Graded Monads and Type-Level Programming for Dependence Analysis

Finnbar Keating
F.Keating@warwick.ac.uk
University of Warwick
Coventry, United Kingdom

Michael B. Gale
M.Gale@warwick.ac.uk
University of Warwick
Coventry, United Kingdom

Abstract

Programmers make assumptions about the order of memory operations, which are not captured in the operations' types and therefore cannot be enforced statically by a compiler. This can lead programmers to accidentally violate those assumptions if they are not careful. To address this issue, we encode the memory locations that are accessed by a given computation using a graded monad. We use the data flow dependencies which arise from this to construct a type-level graph that we analyse to automatically order the computations so that no dependencies are violated. This also allows for computations which have no dependencies on each other to be run concurrently.

CCS Concepts: • **Software and its engineering** → **Functional languages**; • **Theory of computation** → *Type structures*; *Program analysis.*

Keywords: dependence analysis, graded monads, type-level programming, concurrency

ACM Reference Format:
Finnbar Keating and Michael B. Gale. 2021. Graded Monads and Type-Level Programming for Dependence Analysis. In *Proceedings of the 14th ACM SIGPLAN International Haskell Symposium (Haskell '21), August 26–27, 2021, Virtual, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3471874.3472981

1 Introduction

When working with mutable state, programmers may make assumptions about the order in which operations on the same memory locations are performed, such as writing to them before reading from them. Such data dependences are commonly considered for the purpose of program optimisation as part of dependence analysis [1], but may also affect correctness. For example, consider the following Haskell program:

```
getDist :: IO ()
getDist = do x <- readMem distanceSensor
             writeMem distVal x

writeMotor :: IO ()
writeMotor = do x <- readMem distVal
                writeMem motor (-x)

robot :: IO ()
robot = getDist >> writeMotor
```

This simple robot reads a value from a distance sensor, represented by a memory location named `distanceSensor`. The value is then written to some intermediate memory location named `distVal`. Finally, the value is read from `distVal` and written to `motor`, a memory location representing the speed of the robot's motor. The `robot` computation is executed at a fixed time interval, *i.e.* within some loop that is external to `robot`. The interval at which `robot` is called is arbitrary. Therefore, it is important for the current distance to be written to `distVal` before it is read to update the motor's speed. Although storing the distance value in `distVal` is not necessary for the example above, it serves as a placeholder for arbitrary, intermediate computations.

However, if a programmer was to accidentally implement `robot` as follows, the program would still be well typed even though the assumption about the order of operations is violated:

```
robot :: IO ()
robot = writeMotor >> getDist
```

Such assumptions about the order of memory operations may arise in any program that involves mutable state and a main program loop, such as in updating graphical user interfaces and reactive systems such as the robot shown in our example. To prevent violations of these assumptions about the order of operations in a given part of program, programmers must be able to formally express them. In particular, the process we demonstrate in this paper is:

Haskell '21, August 26–27, 2021, Virtual, Republic of Korea

ACM ISBN 978-1-4503-8615-9/21/08…$15.00
https://doi.org/10.1145/3471874.3472981

- Individual computations such as `getDist` and `writeMotor` are typed so that they reflect which memory locations are read from and written to. In other words, their types reflect their data dependencies.
- Given a list of computations, we statically identify a topological ordering based on their data dependencies so that memory locations are written to, before they are read from. This means that instead of sequencing computations with >> or similar, we provide a list of them to be automatically ordered.
- Based on the results of the above analysis, the individual computations are then automatically composed so that no data dependencies are violated.

Concretely, our contributions are organised as follows. Our implementation of all this can be found as part of our *rearrange* library[1]:

- We introduce a technique to build type-level graphs out of a type-level list of nodes and a type family which defines edges between those nodes. We then show type-level implementations of topological sort, and strongly and weakly connected components search (Section 3).
- We then present a generalised approach for the reification of results from our type-level graph algorithms to value-level data structures. This approach is also applicable to other type-level algorithms (Section 4).
- By using a graded monad indexed by data dependencies to type individual computations, we construct type-level graphs of those dependencies. This allows us to automatically order computations accordingly (Section 5.3). Computations which have no dependencies on each other may be run concurrently (Section 5.6).
- Since we have type-level information about data dependencies, we are also able to perform *partial updates* of minimal, connected components at runtime when a full update of all memory locations is not required (Section 5.5).

2 A Graded Monad for Memory Operations

In order to find data dependencies between computations, we need to know what reads and writes individual computations perform. In this section, we define a graded monad for tracking these reads and writes at the type level.

2.1 Graded Monads

To index computations by the reads and writes they perform, we use *graded monads*[2] [11]. These are monads which are parameterised by a *grade* – a monoid that determines what values the grade can take and how to compose those values.

We can represent graded monads in Haskell due to the embedding provided by Orchard and Petricek [15]. This utilises two types, `Unit` and `Plus`, which represent the empty value of our monoid and composition in said monoid respectively. The definition of the two main monad operations in this embedding are as follows:

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

This graded monad embedding is more expressive than what is supported by the default `Monad` type class in Haskell because of the additional parameter. In the default, `>>=` is of type `m a -> (a -> m b) -> m b` which means that the monad `m` has to be the same, while with graded monads we have an additional parameter that varies.

2.2 Representing Memory Operations at the Type-Level

To make use of graded monads, we first need to define the `Unit` and `Plus` types that represent the monoid defining our grade. To do this, we use type-level sets[3] [15] – a type-level data structure which indexes sets with the types of their values and provides operations such as set union via `Union`. These are represented as a list of types contained in the set along with constraints to sort and remove duplicates from said list. To construct these at the value level, constructors `Empty` and `Ext x xs` represent the empty set and the addition of some element `x` to a set `xs` respectively.

For tracking reads and writes, we define the grade of our graded monad as a pair of type-level sets (rs, ws) – rs contains cells read from by a given computation, and ws contains cells written to. We use type-level sets instead of lists, because our goal is to automatically compose computations according to the results of our dependence analysis. We view computations as the smallest unit of operations that we consider as part of that process. Programmers always have the option to compose computations by hand if the order does not matter.

This pair of sets definition gives us a monoid for composing reads and writes: element-wise union of the tuples of the two computations. This means that a computation represented by (rs, ws) composed with a computation represented by (rs', ws') are represented together as $(rs \cup rs', ws \cup ws')$ – read from both rs and rs', and write from both ws and ws'. This works even when working with two computations that interact with the same memory cell, since the union of two sets both containing some memory cell will contain that memory cell.

[1] https://github.com/finnbar/rearrange

[2] These were originally introduced as *parametric effect monads*, but gained their new name in later work.

