A Graded Monad for Deadlock-Free Concurrency (Functional Pearl)

Andrej Ivašković
Department of Computer Science and Technology
University of Cambridge
Cambridge, United Kingdom
andrej.ivaskovic@cst.cam.ac.uk

Alan Mycroft
Department of Computer Science and Technology
University of Cambridge
Cambridge, United Kingdom
alan.mycroft@cst.cam.ac.uk

Abstract

We present a new type-oriented framework for writing shared memory multithreaded programs that the Haskell type system guarantees are deadlock-free. The implementation wraps all concurrent computation inside a *graded monad* and assumes a total order is defined between locks. The grades within the type of such a computation specify which locks it acquires and releases. This information is drawn from an algebra that ensures that types can, in principle, be inferred in polynomial time.

CCS Concepts: • **Theory of computation** → **Program analysis**; *Type structures.*

Keywords: graded monads, concurrency, synchronization, deadlock

ACM Reference Format:
Andrej Ivašković and Alan Mycroft. 2020. A Graded Monad for Deadlock-Free Concurrency (Functional Pearl). In *Proceedings of the 13th ACM SIGPLAN International Haskell Symposium (Haskell '20), August 27, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3406088.3409024

1 Introduction

Writing concurrent programs that deal with shared memory is frequently challenging, particularly if they use the thread model. Race conditions can give rise to unexpected outcomes of parallel computations when synchronization is not done properly, while deadlock is a persistent problem. To address this, alternatives to the threaded model of concurrency exist – for example, a popular model seen in Haskell is *software transactional memory*, exemplified by the STM monad [7]. These alternatives sometimes come with a performance hit or lack a progress guarantee, so there is a tradeoff between efficiency and prevalence of bugs.

Threads are still the most common approach to concurrency in most imperative languages, with synchronization performed through facilities such as monitors (Java, C++11) or semaphores (POSIX `pthreads`). Static analysis tools such as `rccjava` [3] for Java are sometimes used to detect possible concurrency bugs. Formal models of concurrency (CCS [13], π-calculus [14], session types [8]) are mature and well-known approaches to reasoning about thread-safety.

Effect systems are a common framework for static analysis of programs in functional languages. They were first defined by Lucassen and Gifford [12] and applied to reasoning about side-effects in parallel programs, and since then have been used in numerous other contexts. An effect system defines type-and-effect judgements of the form $\Gamma \vdash e : A \& F$, where Γ is the context (types of variables that may occur freely in e), e is an expression, A is the type of e, and F represents the *effect* of e. Note that such effects F, in general, represent the composition of the various effects of the sub-expressions of e rather than being seen as mere individual primitive effects.

Wadler and Thiemann [22] show that effect systems correspond to monads, as they serve a similar purpose (delimiting computational effects). In their work, Wadler and Thiemann demonstrate how a closed expression e satisfying the type-and-effect judgement $\vdash e : A \& F$ in a language with impure features corresponds to an expression e' of type $T^F A$ in a pure language with support for monads, where T^F represents a computation *indexed* by effect F (there is a separate type constructor T^F for every possible effect F). The bind operation in these monads keeps the effect annotation of the monad 'fixed', so the type of bind is $(\gg=)_{F,G} : T^F A \to (A \to T^G B) \to T^{F \cup G} B$. Whilst such indexed monads suffice for Lucassen-Gifford effect systems, they fail to model richer (*sequential*) effect systems that have separate operations for sequential composition $\star$ and for alternation $\sqcup$. More recently, Katsumata [9] has shown that the semantics of effect systems is more accurately described by *graded monads.* This means that all T^F now form a family of type constructors, but not all of them correspond to a monad – instead, F are drawn from an *effect algebra* which is a monoid with some binary operator $\star$. Graded monads thus compose in a way that more closely resembles how effects interact, as now bind is actually a family of operators $(\gg=)_{F,G}$ of type $(\gg=)_{F,G} : T^F A \to (A \to T^G B) \to T^{F \star G} B$.

Haskell '20, August 27, 2020, Virtual Event, USA
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8050-8/20/08.
https://doi.org/10.1145/3406088.3409024

This is the basis of the Haskell embedding of effect systems by Orchard and Petricek [15].

The natural question arises: is there a graded monadic view of concurrency that guarantees that well-typed programs cannot deadlock?

