motor" Int]) ()
writeToMotor = getScreenVal >>= putMotor
```

2. Order computation types to prevent dependency violation.

We don’t want to worry about manually ordering our computations

```
robot = removeError :+:
  hitIfDark :+:
  writeToMotor :+: HNil
```


We combine these computations **arbitrarily** using a `HList`.

```
data HList :: [*] → * where
    HNil :: HList '[]
    (:+:) :: x → HList xs → HList (x ': xs)
```

We don’t want to worry about manually ordering our computations

```
robot :: HList ‘[
  MIO (‘[“dist”], ‘[“screen”]) (),
  MIO (‘[“light”], ‘[“golf”]) (),
  MIO (‘[“screen”], ‘[“motor”]) ()]

robot = removeError :+:
  hitIfDark :+:
  writeToMotor :+: HNil
```

Using this type information…

… find a valid ordering for these that does not violate any dependencies.

We need to determine the order of computations

We focus on *data dependencies* [Bernstein 1966] - e.g. `writeToLegs` has a dependency on `removeError` since the latter writes to a cell used by the former.

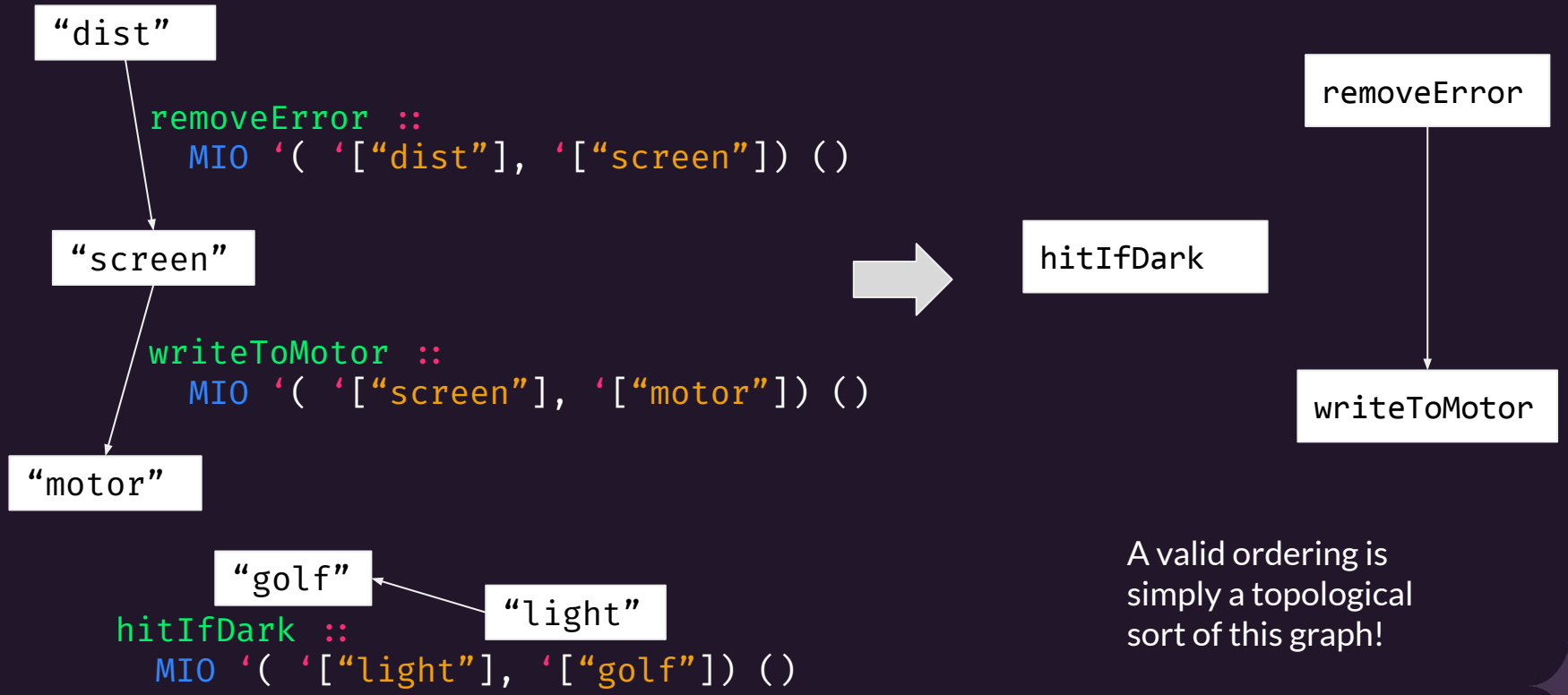
```
removeError :: MIO ('["dist"], '["screen"]) ()
writeToLegs :: MIO ('["screen"], '["motor"]) ()