[3] https://hackage.haskell.org/package/type-level-sets

This gives us types for the monoid as follows. Note that `Union` takes the union of two sets at type level, sorting and removing duplicates from their list representations in the process.

```
type Unit = '( '[], '[])
type Plus '(rs, ws) '(rs', ws') =
  '(Union rs rs', Union ws ws')
```

With this in place, we now need some way of representing the memory being read from and written to at the type level, as to populate our sets. Our implementation supports a number of different types of memory locations, such as `IORef`, `Ptr` and `STRef`. For simplicity we first introduce a definition specialised to `IORef`, called `IOCell`, before generalising to other memory location types.

```
data IOCell (s :: Symbol) t where
  Cell :: IORef t -> IOCell s t
```

`s` is a phantom type parameter [12] representing the label of a mutable variable, allowing each variable to be represented uniquely at type-level by a unique label. This means that our `IOCell`s must be given an explicit type-level string to identify them, for example `IOCell @"name" ptr` using GHC's TypeApplications extension.

We now provide an alternative, generalised `Cell` that works with other mutable variable types.

```
data Cell (v :: * -> *) (s :: Symbol) t where
  Cell :: (Monad m, MonadRW m v, Constr m v t)
    => v t -> Cell v s t

type IOCell s t = Cell IORef s t
```

This definition adds `v`, the type of the mutable variable that the `Cell` contains. It also adds three new constraints, which we explain briefly:

- The type class constraint `MonadRW m v` means that values of type `(v t)` can be read from and written to in the monad `m`. An example of this is `MonadRW IO IORef`. That is, `IORef`s can be worked with in `IO`.
- `Constr m v t` defines a constraint on which types `t` can be used in `(v t)`. When working with foreign memory we can only store certain types of data, so require the `Storable` constraint – hence `Constr IO Ptr t = Storable t`[4]. For other types, `Constr m v t = ()`.

2.3 Reads and Writes with `Cell`

We now present primitive functions for reading and writing to locations represented by `Cell`. We introduce a graded monad, `Memory`, that utilises the `Unit` and `Plus` defined previously to compose the reads and writes done by the composed computations. This graded monad uses a definition similar to Ivašković and Mycroft's `Sync` monad [9], except with sets of mutable memory cells rather than a list of locks.

We first define the `Memory` type, along with a shorthand `MIO` for when working in `IO`.

```
newtype Memory (m :: * -> *) (s :: ([*], [*])) a =
  Mem { runMemory :: Set (MemoryUnion s) -> m a }

type MIO s a = Memory IO s a
```

`Memory` consists of a monad `m` to run the reads and writes in, a grade `s` identical to the one in the previous section and `a`, the return type of the computation. `runMemory` runs a computation that reads and writes – it takes in the union of the cells read from and written to as defined by `MemoryUnion`, and returns a computation that performs the reads and writes.

```
import Prelude qualified as P

return :: a -> Memory m Unit a
return x = Mem $ \Empty -> P.return x
```

Now we define `Memory` as a graded monad by providing `return` and `(>>=)`. `return` does not read nor write, so the input is the empty set as represented by `Empty` and the output is the value `x` returned.

```
(>>=) :: Memory m x a -> (a -> Memory m x' b)
  -> Memory m (Plus x x') b
(Mem e) >>= k = Mem $ \fg ->
  let (f, g) = split fg
  in e f P.>>=
    \x -> (runMemory . k) x g
```

`(>>=)` uses `split` to split the combined environment `fg` into the environments needed for `Mem e` and the result of `k`. The computations given by `runMemory` are then sequenced like non-graded monads using `P.>>=`.

We now define read and write functions within our `Memory` graded monad. These definitions utilise `readVar` and `writeVar` from `MonadRW`, which perform reads and writes respectively to a `Cell`.

```
readCell :: (MonadRW m v, Constr m v t) =>
  Memory m '( '[Cell v s t], '[]) t
readCell = Mem $ \(Ext (Cell pt) Empty)
  -> readVar pt

writeCell :: (MonadRW m w, Constr m v t) =>
  t -> Memory m '( '[], '[Cell v s t]) ()
writeCell x = Mem $ \(Ext (Cell pt) Empty)
  -> writeVar pt x
```

Each of these definitions use `Ext x y` to get the first element of the set `x` and the rest of the set `y`. Since both definitions

[4] This is why we define `MonadRW` rather than using the existing `MonadRead` and `MonadWrite` from `monad-var` (https://hackage.haskell.org/package/monad-var) – we need to set arbitrary constraints on our reads and writes.

contain only one `Cell` in their grade, the `Cell` that we want to read or write must be the first one present in the set.

With reads and writes now defined for `Memory`, we can build up more precise types by binding operations together. These types now reflect all of the reads and writes performed when running a given computation. We can rewrite the examples from the introduction using `Memory` as follows. Note that we use TypeApplications to set the type `s` for each memory operation.

```
getDist :: MIO '( '[Cell Ptr "dist" CInt],
  '[Cell IORef "distVal" CInt]) ()
getDist = do x <- readCell @"dist"
  writeCell @"distVal" (normalise x)

writeMotor :: MIO '(
  '[Cell IORef "distVal" CInt],
  '[Cell Ptr "motor" CInt]) ()
writeMotor = do x <- readCell @"distVal"
  writeCell @"motor" (-x)
```

2.4 Collections of Memory Computations via HList

While these more precise types allow for better understanding of reads and writes done by computations, they pose problems when wanting to work with collections of computations – which we need since we aim to provide a list of computations to automatically order. We can no longer use a homogeneous list such as `[IO ()]` due to the different types introduced by differing grades. Therefore, in order to group computations together we need a heterogeneous list.

HList [13] is a common interface for such lists. The implementation we use is defined below along with a rewriting of `robot` from the introduction as a list of computations that can be reordered rather than a sequence of computations in a fixed order.

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

robot :: HList '[Memory ..., Memory ...]
robot = getDist :+: writeMotor :+: HNil
```

3 Type-Level Graph Analyses

In the previous section, we introduced data dependency types for showing exactly what memory a given computation reads from and writes to, and indexed a graded monad with these, allowing us to compose those computations. This graded monad tracks the data dependencies of composed computations and our next step is to analyse these to build a dependency graph and sort its nodes. We begin by implementing a type-level graph structure.

We then implement topological sort, strongly connected components search and weakly connected components search on that graph structure. Since Haskell is not yet a fully dependently typed language, value-level functions and type-level functions must be defined separately. Therefore, even though there are well established term-level graph libraries in Haskell, we must implement all this from scratch at the type-level.

3.1 First-Class Families

Closed type families [6] provide programmers with the ability to perform computations on types. However, these type families are not first-class so lack the ability to be specialised via partial application and thus reused. This means any common code reuse patterns like higher-order functions are impossible when working with type families – thus requiring a specialised version of *e.g.* map to be written for every type family we want to partially apply this to. This prevents many useful forms of code reuse, thus making the development of larger type families more complex.

Fortunately, Xia's first-class type families[5] package allows us to *simulate* first-class type families and thus allows for partial application of them. These families are defined as data constructors which return types of the form `Exp k :: *`, as to avoid the restriction that data constructors can only result in types of kind `*`. The equations of each type family are then given as instances of the `Eval` type, which are used to run a first-class type family.

An example of this is given below, with two implementations of type-level `elem`.