Our contribution. We provide a simple framework for writing a subset of multithreaded Haskell programs guaranteed to be free of deadlock (Section 3). The implementation wraps all concurrent computation inside a *graded monad* so that the type-checking algorithm explicitly rejects all programs that might lead to deadlock. We demonstrate its use through the examples of the *dining philosophers problem* and *memory regions*. We are able to get far without relying on anything beyond the type system of Haskell (for example, dependent types).

The theoretical foundation for this implementation is an algebra representing the concurrency information (Section 5). This algebra is a partially ordered monoid, the partial order is a lattice and all of the operators used in the emitted type constraints of the monad operations are monotonic. This ensures that inferring safety can be computed in polynomial time in language similar to Haskell, but with added support for equirecursive types. There is a slight disconnect between the theory and the Haskell implementation, since the mathematical view of graded monads makes totality assumptions about the underlying algebraic structure – the difference is seen in the way ill-typed expressions are handled.

2 Big Picture

Before taking a deep dive into the implementation, we set the stage by outlining the key components. In this section, we first discuss the members of a Haskell typeclass `GradedMonad`. We then describe the requirements of this monad, that is, what programs written using our particular graded monad – which we call `Sync` – look like, as well as what kinds of computation are permitted.

2.1 Graded Monads

Haskell programmers are well acquainted with programming using monads. To define a monad `m`, the programmer has to implement a function `return :: a → m a` that wraps a pure value inside this monad, as well as a bind function (commonly used as an infix operator):

```
(≫=) :: m a → (a → m b) → m b.
```

A graded monad is defined by the *grading algebra* over which the effect annotation range, the return function and a family of bind (≫=) operations. The grading algebra is a partially ordered monoid: its identity element corresponds to the *unit* effect, its binary operator ⋆ corresponds to sequencing two monadic values, and the partial order is the basis of the subtyping relation between two monadic values. One way of representing a graded monad is using a typeclass:

```
class GradedMonad (m :: d → Type → Type) where
  type Unit m :: d
  type Seq m (r :: d) (s :: d) :: d
  type Sub m (r :: d) (s :: d) :: Constraint
  return :: a → m (Unit m) a
  (≫=) :: m r a → (a → m s b)
          → m (Seq m r s) b
  sub :: Sub m r s ⇒ m r a → m s a
  -- The following lines are logically inessential
  -- but ensure that Haskell's (≫) operator works
  -- as expected with RebindableSyntax.
  (≫) :: m r a → m s b → m (Seq m r s) b
  x ≫ y = x ≫= const y
```

The identifier `d` refers to the grading algebra implemented at the type level; note that no values of type `d` are ever constucted at run-time. This grading algebra comes with a sequencing operation supplied using the type constructor `Seq` (so the type `Seq m r s` yields a value of the same graded monadic type wrapped inside `m` whose grade is $r \star s$) and as a unit grade (`Unit`). This implementation leverages the `Constraint` GHC extension to model subtyping: `Sub m r s` represents a fact (or constraint) stating that type `m r a` is a subtype of `m s a` for any type `a`.

The definition of (≫) is optional. If it is included and if the `RebindableSyntax` extension is used, the meaning of `do` changes – instead of being syntactic sugar for monadic computation, it becomes syntactic sugar for sequenced graded monadic operations. All usages of `do` in this paper refer to the graded monadic version.

2.2 Concurrency Framework

The goal is to allow the programmer to write parallel programs that are guaranteed to not deadlock. This is possible using the `GradedMonad` typeclass as previously defined.

All concurrent expressions have type `Sync r a`, where type `r` represents information about the locks that are to be acquired and released during execution, and `a` is the type of the result of this computation. The key idea is that locking is structured: each lock, once acquired, has to be released within the same context. An expression of this graded monadic type is then evaluated using the `runSync` function.

The two operations (in addition to the existing monad operators (≫=, `return` and polymorphic operators such as if-then-else instantiated at monadic type) we use are:

Synchronization Given a finite set of global named locks `X`, `Y`, . . . , we define a family of functions `syncX`, `syncY`, . . . that take a computation `e` wrapped inside the `Sync` graded monad and return another computation wrapped inside `Sync` that does the same as `e`, but has exclusive access to the respective lock. Thus `syncX e` could be implemented as

```
lockX ≫ e ≫= λr → unlockX ≫ return r
```

provided `lockX` and `unlockX` are not exposed to the user.
The expression `syncX e` should type-check if and only if the type of `e` is `Sync r a`, where `r` contains only those locks that are allowed to be acquired *after* acquiring `X`. In this case the annotation `s` in the type of the result `Sync s a` should be `r` with `X` added.

