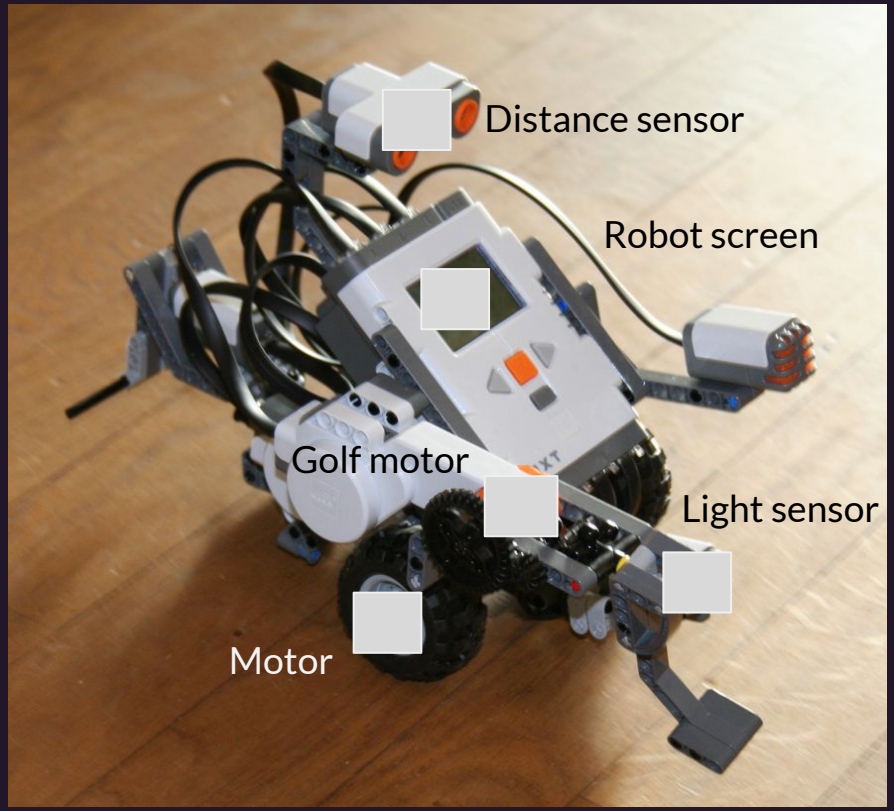
Graded Monads and Type-Level Programming for Dependence Analysis

Finnbar Keating
f.keating@warwick.ac.uk

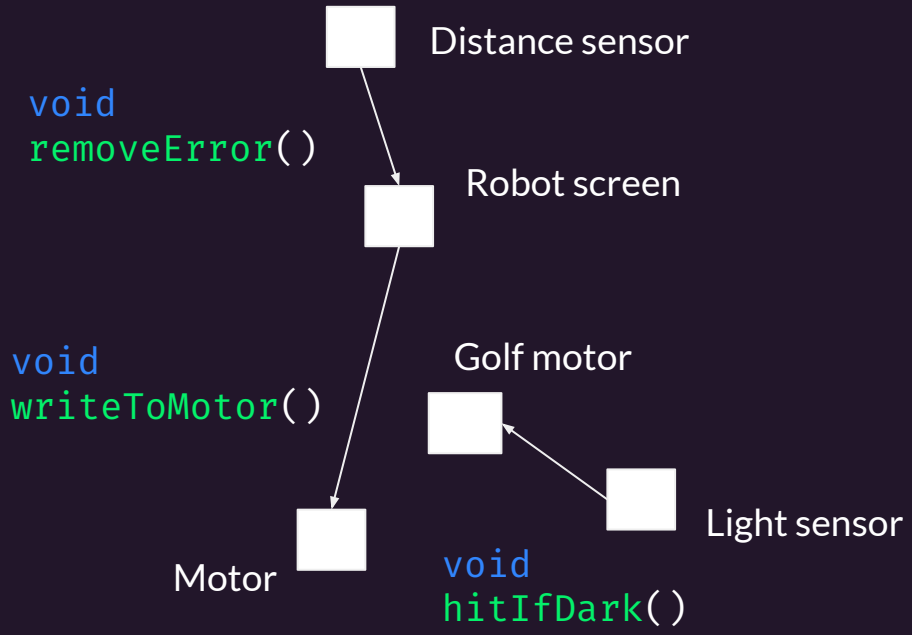
WPCCS'21

Joint work with Michael B. Gale (michael.gale@tweag.io)
Work presented in the Haskell Symposium 2021