```
type family Elem (x :: k) (xs :: [k])
  :: Bool where
  Elem x '[] = False
  Elem x (x ': xs) = True
  Elem y (x ': xs) = Elem y xs

data Elem :: k -> [k] -> Exp Bool
type instance Eval (Elem e '[]) = 'False
type instance Eval (Elem e (x ': xs)) =
  Eval (UnBool (Elem e xs) (Pure True)
    (Eval (TyEq e x)))
```

Two differences should be noted between closed and first-class type families. Since first-class type families use type instances, we cannot define an ordering on cases like in closed type families. As such, we need to check whether the input types are equal in the first-class `Elem`. This uses `UnBool`[6] – a type-level if statement which uses the third argument as a branching condition, returning the second argument if it

[5] https://hackage.haskell.org/package/first-class-families
[6] From `Fcf.Data.Bool`: https://hackage.haskell.org/package/first-class-families-0.8.0.1/docs/Fcf-Data-Bool.html

is true and the first argument otherwise. `UnBool` requires that its two branches are `Expr`, so `Pure` is used to make `True` into an `Expr`.

To call another first-class type family, we must use `Eval` to retrieve values from the `Exp` returned by the called family, such as in the `Eval (Elem x ys)` call. To avoid having to do this with every call, first-class families provides us with `(=<<) :: (a -> Exp b) -> Exp a -> Exp b`, which allows `Exp` to work like a monad and thus easily compose first-class type families.

We now look at a brief example to show the strengths of these families – the `NonEmptyIntersect` type family, which finds the intersection of two lists and returns whether that intersection is non-empty.

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

In closed type families, we would have to implement a version of `Filter` specialised to `Flip Elem`, which would need a specialised implementation of `Flip`. We avoid this entirely here, however, due to the partial application allowed by first-class families. We also use `=<<` to pass the result from one family to another.

In the rest of the paper, we use these first-class families to implement our type-level computations.

3.2 Type-Level Graph Analyses

3.2.1 Type-Level Adjacency List. To perform type-level graph analyses, we first need to build a type-level graph. The nodes of our graph are given by an input list of types, and the edges are defined by some type family `Comp :: * -> * -> Exp Bool` run on each pair of nodes – if `Comp a b` returns `True`, then there is a directed edge from `a` to `b`.

We represent this graph as an adjacency list, which is implemented below as a type-level mapping[7] from nodes to a tuple of their out- and in-edges using `node :-> '(out, in)`.

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

data ToAdjacencyList :: Comp -> [*] -> Exp
    AdjacencyList
type instance Eval (ToAdjacencyList comp nodes) =
  Eval (Map (GetAdjacent comp nodes) nodes)

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

[7] Data.Type.Map from https://hackage.haskell.org/package/type-level-sets.

3.2.2 DFS. We can now implement a variety of graph algorithms more easily using our type-level adjacency list. We start with an implementation of DFS, which is used to implement our later graph algorithms. The implementation of DFS is given in full below.

```
type Acc = ([*], [*]) -- (stack, used nodes)
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)
```

The `search :: SearchFn` argument of DFS is a type family that dictates which nodes are expanded next, given the graph and some node – for example, this could return only nodes connected via out-edges, or even both in- and out-edges.

Otherwise, the implementation is as expected for DFS. First, we check if the node is in `used` using `UnBool`. If the node has not been used, then we use `search` to get the nodes we should expand with DFS. Then we add the input node to `used` and apply DFS to each node that we are expanding, using `Foldr` to keep track of the updated `stack` and `used` values from each DFS call. Finally, we add the input node to `stack` with `UpdateStack` and return the resulting `stack` and `used` values.

3.2.3 Topological Sort and Strongly Connected Components Search. Given a graph, we would like to find a topological sort of it and to make sure that no loops are present. To do this, we use topological sort alongside strongly connected component search to check for any loops and error if they are present.

Our existing DFS implementation means that we can express it using first-class type families. Our definition is split into three parts, as exemplified by the `RunTopsort` definition below – topological sort, strongly connected components search, and then verifying that no loops are present.

```
data RunTopsort :: AdjacencyList -> Exp [*]
type instance Eval (RunTopsort adj) =
  Eval (FlattenSingletons
    =<< SCCsFromTopsorted adj
    =<< DoTopsort adj)
```

First, we perform a topological sort. Topological sort is implemented by applying DFS by expanding out-edges and building a stack of nodes via postorder traversal [20]. Since our earlier `DFS` implementation already builds a stack via postorder traversal, the implementation of `DoTopsort` is simple – apply `DFS GetOutEdges` to every node in the graph, which expands every out-edge and thus builds up a topological ordering.

```
data DoTopsort :: AdjacencyList -> Exp [*]
type instance Eval (DoTopsort adj) =
  Eval (Fst =<< Foldr (DFS GetOutEdges adj)
    '( '[], '[]) (Eval (Nodes adj)))
```

We next find the strongly connected components of the graph. First, we consider a generic connected component search, where we use DFS to find all nodes reachable from a given node to form a component. The implementation below takes a search function as input to dictate how the internal DFS expands nodes.