Parallelism Given computations `e1` and `e2` wrapped inside the `Sync` graded monad, there is a computation `e1 || e2` wrapped inside `Sync` that runs `e1` and `e2` in parallel.

It is important that the internals of `Sync` are not exposed to the user of the library – otherwise, it is possible to write programs that override the type annotations that guarantee deadlock freedom. Instead, the user should have access to the synchronization and parallelism operations. This means that the set of locks has to be hardwired into the library and known ahead of time.

Path to monitors. An extension to the basic model of concurrency described above allows the programmer to synchronize *on* data. This means that two threads sharing state (possibly reading and writing to it) will each, in turn, have exclusive access to the shared state. This resembles the *monitor* framework of concurrency (seen, for example, in Java). We present the details of how this can be implemented, which comes with changes in types and usage of the synchronization operations.

Synchonizing on data is nothing more than an extension of synchronizing on a single lock: a mutual exclusion lock can be seen as a shared piece of unit-type data.

Limits. The kind of concurrency we consider safe is limited. For example, the following singlethreaded program, guaranteed to not deadlock, should be explicitly rejected by the type system:

```
syncX (syncY (return 1)) ≫
syncY (syncX (return 2))
```

This would be the sequence of lock acquisions and releases:

1. acquire lock X;
2. acquire lock Y;
3. release lock Y;
4. release lock X;
5. acquire lock Y;
6. acquire lock X;
7. release lock X;
8. release lock Y.

This is because the structured lock ordering assumption is violated: in the first half of the program (lines 1–4), X takes precedence over Y, but in the second half of the program (lines 5–8), Y takes precedence over X.

Contrast with theory. The type of `syncX` requires the argument to satisfy an ordering constraint. This is not difficult to do in Haskell using $\Longrightarrow$, where the constraints are listed on the left-hand side. This can model a *partial* addition operator on types: $add(x, \ell)$ is defined only when x comes before all locks in ℓ in terms of the lock ordering, in which case it is equal to the set ℓ extended with x.

The mathematical treatment of graded monads assumes all operators are *total*. The grading information is thus either a set or an element denoting unsafe computation, which we call *error*.

3 Implementation

We demonstrate, step-by-step, how to construct a graded monad and define its operations so that code written using them is guaranteed never to deadlock. The idea is that the type checker only accepts those programs that obey an *ordering discipline*: there is a total order defined on the set of locks that determines in what order a thread can acquire them before releasing these locks. Programs that obey this discipline are guaranteed to be deadlock-free.

In this section, we first explain how to manipulate the `r` in `Sync r a`, that is, how to implement operations necessary for respecting the lock order at the type level (Section 3.1). Next, we define the `Sync` graded monad with all its operations (Section 3.2). The key part of the implementation concerns lifting IO operations such as threads sleeping or printing to `stdout` (Section 3.3). Finally we implement the synchronization and parallelism operations (Section 3.4). These features are sufficient to demonstrate our first example, which is the dining philosophers problem (3.5). This approach, although simple, is not particularly elegant and only makes sense under the assumption that `syncX`, `syncY`, ... are all used as binary semaphores (mutexes). We demonstrate a how a simple modification based on a stack of `ReaderT` monad transformers can be used to give a *monitor* rather than a simple mutex (Section 3.6), which then allows us to implement shared-state concurrency based on memory regions – instead of using a simple mutex lock to synchronize IO, we make it possible to synchronize *on* data.

3.1 Lock Algebra

Before defining the graded monad, it is necessary to describe how lock information is expressed at the type level (this is the `r` in `Sync r a`). The idea behind the implementation is that sets of locks should be represented as lists whose items are comparable (that is, have a total order defined). The locking discipline should be such that, if locks x and y satisfy $x < y$, then x needs to be acquired before y does. The natural requirement is that lists representing sets of locks are sorted and do not repeat items. Lists can be represented at the type level in Haskell using `DataKinds`, but an alternative implementation using GADTs is also possible. The `DataKinds` approach uses the notation `'[]` to represent the type-level equivalent of an empty list, and `':` as the type-level version of : (cons).