type family IsLessThan x y where
  IsLessThan (MIO '(rs, ws) a) (MIO '(rs', ws') a) =
    NonEmptyIntersect ws rs'

IsLessThan (MIO ('["dist"], '["screen"]) ())
  (MIO ('["screen"], '["motor"]) ()) ~ 'True
```

We have a type family to determine whether one computation depends on another.

Dependency Graphs



A valid ordering is simply a topological sort of this graph!

Representing a type-level graph

We represent a type-level graph as an adjacency list, mapping types (nodes) to in- and out- edges.

```
type AdjacencyList = [Mapping * ([*], [*])]
```

We also have type families like `GetInEdges` and `GetOutEdges` that get the edges from a given node.

First-class type families [`first-class-families`, L. Xia]

- Partially applied type families are simulated through data constructors with defined `Eval`.
- `Exp` is the return type from these type families.
- `Exp` is a type-level monad (with `=<<` and `Pure` specialised to `Exp`).

```
data NonEmptyIntersect' :: [k] → [k] → Exp Bool
type instance Eval (NonEmptyIntersect' xs ys) =
  Eval (Not =<< Null =<< Filter (Flip Elem xs) ys)
```

`Filter` gives us `Exp [k]`, so we can bind to pass the result to `Null :: [k] → Exp Bool`.

Building our adjacency list

```
type Comp = * → * → Exp Bool
type AdjacencyList = [Mapping * ([*], [*])]

data ToAdjacencyList :: Comp → [*] →
  Exp AdjacencyList
type instance Eval (ToAdjacencyList cmp nodes) =
  Eval (Map (GetAdjacent cmp nodes) nodes)

data GetAdjacent :: Comp → [*] → * →
  Exp (Mapping * ([*], [*]))
type instance Eval (GetAdjacent cmp nodes node) =
  node :→
    '(Eval (Filter (cmp node) nodes),
      Eval (Filter ((Flip cmp) node) nodes))
```

Apply `GetAdjacent` to each node in nodes…

…which applies a comparator from the given node to every node in the graph to find in- and out-edges.

Depth-first search on a type-level adjacency list

```
type Acc = ([*], [*])
type SearchFn = AdjacencyList → * → Exp [*]

data DFS :: SearchFn → AdjacencyList → * → Acc → Exp Acc
type instance Eval (DFS search adj node '(stack, used)) =
  Eval (UnBool
        (UpdateStack node
         =<< Foldr (DFS search adj) '(stack, node ': used)
         =<< search adj node)
        (Pure '(stack, used))
        (Eval (Elem node used)))

data UpdateStack :: * → Acc → Exp Acc
type instance Eval (UpdateStack node '(stack , used)) =
  '(node ': stack, used)
```

… and update the stack of nodes to include the one we just explored.

… then apply DFS to each of those…

… then apply our search strategy to get a list of nodes…

If we've not encountered this node yet…

Useful graph algorithms are applications of DFS

- Topological sort is just applications of DFS `GetOutEdges`.
- Connected components search is just applications of DFS `GetEdges`.
- DFS on the graph transpose (for Kosaraju's Strongly Connected Components search) is just DFS `GetInEdges`.

One Step Closer...

Builds a graph with `IsLessThan` and then topologically sorts it.