A robot friend



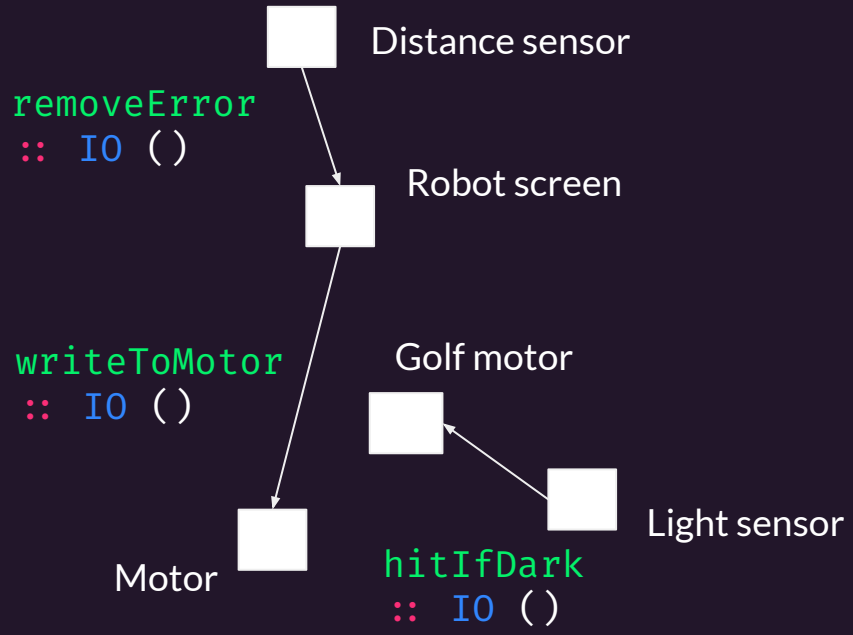
A robot friend




```
void robot() {
    removeError();
    writeToMotor();
    hitIfDark();
}

void robot2() {
    writeToMotor();
    removeError();
    hitIfDark();
}
```

A robot friend



Performs input/output (like writing to memory) and returns a success value.

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

robot2 :: IO ()
robot2 = writeToMotor >>
         removeError >>
         hitIfDark
```

Sequences two IO computations, ignoring their results.

We resolve data-flow dependencies in two steps

1. Make computation types more specific.
2. Order computations based on their types to prevent dependency violation.

1. Make computation types more specific.

Working with Memory in Haskell

We have seen `IO t` used to represent computations that return a value of type `t`, after performing some IO. An example of this is reading and writing to memory:

```
readIORef :: IORef t → IO t
writeIORef :: IORef t → t → IO ()
```

These can be sequenced using bind:

```
(>>=) :: IO a → (a → IO b) → IO b
```

This idea generalises to *monads*, which are operations that can be sequenced.

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]

```
(>>=) :: IO a → (a → IO b) → IO b
```

```
(>>=) :: MIO f a → (a → MIO g b) → MIO (Plus f g) b
```

Adds type parameter which describes what happens within a computation.

We are able to combine these parameters with Plus. (This is a *type family*.)

A graded monad for memory operations

We introduce `IOCell` - memory locations with name `s` which contain an element of type `t`.

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