Let `xs` and `ys` be sorted lists representing sets of locks. We define the lock-algebra operators on such lists at the type level, starting with the *merge* operator `:∨:`. It leverages Haskell extensions for type families and type operators. Its implementation is simple:

```
infixr 5 :∨:
type family (xs :: [k]) :∨: (ys :: [k]) :: [k] where
  '[] :∨: ys = ys
  xs :∨: '[] = xs
  (x ': xs) :∨: (y ': ys) =
    If (Cmp x y == LT) (x ': (xs :∨: y ': ys))
   (If (Cmp x y == GT) (y ': (x ': xs :∨: ys))
                       (x ': (xs :∨: ys)))
type family Cmp (a :: k) (b :: k) :: Ordering
```

As we will see, subtyping in the graded monad is based on the relationship between lock order indices. If an expression potentially acquires locks in the order given by list `s`, then we know it also potentially acquires locks given by any `t` of which `s` is a subsequence. Type-level sublists (or subsequences) can be implemented using typeclasses.

```
class Sublist s t
instance Sublist '[] '[]
instance {-# OVERLAPPABLE #-}
  Sublist s t ⇒ Sublist s (x ': t)
instance {-# OVERLAPS #-}
  Sublist s t ⇒ Sublist (x ': s) (x ': t)
```

We implement a *partial addition* operator to represent adding a lock to a list. Recall the assumption that `syncX e` is valid only if the lock `X` is *less than* all the locks acquired in `e`; in this case `syncX e` acquires both `X` and the locks acquired in `e`. This partial operator `PartialCons` is implemented using Haskell typeclasses, where the third parameter is the result. The implementation requires specifying a functional dependency (→ below) in order for Haskell's type system to be able to infer that the result of partial addition is unique.

```
class PartialCons (x :: k) (xs :: [k]) (ys :: [k])
  | x xs → ys
instance (LessThanAll x xs)
  ⇒ PartialCons x xs (x ': xs)
```

Implementing `LessThanAll x xs` is simple: given the underlying assumption that the list of locks is sorted, it suffices to compare `x` with the head of `xs` (if it is non-empty). Type-level comparison is done using a typeclass `BoolAsConstraint` due to Orchard and Petricek [15] who call it `Conder`; it provides a way to express a boolean value as a Haskell constraint:

```
class BoolAsConstraint b
instance BoolAsConstraint True
class LessThanAll (x :: k) (ys :: [k])
instance LessThanAll x '[]
instance BoolAsConstraint (Cmp x y == LT)
  ⇒ LessThanAll x (y ': ys)
```

3.2 The Graded Monad

The underlying implementation of the `Sync` graded monad has to provide every thread access to all the existing locks in order to synchronize. A lock is just an `MVar` (imported from `Control.Concurrent`). Thus a set of mutual exclusion locks (call this set `Locks`) can be represented as a list whose items all have type `MVar ()`, and an expression with IO side-effects that needs to access these locks has type `Locks → IO a`. Hence the following is a sensible representation for `Sync`:

```
type Locks = [MVar ()]
newtype Sync (lockList :: [Symbol]) a =
  Sync { unSync :: Locks → IO a }
```

This is very similar to the `Reader` monad – in fact, the type `Locks → IO a` can be wrapped into `Reader Locks (IO a)`. The `lockList` type represents locks potentially acquired by an expression. We choose to associate locks with strings (`Symbol`), but alternative representations are possible. If the lock order is lexicographic, we instantiate `Cmp` with `Symbol`s:

```
type instance Cmp (v :: Symbol) (u :: Symbol) =
  CmpSymbol v u
```

`Sync` as defined above is a graded monad, which we now show by instantiating the required operations. To instantiate it, we require using the bind and `return` operations for the `IO` monad. In order to access everything in `Prelude`, yet overwrite the standard monad operations we write the following two lines at the start of the program:

```
import Prelude hiding (Monad)
import qualified Prelude as M
```