```
type Acc' = ([[*]], [*])

data GetComponents :: SearchFn -> AdjacencyList ->
  [*] -> Exp [[*]]
type instance Eval (GetComponents srch adj ns) =
  Eval (Fst =<< Foldl (AddToComponent srch adj)
    EmptyAcc' ns)

data AddToComponent :: SearchFn -> AdjacencyList
  -> Acc' -> * -> Exp Acc'
type instance Eval (AddToComponent search adj
  '(ccs, used) node) =
    Eval (AddIfNonEmpty '(ccs, used) =<<
      DFS search adj node '( '[], used))
```

The above implementation works as follows. `GetComponents` gets connected components of the graph via DFS with the given search function. This runs `AddToComponent` on each node of the graph in order, which builds a connected component by adding all unused nodes that it encounters to a stack. Once the DFS run by `AddToComponent` is complete, `AddIfNonEmpty` adds the connected component to the list of connected components `ccs` if the component contains any nodes.

To apply our generic implementation above to strongly connected components, we use Kosaraju's algorithm [19]. This algorithm has two steps. First, perform a topological sort. Then, on each of the nodes of that sort in order, build components through a second DFS on the transpose of the graph. Notably, if each strongly connected component is of size one, the original topological ordering is preserved – since if every node has its own connected component, the order of nodes does not change.

`DoTopsort` requires that `SCCsFromTopsorted` finds the strongly connected components given a list of topologically sorted nodes in as input. This is exactly the second step of Kosaraju's algorithm, so we get components via out-edges of the transpose graph. Out-edges of the transpose graph are the same as in-edges of the original graph – so we simply define `SCCsFromTopsorted` as `GetComponents GetInEdges`.

Now we need to check for loops using the results of strongly connected components search. We do this by now attempting to flatten the list of components into a single list of nodes. If any of the components contain more than one node, we error since a strongly connected component with more than one node in it dictates a loop in the graph. If we do not error, then every strongly connected component is of size one and thus the original topological ordering is returned.

This is implemented by `FlattenSingletons` as shown below, which is as a wrapper around a closed type family `FLL`.

```
data FlattenSingletons :: [[*]] -> Exp [*]
type instance Eval (FlattenSingletons xss) =
  FLS xss

type family FLS (xss :: [[*]]) :: [*] where
  FLS '[] = '[]
  FLS ('[x] ': xs) = x ': FLS xs
  FLS (xss ': xs) = TypeError ...
```

3.2.4 Weakly Connected Components Search.

Weakly connected components search is implemented using DFS that expands both in- and out-edges. This provides us with all weakly connected components, as such a component only consists of nodes that are reachable from some node in that component [7]. As such, we reuse the generic component search implemented earlier, but with the expansion of in- and out-edges:

```
type ConnectedComponents = GetComponents GetEdges
```

4 Reifying the Results

We have discussed several graph algorithms for sorting nodes, implemented at the type-level. While it is useful for us to perform these at the type-level to ensure statically that there is a valid ordering of the computations, we now wish to produce a resulting, value-level program in which the computations are composed according to their ordering. This presents a challenge because our graph algorithms work entirely at the type-level and the results are therefore disconnected from the value-level computations. We remedy this in this section by describing a general reification technique for reassociating the computations with the results of the graph algorithms. For example, consider a function with the following type signature, where `Sort` is some type family that sorts a type-level list:

```
toSorted :: xs' ~ Sort xs
  => HList xs -> HList xs'
```

We now have to find a definition which gives some `HList xs'`, where `xs'` is the result of the sort. There are traditionally two ways to bring a type directly down to the value level. The first is via some form of *reification*, where a direct mapping from types to values is used to convert any type to its respective value. This is easy if we know that the types are singletons – if a type represents only a single value, we simply map that type to its corresponding value. However, it is much harder if there are many possible values for a type – how would we reify `MIO '( '[Cell Ptr "x" CInt], '[]) ()` to a single, unique memory computation?

The alternative method is to just re-implement a value-level version of our type-level computation via type classes. This allows us to work with any type regardless of whether it is a singleton or not, but requires two implementations of the algorithm – one at type-level and another at value-level. Furthermore, both versions of an algorithm will have to be updated if either changes.

It would be ideal to have some way of avoiding this issue. To this end, we present a generic approach of reification for the results of certain type families, like those described in Section 3 – *rearrangement*. This gives us value-level implementations of our type-level graph analyses for free.

4.1 The High-Level Idea

The key idea behind rearrangments is that by providing a value-level input – our `HList` – we already have values that correspond to the types in the type-level list. We make the assumption that a given type in the output list has the same corresponding value as the value of that type in the input list. This is a fair assumption to make for a variety of type-level analyses, such as type-level sorting and filtering.

As such, when it comes to reifying the result of a type-level algorithm, we can look up the value of a given type in our input and use that as the value of said type in the output. This can be described as rearranging the input list such that it matches the structure defined by the output type.

As an example, consider some value-level list of type `list`, which you would like to rearrange to have the same type as `list'`. We represent this operation with `rearrange list`, which rearranges a value-level list to match the target type. The head of `list'` is a `Bool`, so we look for the first `Bool` in the input list, find `True`, and put that at the head of our output. This is repeated for the rest of the types in the type of `list'` to generate the value-level list below.

```
list :: HList '[Int, Bool, ()]
list = 3 :+: True :+: () :+: HNil

list' :: HList '[Bool, (), Int]
list' = rearrange list
--    = True :+: () :+: 3 :+: HNil
```

This solves the issue of reifying types that have multiple associated values – for example, `Int` could be reified to many different values normally, but because we assume that the value in the input is the correct one to use in the output, we know to reify `Int` to `3` in the above example.

One issue we may come across is multiple elements of the same type in our input list. Consider an alternative `list` and `list'` specified below, where the lists contain duplicate types.

```
list :: HList '[Int, Bool, Bool]
list = 3 :+: True :+: False :+: HNil

list' :: HList '[Bool, Bool, Int]
list' = rearrange list
--    = True :+: True :+: 3 :+: HNil
```

Since `rearrange` selects the first occurrence of a given type when computing the output, you end up including `True` twice, like in `list'` above. If your aim is to permute some input list, this is not ideal as you would like all of the values in the input to be present in the output.

As such, when we use a value in our output, we must remove it from the input we are looking up values from. This means that a value in the input will not be used more than once, and will produce valid permuations. We represent this alternative version with `permute` in the below example, which correctly includes both `True` and `False` in the result list.

```
list'' :: HList '[Bool, Bool, Int]
list'' = permute list
--     = True :+: False :+: 3 :+: HNil
```

4.2 Implementation

Since all of the type families used for graph analysis permute their input in some way, we focus on the implementation of the version which deletes values from the input when they are used in the output. Our aim is to, given an input list `env`, return the output that has type `target` and the sublist of the input `env'` representing unused values. First, we look at the type signature and base case where we want to rearrange to an empty list.

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

instance RearrangeDel env '[] env where
  rDel l = (HNil, l)
```

Now we turn to the two recursive cases, which deal with whether the head of the output is a type or an inner `HList`. First, if the output head is some type, then we need to find the first value of that type in our input list, put it at the head of the output list, and then run the rest of the rearrangement without that value. This is implemented below, along with

the class definition of `GetHListElem` which retreives the first element of a given type and returns the list without that element. We omit its full definition for brevity.

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

instance {-# OVERLAPPABLE #-}
  (RearrangeDel env' target' env'',
  GetHListElem x env env') =>
  RearrangeDel env (x ': target') env'' where
    rDel l = (x :+: xs, l'')
      where (x, l') = getHListElem l
            (xs, l'') = rDel l'
```

To complete the definition, we need the overlapping case for when dealing with an inner `HList`. In this case, we need to rearrange the head itself and then continue to rearrange the rest of the outer list.