```
type Plus '(rs, ws) '(rs', ws') =
  '(Union rs rs', Union ws ws')
```

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

Rewriting our initial example

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

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

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

2. Order computations based on their types to prevent dependency violation.

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.

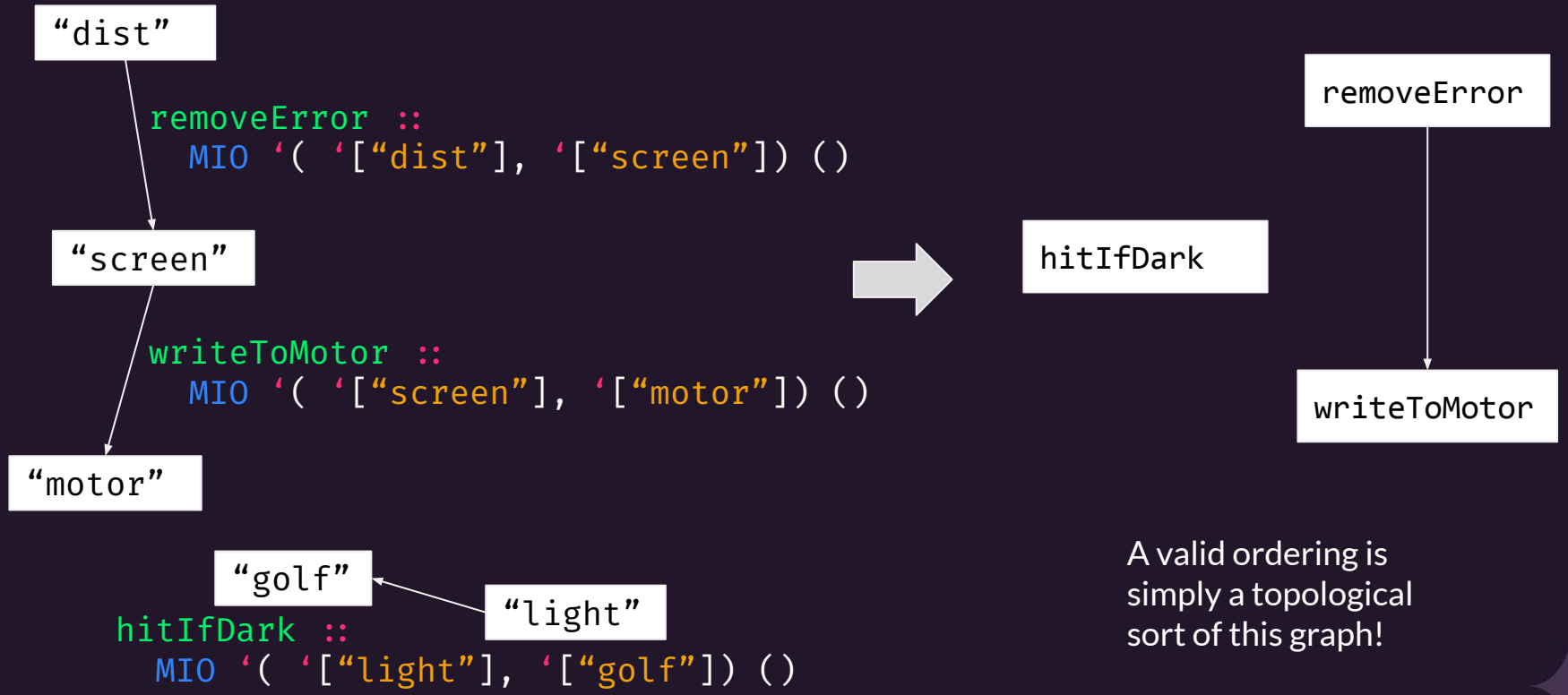
We need to determine the order of computations

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"]) ()
```

This is implemented as a type family that checks whether a cell that is written to by the first computation is read by the second.

Dependency Graphs



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

Representing a type-level graph

This dependency graph is represented as an adjacency list.