This also allows us to define the (≫=) and `return` functions for all graded monads, keeping the original operations for all monads (including `IO`) as (`M.`≫=) and `M.return`. We use this to define `GradedMonad`:

```
instance GradedMonad Sync where
  type Unit Sync     = '[]
  type Seq  Sync r s = r :∨: s
  type Sub  Sync r s = Sublist r s
  return = Sync ∘ const ∘ M.return
  Sync x ≫= k = Sync $ λl →
    let k' y = unSync (k y) l in x l M.≫= k'
  sub (Sync x) = Sync x
```

Most of these instantiations are unsuprising. An expression with no parallelism anywhere in its computation. Sequencing two possibly parallel expressions e_1 and e_2 results in an expression that acquires both the locks in e_1 and the locks in e_2. Sublists correspond to subtypes: an expression potentially acquiring a set of locks r also potentially acquires any superset of r. The implementations of `return`, ≫= and `sub` demonstrate that this is nothing more than a wrapper.

It is necessary to provide a way to run code wrapped inside the Sync graded monad. One simple way of achieving this is by introducing a runSync function that creates initially full MVars, and once all of them are in a list, this list as passed as an argument to the computation wrapped inside Sync. A quick and convenient way of doing this is the following:

```
runSync :: Sync r a → IO a
runSync e =
  let running l k =
    if k > 0 then
      newMVar () M.≫= λx →
        running (x:l) (k - 1)
    else unSync e l
  in running [] 5
```

assuming there are precisely five locks.

3.3 Lifting IO

The Sync monad should allow the programmer to write code with visible side-effects: printing to stdout. Given an expression e with IO side-effects wrapped inside of the IO monad, we want to be able to wrap it inside Sync. We achieve this by writing liftIO e, where liftIO is defined below. Such lifting is also necessary for implementing the synchronization and parallelism operations – they have to manipulate the MVars that represent the locks.

Due to our above definition of Sync, the implementation is simple:

```
liftIO :: IO a → Sync '[] a
liftIO e = Sync (const e)
```

This allows us to immediately lift two useful functions: sleeping and writing to standard output.

```
delay :: Int → Sync '[] ()
delay = liftIO ∘ threadDelay
syncPutStrLn :: String → Sync '[] ()
syncPutStrLn = liftIO ∘ putStrLn
```

3.4 Synchronization and Parallelism

We finally get to the core operations in the Sync graded monad. Given all the introduced facilities, their implementation is not particularly difficult. The most important part of synchronization and parallelism is imposing the right typing constraints.

We assume the RebindableSyntax extension is used. As a result, do {...} blocks use the GradedMonad versions of ≫ and ≫=.

Parallelism. Parallelism is unsurprisingly based on the forkIO function from Control.Concurrent. The implementation first creates two initially empty MVars – meaning that they can be written to – that are later used to store the results of the two threads. After that it is necessary to 'unwrap' the threads wrapped inside of the Sync graded monad in order to write their results to the Mvars that were introduced, as well fork these two modified threads. Finally the implementation blocks until the results of these two threads are known, which are then paired up and returned as the result of the computation. This is written as follows:

```
(‖) :: Sync r a → Sync s b
         → Sync (r :∨: s) (a, b)
Sync p ‖ Sync q = do
  la ← liftIO newEmptyMVar
  lb ← liftIO newEmptyMVar
  Sync $ λl → forkIO (p l M.≫= putMVar la) M.≫
              forkIO (q l M.≫= putMVar lb)
  x ← liftIO $ takeMVar la
  y ← liftIO $ takeMVar lb
  return (x, y)
```

Note the unwrapping necessary in the middle of the implementation. It is necessary to fork two threads, one executing p and the other executing q, but the result is written to an MVar (either la or lb) at the end of each one. This write operation is appended to the bodies of the threads using the (≫=) for the IO monad. Since p and q are expressions of type Locks → IO a and Locks → IO b, respectively, they have to be applied to a list of locks before the IO-level bind is used.

Synchronization. Synchronizing an expression e on lock x consists of four steps:

1. wait on lock x until it is available, then acquire it;
2. run e, call its result r;
3. release x;
4. return r.