```
instance {-# OVERLAPPING #-}
  (RearrangeDel env head env',
  RearrangeDel env' target' env'') =>
  RearrangeDel env (HList head ': target') env''
    where
    rDel l = (head' :+: tail', l'')
      where (head', l') = rDel l
            (tail', l'') = rDel l'
```

Both this and the previous case rely on passing around the modified input list in order to propagate the deletions correctly. In rearrange without deleting from the input, this state passing is unnecessary and leads to similar but simpler definitions.

4.3 Applications

With this definition completed, we can now apply it to a collection of different use cases. First, we consider algorithms that use their input to lookup values, but do not use the entire input – for example, a filter that only finds types that meet certain criteria. For this, we can discard the leftover input, as follows.

```
rearrangeDel :: RearrangeDel env target env' =>
  HList env -> HList target
rearrangeDel = fst . rDel
```

We next consider type-level algorithms that permute their input – so have exactly the same types present in the input and output, just with possibly different ordering and different nesting. This is equivalent to rearranging with deletion with no leftover input. An example of this is the `toSorted` definition we introduced this section with, which we now present an implementation for along with `Permute` for type families exactly like this.

```
type Permute env target =
  RearrangeDel env target '[]

permute :: Permute env target =>
  HList env -> HList target
permute = rearrangeDel

toSorted :: (Permute xs xs', xs' ~ Sort xs)
  => HList xs -> HList xs'
toSorted = permute
```

We finally consider type-level algorithms that may have a larger output than input, possibly by duplicating some elements. In this case, rearrange without deletion is useful – type families that may return duplicate nodes can use this idea.

It is important to note that these ideas do not apply to just type-level graph algorithms. Any type-level algorithm that takes in some type-level list and returns a type-level list that shares all types with the input can utilise these methods.

5 Dependency Resolution & Ordering

We now have a graded monad for typing computations that read and write mutable memory, and mechanisms for building graphs at the type level and performing analyses on them. By combining these components, we now look at ordering individual computations so that none of their data dependencies are violated. We also implement functionality for concurrently running computations with no data flow dependencies.

5.1 Data Dependencies

Now that we have types that fully describe data dependencies, we can apply *Bernstein's condition* [2] to formally determine how computations should be ordered and can possibly be run in parallel. Bernstein's condition, here specialised to `Memory` computations, dictates that if the following holds for two computations `u :: Memory m '(rs, ws) a` and `v :: Memory m '(rs', ws') b` respectively, then their order of execution does not matter.

$$ws \cap ws' = ws \cap rs' = rs \cap ws' = \emptyset$$

We now look at the individual components of the equation. $ws \cap rs' \neq \emptyset$ is a formal statement of a data dependency, where `v` reads something written to by `u`. These are exactly the dependencies that we are focusing on. $rs \cap ws'$ is the inverse statement of a data dependency, so two computations can be run in parallel if there is not a data dependency in either direction.

With these dependencies now formally stated, we implement them as a type family `IsLessThan u v`, presented below. This returns `True` only if there is a data flow dependency from `u` to `v`. This uses `NonEmptyIntersect` defined back in Section 3.1.

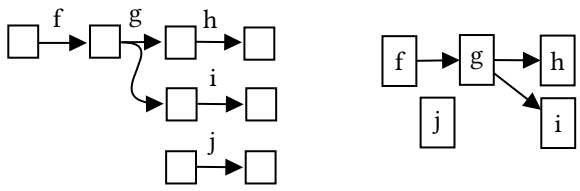

Figure 1. A hypergraph of computations between memory cells and its equivalent dependency graph for flow dependencies.

```
data IsLessThan :: * -> * -> Exp Bool
type instance Eval (IsLessThan
  (Memory m '(rs, ws) a)
  (Memory m' '(rs', ws') a')) =
    Eval (NonEmptyIntersect rs' ws)
```

The final component is $ws \cap ws' \neq \emptyset$, which dictates an *output dependency*, where two computations write to the same memory cell. If two computations have an output dependency between them, then whichever one is run last will set the final value of the memory cells that they share – so the result of the program depends on the ordering of these computations. However, we have no way of determining which computation should be run first with solely the type-level information given to us.

Therefore, we check for any output dependencies and error if they are present. Checking for output dependencies between two computations is implemented in the same way as `IsLessThan`, except checking ws and ws' rather than ws and rs'. This is run on every pair of computations, erroring if any output dependencies are found.

5.2 Dependency Graphs

We now use the formal definition of data dependencies to define a directed graph between computations, of which an example can be seen in Figure 1. In the first hypergraph, our computations are represented by arrows, which point from the memory cells read from to the memory cells written to by that computation. We then transform this graph into a dependency graph by drawing an edge for each data dependency between computations – for example, there is a directed edge from computation f to computation g since there exists a cell pointed to by f that g points out of.

This is implemented by `ToAdjacencyList IsLessThan xs`, a first-class type family that uses the aforementioned `IsLessThan` type family as a comparator for building a graph, given the list of nodes of that graph `xs`.

5.3 Ordering Memory Computations

Now that we have a dependency graph, we can use it to order our computations in such a way as to preserve these dependencies. Fortunately, this is exactly what a topological sort does, and this was implemented back in Section 3.2.3.

With this sort, we can now use rearrangments, specifically `Permute`, to bring the result of this to value level. Note that `Ordered IsLessThan xs` is a wrapper that combines building the adjacency list and performing topological sort on it.

```
ordered :: (Permute xs xs',
  xs' ~ Eval (Ordered IsLessThan xs)) =>
  HList xs -> HList xs'
ordered = permute
```

We have now successfully built a type-level analysis for ordering computations without violating their data dependencies and brought the result of it easily to the value level. With this, we can easily order the example back from the introduction with just:

```
ordered (writeMotor :+: getDistance :+: HNil)
==> getDistance :+: writeMotor :+: HNil
```

5.4 Running Ordered Computations

With this sorted list of computations, we now need a way to run all of them. Recall the definition of `Memory m s a`, which defines `runMemory :: Set (MemoryUnion s) -> m a`. In order to run a `Memory` computation we need to provide an environment of type `Set (MemoryUnion s)` containing all of the mutable references that it works with. This means that in order to run a list of `Memory` computations, we need an environment containing the mutable references needed for every computation in the list.

We start by defining a helper function that allows us to run memory computations given any environment that contains all of the required mutable references, not just the exact environment required. This just applies `subset` to the input environment, which takes the entire environment to the subset needed by this computation.