```
ordered :: xs' ~ Eval (Ordered IsLessThan xs)
        ⇒ HList xs → HList xs'
ordered xs = ???

sortedRobot = ordered robot
```

How do we bring the result of our topological sort to the value level?

3. Apply this result to our computations.

Reifying usually happens in two ways

Two possible approaches when reifying:

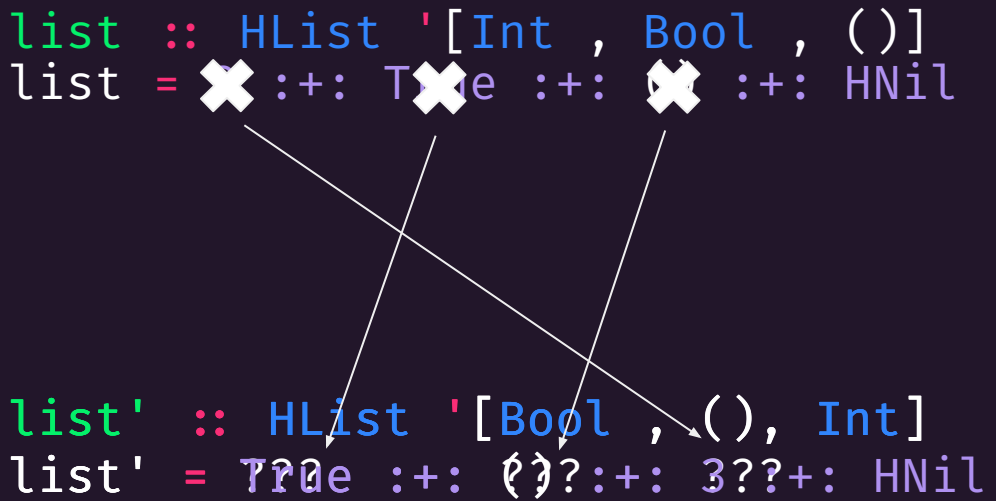
- If a type has a single associated value, just use that value. A computation of type `MIO '(rs, ws) a` has many possible values however.
- Reimplement the type-level algorithm at the value level.

We propose a middle ground - *rearrangements* - which uses the values in the input list as a lookup for values in the output list.

Using the input list as a lookup

- We want a generic way to permute an input into an output.
- As such, we assume that the values of elements of the output HList are the same values in the input list, just reordered.
- To reify an element of the output list, look up the first unused value of its type in the input list.

An example of rearrangements



Our RearrangeDel class

```
class RearrangeDel env target env' | env target → env' where
  rDel :: HList env → (HList target , HList env')
```

Rearranges an input list to the target list and any unused elements from the input.

Success!

```
ordered :: (RearrangeDel xs xs' '[],
  xs' ~ Eval (Ordered IsLessThan xs)) ⇒
  HList xs → HList xs'
ordered = rDel

sortedRobot = ordered robot
```

Conclusions

- We use a graded monad to make the types of computations that mutate state more precise.
- We then use type-level programming to look for dependencies between those computations.
- We finally use *rearrangements* to bring the results of any type-level computation that permutes its input to the value level.
- With this, we guarantee that our computations run in an order that satisfies those dependencies.

Thank you!

Implementation of RearrangeDel

```
class GetHListElem x inp out | x inp → out where
  getHListElem :: HList inp → (x, HList out)

instance (GetHListElem x env env',
  RearrangeDel env' target' env'') ⇒
  RearrangeDel env (x ': target') env'' where
    rDel l = (x :+: xs, l'')
      where (x, l') = getHListElem l
            (xs, l'') = rDel l'
```

If you can find an element of type x in env, leaving env' over...

and you can rearrange env' to some target', leaving env'' over...

then you can rearrange env to (x : target'), leaving env'' over...

by finding the element of type x, prepending it to the result and recursively rearranging the remainder of the list!