This requires first defining locking (waiting and acquiring) and unlocking (releasing) operations. These are based on manipulating MVars in the list of locks. Given the position of a lock in the list of locks, this just requires accessing a list item:

```
lock :: Int → Sync '[] ()
lock n = Sync (λl → takeMVar (l!!n))
unlock :: Int → Sync '[] ()
unlock n = Sync (λl → putMVar (l!!n) ())
```

This allows us to create a generic approach to building synchronization functions. To this end we introduce the createSync function. It implements the four step process mentioned previously and it enforces the lock total order using PartialCons. Explicit application of Sync and unSync is required in order to modify the lock annotation for the expression being synchronized.

```
createSync :: PartialCons x r s
  ⇒ Int → Sync r a → Sync s a
createSync n e = Sync $ unSync $
  lock n ≫ e ≫= λr → unlock n ≫ return r
```

Finally, `createSync` can be applied to different integer constants. Their types explicitly name the lock identifier using the `PartialCons` typeclass. For example, for locks `X` and `Y`:

```
syncX :: PartialCons "X" r s
  ⇒ Sync r a → Sync s a
syncX = createSync 0
syncY :: PartialCons "Y" r s
  ⇒ Sync r a → Sync s a
syncY = createSync 1
```

This finalizes the implementation of the graded monad and its main operations. A simple test case on which to experiment with these functions is the following expression that does not enforce isolation between two threads that write to `stdout`:

```
syncPutStrLn "[thread␣1]" ||
syncPutStrLn "[thread␣2]"
```

In one run, this function might write the following to the standard output:

```
[thre[atdh r2e]a
d 1]
```

This is fixed by synchronizing on a common lock:

```
syncX (syncPutStrLn "[thread␣1]") ||
syncX (syncPutStrLn "[thread␣2]")
```

This only allows two possibilities: either `"[thread␣1]"` is written first in its entirety, or `"[thread␣2]"` is.

3.5 Example: Dining Philosophers

The dining philosophers problem is a commonly used example of concurrent programming where deadlock may occur. It can be phrased in the following way:

> Five philosophers are sitting at a round table and eating spaghetti for dinner. Their dexterity leaves a lot to be desired, for they are easily distracted and any attempt at eating with only one fork will result in a mess. Hence each philosopher will need two forks. There are exactly five forks on the table, one between each pair of adjacent philosophers. The philosophers must remain seated for the duration of the dinner and all of them must eventually finish eating.

A naïve first attempt at a solution to this problem assigns each philosopher the following behaviour:

1. think
2. wait for the fork to the left to be available, then take it
3. wait for the fork to the right to be available, then take it
4. think some more
5. eat
6. put the right fork down
7. put the left fork down

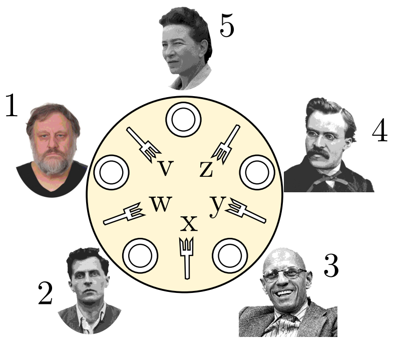

Figure 1. Setup of the dining philosophers problem. Deadlock happens if each philosopher acquires the fork to their left before the one on their right.

Unfortunately, this simple solution can lead to deadlock and hungry philosophers. Suppose the five forks are labelled `V`, `W`, `X`, `Y`, `Z` as in Figure 1. Suppose each philosopher manages to first grab the fork to their left: 1 acquires `V`, 2 acquires `W` and so on. Then the philosophers are stuck, since all of them are waiting for a fork to be released.

If the locks (forks) are acquired in a way that obeys a total order, there is no deadlock. Assuming the forks are ordered lexicographically, this means that Simone de Beauvoir (philosopher 5) should first pick up the fork to her right, and then the one of her left.

It is not difficult to implement this using the functions we defined so far. For example, the code implementing the behaviour of Slavoj Žižek (philosopher 1) is:

```
philosopher1 = syncV $ do
  delay 100000
  syncW $ do
    delay 100000
    syncPutStrLn "philosopher␣1"
```

Other philosophers are analogous, each acquiring the two forks in alphabetical order. Then their concurrent dining type-checks:

```
dining = philosopher1 || philosopher2 ||
         philosopher3 || philosopher4 ||
         philosopher5
```

Note, by contrast, that the naïve approach is rejected: the code below where Simone de Beauvoir (philosopher 5) attempts to take the left fork (`Z`) first is type-incorrect:

```
philosopher5 = syncZ $ do
  delay 100000
  syncV $ do
    delay 100000
    syncPutStrLn "philosopher␣5"
```

3.6 Memory Regions via Monad Transformers

The presented implementation of `Sync` has some drawbacks:

- there is no non-trivial shared state, all of the locks are simple mutual exclusion locks;
- the type system does not prevent `createSync` function accessing non-existent items in the lock list – that is, `createSync` may be called with an argument that leads to accessing an invalid index in the `Locks` list.