```
runMem :: Subset (TupleUnion s) env =>
  Memory m s a -> Set env -> m a
runMem mem env = runMemory mem (subset env)
```

To run a list of `Memory` computations in order, we need to just run `runMem` on each computation in the list. This just involves building a typeclass to do exactly that and collect the results together. With this defined, we now have the same functionality as the example at the start of the paper, but with the guarantee that the computations will be ordered correctly.

5.5 Partial Updates

So far, we are able to run a given list of computations in order. However, in some settings this may be inefficient – to go back to the robot example, what if we know that some inputs update less frequently than others? In that case, we would not actually need to run computations that solely take in those slower inputs as frequently as the other computations – since our computations read some mutable state as input, perform pure calculations and then possibly write to some outputs, if the input is unchanged then the output will be too.

As such, it would be helpful to work with *partial updates* – where we tell the program that only some subset of input mutable data has been updated since the last run of the network, and as such we only need to run those computations that have possibly changed inputs and thus are possibly able to change their outputs. We do this conservatively – that is, we run a computation if there is any chance of its inputs having changed – so that it produces the same result as a regular run.

We need a more complex example to show these partial updates at work. We consider a robot which only acts in the dark, but has an unreliable light sensor that produces noisy data. We split this functionality into two parts – a computation for retrieving the light sensor value, and a computation that averages sensor values to remove noise, writing `1` if the robot should act and `0` otherwise. We present code for these below.

```
getLight :: MIO '( '[IOCell "sensor" Int],
  '[IOCell "normsensor" Int]) ()
getLight = do x <- readMem @"sensor"
  writeMem @"sensorVal" x

shouldAct :: MIO '(
  '[IOCell "avg" [Int], IOCell "normsensor" Int],
  '[IOCell "active" Int, IOCell "avg" [Int]]) ()
shouldAct = do x <- readMem @"sensorVal"
  vals <- readMem @"avg"
  let readings = x : vals
  writeMem @"avg" readings
  let acts = if (average readings) > 100 then 0
    else 1
  writeMem @"active" acts

robot' = getLight :+: shouldAct :+: HNil
```

An important point should be made about `shouldAct`, which updates a cell rather than just reading or writing it. This means that its inputs change every time it is run, since it writes to one of its own inputs – so would need to be run even if none of its other inputs have changed. If we were to run this when `"sensor"` was unchanged, then we would not need to run `getLight` since none of its inputs have changed, but we would need to run `shouldAct` since `"avg"` may have changed.

The high-level idea behind partial updates is to look at each computation in order just like when performing a full update – so that the result will be the same as a full update – and decide whether the inputs could have changed. We assume that the only cells that would have been updated outside of the network are never written to – since otherwise a full run of the network would just overwrite that changed input since we write before reading. This process can be summarised as follows, with references to the simplified (without all typeclass constraints) Haskell definition following it.

1. Initialise a set of updated cells with the input provided. This stores the cells that *could* have been updated.
2. For each computation, in order:
   a. If any of its inputs are in the updated cells set, run the computation as its inputs have changed (line 8).
   b. If it writes to a cell that it also reads, like `shouldAct`, we must run this computation as the input that it also writes to was updated in the previous run of the computation (lines 5 and 9).
   c. If we ran this computation, add all cells that the computation wrote to the updated cells set (line 13).

```
1  class RunPartialMems m xs env where
2    runPartialMems :: HList xs -> Set env ->
3      [CellUpdate] -> m ()
4
5  instance (RunPartialMems m xs env, b ~
       NonEmptyIntersect rs ws) =>
6    RunPartialMems m (Memory m '(rs, ws) c ': xs)
       env where
7    runPartialMems (mem :+: mems) env upd =
8      if any (needsUpdate (Proxy :: Proxy rs)) upd
9        || reifyBool (Proxy :: Proxy b)
10     then do
11       res <- runMem mem env
12       runPartialMems mems env
13         (updateEffects (Proxy :: Proxy ws) upd)
14     else runPartialMems mems env upd
```

With this, we now have a way of effectively simulating a full run of the network while only running part of it. This means that if we know an input has not changed, we can omit that input from the list of cells that have changed – thus avoiding unnecessary computation.

5.6 Concurrency

The other analysis that we consider is looking at whether certain computations can be run in parallel. We have already discussed Bernstein conditions, which dictate whether the

order of two statements matters – if there are no dependencies between two statements, their order does not matter and thus they can be run in parallel.

We note that if there is no directed edge between two nodes in our dependency graph, there is no data dependency in either direction between the two computations. We have already disallowed output dependencies by construction, so if there is no edge between two nodes in our dependency graph, the computations they represent can be run in parallel.

With this, we can represent the problem of finding groups of computations that can be run in parallel as finding weakly connected components in the dependency graph. We have already seen an implementation for this, `ConnectedComponents`, back in Section 3.2.4. This means that the implementation of connected components search at the value level is almost identical to `ordered` earlier, except the type family used with `permute` is instead for connected components.

Running this value-level list of components concurrently is fairly trivial – call `runMems` on each component in parallel using `forkIO`.

With this implementation, we now have a way to automatically run computations which do not share mutable memory concurrently. This is ideal for computations which do significant work and are thus sensible to parallelise, however we do no such checking for this – meaning that in some cases, introducing concurrency will actually reduce performance.

5.7 Bringing It All Together

We can now express our example from the start of the paper with our system. For this section, we consider a robot which does what our previous two examples do in parallel – both avoiding obstacles and checking whether it is dark. To run our combined robot, we need to first build an environment of the mutable variables we have access to.

```
getDistanceCell, getActiveCell :: IO (Ptr CInt)

getEnv :: IO (Set ...)
getEnv = do
  -- distribute runs each computation in a HList
  addrs <- distribute @IO $
    -- toCell :: IO (v t) -> IO (Cell v s t)
    toCell @"dist" getDistanceCell :+:
    toCell @"distVal" (newIORef @CInt 0) :+:
    ...
    toCell @"active" getActiveCell :+: HNil
  return $ toSet addrs
```

With this, we can now run our robot's actions correctly. We provide a new type `Program` for combining the environment and computations into one object, which is built via `makeProgram`. This also orders the input computations using the earlier discussed `ordered`, and verifies that there are no cells with the same name but different types. We can therefore run this example with this environment as shown below.

```
main = do
  let robot = shouldAct :+: getLight :+: getDist
    :+: writeMotor :+: HNil
  env <- getEnv
  program <- makeProgram robot env
  -- run when both inputs are changed
  runProgram robot
  -- run when only distance sensor is changed
  runPartialProgram robot [CellUpdate "dist"]
```

This is great, but has two minor flaws that we will briefly discuss solutions for. First, the types produced for computations are very long so need to be easily inferrable. However, due to how polymorphic `Cell` is, we need multiple type annotations for each call of `readCell`. Without the type annotation, we would have to write `getDist` as follows.