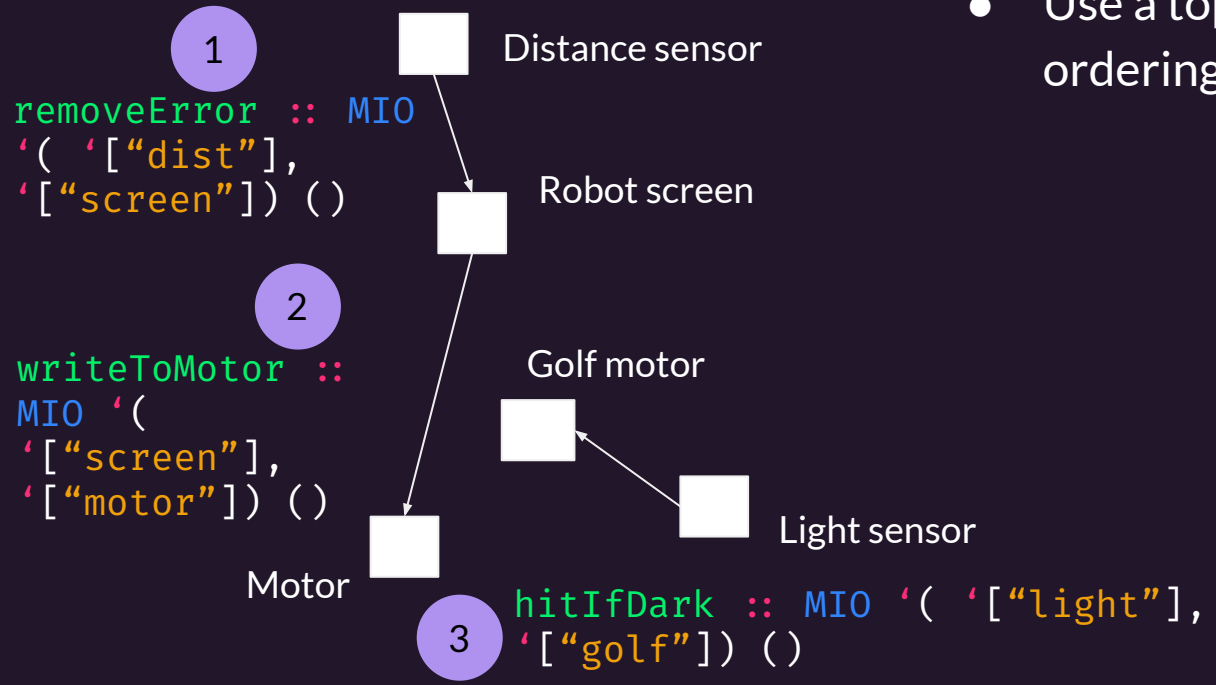
```
'[
  MIO `( `["dist"], `["screen"]) () :→
    `[MIO `( `["screen"], `["motor"]) ()],
  MIO `( `["screen"], `["motor"]) () :→ `[],
  MIO `( `["light"], `["golf"]) () :→ `[]
]
```

Computations of this type…

… must be run before computations of this type.

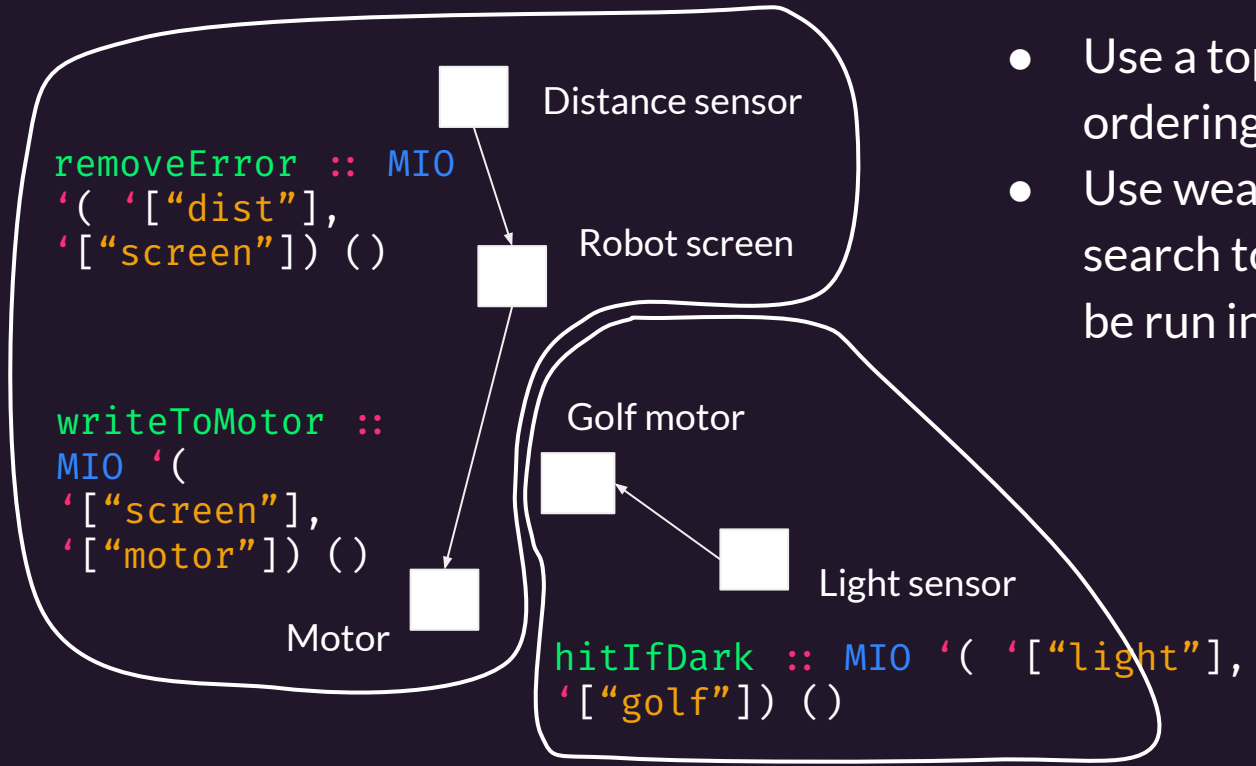
So we can write type families on these graphs!