Both of these issues are caused by the underlying representation of `Sync a` as `Locks → IO a`. We now turn to an alternative approach via monad transformers, effectively interpreting `Sync` as a stack of memory regions. The aim is to synchronize *on* data – the main change is that new synchronization functions are now applied to functions of type `s → (Sync r a, s)`, where `s` is the type of the data synchronized on, and `a` is the type of the result of the computation.[1] Before, they were just applied to values of type `Sync r a`. This way, threads are able to share data. The goal is to provide monitor-like protection of variable access only in lock-protected critical sections.[2]

Thus the goal is to produce a new framework in which the following definition type-checks:

```
thr1 = do
  delay 1000000
  syncP $ λp → syncQ $ λq →
    return ((0, RegionP (2 ∗ qa q + 1) (pa p)),
            RegionQ (qa q) "touched␣by␣1")
```

where `RegionP` and `RegionQ` are structures defined as follows:

```
data RegionP = RegionP { pa :: Int, pb :: Int }
data RegionQ = RegionQ { qa :: Int, qs :: String }
```

The internal implementation so far can be represented as `Reader [MVar ()] (IO a)`. This list representation of the locks can be seen as a special case of a more general approach that uses a stack of monad transformers. We first introduce a transformer for locks:

```
type LockT l m = ReaderT (MVar l) m
```

These transformers can be combined to form an alternative definition of the `Sync` graded monad:

```
type InternalRep a = LockT RegionP
  (LockT RegionQ IO) a
newtype Sync (lockList :: [Symbol]) a =
  Sync { unSync :: InternalRep a }
```

The actual `InternalRep` type is defined depending on what memory regions (that is, locks) are used by the rest of the program. If there is need for an additional lock on another region `RegionR`, then the definition of `InternalRep` has an additional layer in the monad transformer stack:

```
type InternalRep a = LockT RegionP
  (LockT RegionQ (LockT RegionR IO)) a
```

The instantiation of `Sync` as a graded monad is similar to before, but we take advantage of bind at the `ReaderT` level:

```
instance GradedMonad Sync where
  type Unit Sync     = '[]
  type Seq  Sync r s = r :∨: s
  type Sub  Sync r s = Sublist r s
  return x = Sync $ M.return x
  Sync x ≫= k = Sync $ x M.≫= (unSync ∘ k)
  sub (Sync x) = Sync x
```

Running the monad can now be changed to take advantage of `runReaderT`:

```
performSync :: Sync r a
  → MVar RegionP → MVar RegionQ → IO a
performSync m lp lq =
  runReaderT (runReaderT
  (unSync m) lp) lq
runSync :: Sync r a →
  → RegionP → RegionQ → IO a
runSync m p q =
  newMVar p M.≫= λlp →
  newMVar q M.≫= λlq →
  performSync m lp lq
```

The IO lifting functions seen in the previous implementation now have to be lifted through several levels of `ReaderT` monad transformers. Assuming the definition of `Sync` given above, with two regions, the definition is as follows:

```
pureReaderT :: M.Monad m ⇒ m a → ReaderT s m a
pureReaderT = ReaderT ∘ M.return
liftIO :: IO a → Sync '[] a
liftIO = Sync ∘ pureReaderT ∘ pureReaderT
```

This has the desired result because `M.return` ignores the environment.

Synchronization. Synchronization is easier to implement than parallelism, so we turn to it first. As before, the synchronization functions (now `syncP` for region `P` etc.) can all be created using a single 'synchronization factory' function, now called `makeSync`, generalising the previous `createSync` function. This function takes as input a possibly parallel computation that eventually returns a reference to a lock, and returns a synchronization function of the kind described above:

```
makeSync :: Sync '[] (MVar l)
  → (l → Sync r (a, l)) → Sync s a
makeSync l k = Sync $ unSync $ do
  lock ← l
  v ← liftIO (takeMVar lock)
```

[1] Readers may observe that `s → (a, s)` is the underlying implementation of the `State` monad. This is unsurprising: the aim is to mutate the contents of a memory region.