```
getDist = do
  x <- readCell @"dist" @Ptr @CInt
  writeCell @"distVal" @IORef x
```

This is irritating not just because writing out this many types with each use of a memory cell is annoying, but also because we have information as to what these types are in `env`. We provide `withEnvM` to solve this, which fills in the missing types from the computation by looking up each named `Cell` in the type of the environment with a type family, and unifying that type in the environment with the type in the computation.

For example, `readCell @"dist" :: ... Cell a "dist" b ...`, but by unifying the `Cell` type with `Cell Ptr "dist" CInt` from the environment, `a` and `b` are found. This allows us to rewrite the above as the following, thus avoiding all type annotations except for the names of cells being used.

```
-- withEnvM :: m (Set env) -> Memory m '(rs, ws) a
     -> Memory m '(rs, ws) a
getDist = withEnvM getEnv $ do
  x <- readCell @"dist"
  writeCell @"distVal" x
```

The other issue we provide a solution for is the fact that we have to add every cell used into the environment. While this is impossible to avoid for the inputs and outputs of the network, all of the intermediate cells used have to also be allocated manually by the programmer – leading to very large environments.

To this end, we provide the means for cells to be automatically allocated when creating the program in `makeProgram` via a new type `AutoCell s t`. When a computation uses an `AutoCell`, `makeProgram` automatically allocates it as an `IORef` with the default[8] value for that type. We could further rewrite the above

[8] via `data-default` (https://hackage.haskell.org/package/data-default)

example to take advantage of this – the below example does not need a `Cell` called inter to be present in the environment. Note that `writeAutoCell` needs to be applied to an `AutoCell` type, so `DVal` in the below example.

```
type DVal = AutoCell "distVal" CInt
getDist = withEnvM getEnv' $ do
  x <- readCell @"dist"
  writeAutoCell @DVal x
```

These two techniques greatly aid type inference of the very polymorphic `readCell` and `writeCell` while still letting us express a variety of stateful operations and track the effects of those operations.

5.8 A Larger Example

Finally, we consider a larger example of the same structure as in Figure 1 to show our system at work. In this example, we use some pointers to foreign memory alongside `IORefs` – these effectively allow for communication between C and Haskell via memory locations that both can access. In our implementation we have provided a very simple C wrapper which just asks the user for the value of its inputs – however this could be expanded to work directly with more complex C code, such as low-level GUI bindings and code for working with hardware.

In this example, we consider the concurrent execution of two sets of computations. The large subtree, containing all computations excluding j, checks for high values from some important input sensor. This is implemented through the following: first f retrieves the data from that sensor and converts it to the right type. g then retrieves that value and writes the same value to the input of h, and an averaged version to the input used by i. Then, each of h and i write a value if their input is above a threshold – thus finding immediate high values and more consistent peaks respectively. Meanwhile, the small subtree containing just j averages another input while the other subtree is running.

This setup could be used for tracking a sensor while also keeping track of metadata about it, *e.g.* the delay between accesses of that important sensor. This example has a dependency graph which has a few useful properties for showing that our ordering works – both in being complex enough to require that g is executed at a specific time and having subtrees that can be run concurrently.

The full code for this example can be found alongside the rest of the our implementation[9]. For brevity, we only look at part of the code here and talk about the results of its execution.

It is interesting to note that through the use of `AutoCell` and `withEnvM`, all function types can be inferred with minimal use of explicit type applications. To give an example of this, here is the implementation of g. Note that `Interm` is a type representing the memory location that forms the input to g, while `Interm2` and `Interm3` represent the inputs to h and i respectively, and `Avgg` represents a memory location storing the average.