With this dependency graph, we can...



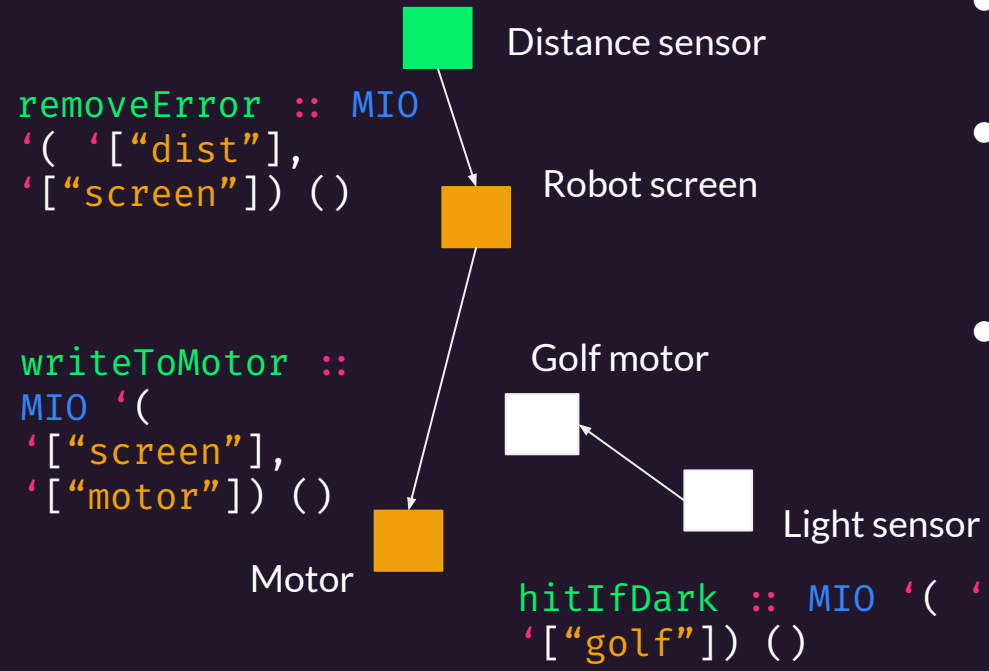
- Use a topological sort to find a valid ordering

With this dependency graph, we can...



- Use a topological sort to find a valid ordering
- Use weak connected components search to find computations that can be run in parallel

With this dependency graph, we can...



- Use a topological sort to find a valid ordering
- Use weak connected components search to find computations that can be run in parallel
- Determine which computations to run if you know which memory cells have been changed

Conclusions

- We use a graded monad to make the types of computations that mutate state more precise.
- We then use type families to look for dependencies between those computations, build a dependency graph and order the computations at compile time.
- This is all implemented in Haskell and ensures that our computations run in an order that satisfies their dependencies without ordering effort from the programmer.

Thank you!