```
g = withEnvM getEnv $ do
  inp <- readAutoCell @Interm
  memoryIO $ putStrLn $ "g " ++ show inp
  averaging <- readAutoCell @Avgg
  let averaging' = take 5 (inp : averaging)
  let avg = sum averaging' `div` 5
  writeAutoCell @Avgg averaging'
  writeAutoCell @Interm2 inp
  writeAutoCell @Interm3 avg
```

Each computation in our example has a single call to `memoryIO` – which runs an `IO` computation within `Memory` – as to show which order they are run in. When running this example, our dependence analysis chooses a valid ordering: j, f, g, h, i. When we run the program concurrently, our analysis finds that j can be run separately to the rest of the computations. As such, its position within the eventual order that these computations are executed in can also change – by adding a small `threadDelay`, j moves later within the order of execution. None of the other computations change their order if any of them are delayed.

6 Related Work

Applications of graded monads. Graded monads have been used in a variety of different contexts to provide additional information at compile time. The original embedding of graded monads into Haskell by Orchard and Petricek [15] presents a variety of uses and examples of graded monads, such as a variant of `Reader` which tracks what parts of the environment are read. They also briefly discuss use of graded monads for program analysis where the grade has no representation at value level. Ivašković and Mycroft [9] use graded monads to specify which locks are acquired by a given concurrent computation, with the taking of locks verified as correct via constraints on the primitive which takes said locks. Ivašković, Mycroft and Orchard [8] represent dataflow analyses such as liveness and constant propagation as a graded monad, with the grade tracking the set of live variables via a more complex definition of `Plus`. All of these explore alternatives to program analysis via the explicit graph transformation we perform in this paper.

Concurrency. Haskell allows for concurrency through simple lightweight threads via `forkIO` alongside helpful types for dealing with shared memory such as `MVar` [17]. However, this can lead to a variety of concurrency bugs that are too

[9] in `app/ComplexExample.hs`

complex to express in the type system. We have already discussed Ivašković and Mycroft's use of graded monads [9], which prevents deadlocks within explicit concurrency. Orchard and Yoshida [16] discuss a variety of implementations for working with session types in Haskell, which allow for verification about how concurrent processes send messages to each other. These approaches allow for type-level verification of more explicit and fine-grained concurrency than what we consider in this paper, but do not allow for simple automatic concurrency like ours.

Networks of computations. In finding dependencies between computations, we have been considering networks of computations between memory cells. This is useful in a few contexts. Chupin and Nilsson [4] provide a formulation of arrowised functional reactive programming that is modelled exactly like that – memory cells between arrows. Dataflow programming [10] can also be formulated like this – some visual versions are already presented as acyclic graphs, except with nodes being computations and edges being the inputs and outputs piped between said computations. Communication between actors in the actor model can be viewed in a similar way, with the passing of messages being edges in a graph between actors – Christakis and Sagonas [3] use this idea to build a communication graph, which marks which actors talk to each other in the same way that we mark which computations share memory as to find communication deadlocks within Erlang code.

Our model of computations as hyperedges between mutable memory is also similar to that of propagator networks as described by Radul [18]. In this model, we have a collection of propagators between inputs and outputs that are run when any of their inputs change, until it is determined that a network has reached a fixed point. Our type-level analyses of what is read and written could apply here, to determine what propagators may need to be run after a given propagator fires.

Type families. We use the first-class type families technique to write more reusable code at the type-level. This workaround will no longer be necessary once the work by Kiss et. al [14] is implemented in GHC, which would then allow for the partial application of type families out-of-the-box. We also use our rearrangements technique to avoid duplicating implementations at the type-level and value-level. Eisenberg and Stolarek [5] provide a Template Haskell approach for promoting most value-level functions to type-level ones.

7 Conclusions

We have shown how to perform dependence analysis in Haskell's type system. This allows us to automatically compose individual computations in an ordering that ensures no data dependencies are violated. This is accomplished by using a graded monad which is indexed over data dependencies. These types are used to build a dependency graph whose nodes can then be sorted accordingly. We guarantee that writes to memory happen before reads to that same memory, thus avoiding ordering bugs in systems where this ordering makes sense.

We have also presented additional runtime benefits using these analyses. Partial updates of a set of computations allows us to only run computations whose inputs changed, thus allowing fewer computations to be run. We also use our analysis to get concurrency for free by looking for groups of computations which do not affect each other.

In building these type level analyses, we highlighted a useful implementation technique: our rearrangements (Section 4) avoid reimplementing the same type-level algorithm at value level. This technique allows us to change the analysis used if needed with minimal effort compared to traditional type-level implementations of algorithms. They also apply generally to analyses that look to reorder a collection of types, so could be used in other domains.

We also briefly discussed how to make programming with our type-level data dependencies easier by making them more easily inferrable. This leads to the need for fewer explicit type annotations, which could also apply when using graded monads in other contexts.

8 Future Work

There are a few avenues for future work and future exploration in this project. The concurrency we have implemented is useful in some contexts, such as the robots presented which have obviously separate tasks. However, programs that aim to build up a single result from some set of inputs cannot be split into concurrent groups because their dependency graphs form a tree. To this end, a future analysis may be determining components which are nearly separate, running the parts that are separate concurrently, and then finally running the computation which combines them once those concurrent processes are done.

In this paper, we have focussed specifically on data dependencies, which have allowed us to correctly order computations that build up some outputs from a given set of inputs. Future work could instead look at the reverse, anti-dependencies. This may be useful in reading an entire state first before writing to it, thus updating said state with each run. This would require a rethink of partial updates, however. Similarly, allowing computations to be partially ordered by the programmer via a priority number or similar could allow for more fine-grained specification of our ordering assumptions.

Acknowledgments

We would like to thank the three anonymous reviewers for their detailed comments – they were constructive and helpful in making this a better paper. We would also like to thank Gihan Mudalige for his comments on an early draft of this manuscript, and Arved Friedemann for discussion of propagators. The first author is funded via EPSRC grant #2436228.

References

[1] Utpal Banerjee. 1988. An introduction to a formal theory of dependence analysis. *The Journal of Supercomputing* 2, 2 (1988), 133–149.

[2] A. J. Bernstein. 1966. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers* EC-15, 5 (1966), 757–763. https://doi.org/10.1109/PGEC.1966.264565

[3] Maria Christakis and Konstantinos Sagonas. 2010. Static Detection of Race Conditions in Erlang. In *Practical Aspects of Declarative Languages*, Manuel Carro and Ricardo Peña (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 119–133.

[4] Guerric Chupin and Henrik Nilsson. 2019. Functional Reactive Programming, Restated. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming* (Porto, Portugal) *(PPDP '19)*. Association for Computing Machinery, New York, NY, USA, Article 7, 14 pages. https://doi.org/10.1145/3354166.3354172

[5] Richard A. Eisenberg and Jan Stolarek. 2014. Promoting Functions to Type Families in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 95–106. https://doi.org/10.1145/2633357.2633361

[6] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. 2014. Closed Type Families with Overlapping Equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 671–683. https://doi.org/10.1145/2535838.2535856

[7] John Hopcroft and Robert Tarjan. 1973. Algorithm 447: Efficient Algorithms for Graph Manipulation. *Commun. ACM* 16, 6 (June 1973), 372–378. https://doi.org/10.1145/362248.362272

[8] Andrej Ivašković, Alan Mycroft, and Dominic Orchard. 2020. Data-Flow Analyses as Effects and Graded Monads. In *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 167)*, Zena M. Ariola (Ed.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 15:1–15:23. https://doi.org/10.4230/LIPIcs.FSCD.2020.15

[9] Andrej Ivašković and Alan Mycroft. 2020. A Graded Monad for Deadlock-Free Concurrency (Functional Pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell* (Virtual Event, USA) *(Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/3406088.3409024

[10] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. 2004. Advances in Dataflow Programming Languages. *ACM Comput. Surv.* 36, 1 (March 2004), 1–34. https://doi.org/10.1145/1013208.1013209

[11] Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. *SIGPLAN Not.* 49, 1 (Jan. 2014), 633–645. https://doi.org/10.1145/2578855.2535846

[12] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. 2010. *Fun with Type Functions.* Springer London, London, 301–331. https://doi.org/10.1007/978-1-84882-912-1_14

[13] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. 2004. Strongly Typed Heterogeneous Collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/1017472.1017488

[14] Csongor Kiss, Tony Field, Susan Eisenbach, and Simon Peyton Jones. 2019. Higher-Order Type-Level Programming in Haskell. *Proc. ACM Program. Lang.* 3, ICFP, Article 102 (July 2019), 26 pages. https://doi.org/10.1145/3341706

[15] Dominic Orchard and Tomas Petricek. 2014. Embedding Effect Systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell* (Gothenburg, Sweden) *(Haskell '14)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/2633357.2633368

[16] Dominic Orchard and Nobuko Yoshida. 2017. *Session types with linearity in Haskell.* River Publishers, Delft, The Netherlands, Chapter 10, 219–242. https://www.riverpublishers.com/book_details.php?book_id=439

[17] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. 1996. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg Beach, Florida, USA) *(POPL '96)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/237721.237794

[18] Alexey Radul. 2009. *Propagation networks: A flexible and expressive substrate for computation.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[19] M. Sharir. 1981. A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications* 7, 1 (1981), 67–72. https://doi.org/10.1016/0898-1221(81)90008-0

[20] Robert Endre Tarjan. 1976. Edge-Disjoint Spanning Trees and Depth-First Search. *Acta Inf.* 6, 2 (June 1976), 171–185. https://doi.org/10.1007/BF00268499