[2] Monitors provide syntactic scoping to protect variable access. However, synchronization with monitors is done using `wait` and `signal` primitives. We only focus on the former aspect of monitors.

```
  (res, newv) ← k v
  liftIO (putMVar lock newv)
  return res
```

This is similar to createSync before, except that the lock reference is no longer an index in a list. Defining syncP and syncQ is simple and uses ask, from ReaderT, to access the environment:

```
syncP :: PartialCons "P" r s
  ⇒ (RegionP → Sync r (a, RegionP)) → Sync s a
syncP = makeSync $ Sync ask
syncQ :: PartialCons "Q" r s
  ⇒ (RegionQ → Sync r (a, RegionQ)) → Sync s a
syncQ = makeSync $ Sync (lift ask)
```

The ask value represents the computation that returns the value in the outermost level of the ReaderT monad transformer (on its own, ask has type Reader s s). To access the inner levels of the monad transformer stack, we use the lift function. If there are more regions in the monad transformer stack, the number of lifts increases.

Parallelism. Looking at the definition of ∥, it is clear that the main change in implementation has to be in spawning the two threads: there are several layers of LockT transformers and it is necessary to manipulates values wrapped inside the IO monad. Therefore it is necessary to 'unlift' the monad until the IO level is reached, append to the two threads the operations of writing their results to two fresh MVars, and use forkIO to spawn threads. Pushing the underlying IO-level bind is done by first introducing two helper functions.

```
pushBind :: M.Monad m
  ⇒ (a → m b) → LockT s m a → LockT s m b
pushBind k m = ReaderT $
  λx → runReaderT m x M.≫= k
pushAllBind :: (M.Monad m, M.Monad n)
  ⇒ ((a → n b) → m a → m b)
  → (a → n b) → LockT r m a → LockT r m b
pushAllBind l k m = ReaderT $
  λx → l k $ runReaderT m x
```

In pushBind the monad m is IO in our application, whereas in pushAllBind it is n that is IO. Combining these two functions then allows us to turn a side-effecting function into a function that appends an operation at the end of a concurrent expression:

```
bindIO :: (a → IO b)
  → InternalRep r a → InternalRep r b
bindIO = pushAllBind pushBind
```

For example, if p is a computation returning a value of type a (that is, its type is InternalRep a), then a computation that writes this result to an MVar named l is given by bindIO (putMVar l) p.

Forking two computations requires a similar 'unlifting'.

```
deepFork :: Sync r () → Sync s ()
  → Sync (r :∨: s) ThreadId
deepFork m n = Sync $
  ReaderT $ λp → ReaderT $ λq →
  forkIO (performSync m p q) M.≫
  forkIO (performSync n p q)
```

Tying these together gives the following definition of parallelism:

```
(∥) :: Sync r a → Sync s b
  → Sync (r :∨: s) (a, b)
Sync p ∥ Sync q = do
  la ← liftIO newEmptyMVar
  lb ← liftIO newEmptyMVar
  Sync $ unSync $
    deepFork (Sync $ bindIO (putMVar la) p)
             (Sync $ bindIO (putMVar lb) q)
  x ← liftIO $ takeMVar la
  y ← liftIO $ takeMVar lb
  return (x, y)
```

4 Reconciling Sync Programs and Haskell

We described two full implementations of Sync in Haskell syntax. Some examples work out of the box, like dining philosophers. However, both implementations come with various challenges and shortcomings, mainly due to interaction with Haskell's type system. For all the trouble spots we identified, there are workarounds in Haskell. This demonstrates the power and flexibility of the Haskell type system – we are able to write safe programs without needing the full power of a dependently typed programming language.

Alternation. Haskell branching, if b then m else n, is used under the assumption that the types of m and n are the same. This prevents us from writing programs such as:

```
if True then syncX (syncPutStrLn "a")
        else syncY (syncPutStrLn "b")
```

Our first instinct is to use the sub function to find a common supertype of the two branches – that is, use

```
sub (syncPutStrLn "a")
```

and

```
sub (syncPutStrLn "b").
```

The Haskell type system is not equipped to solve such constraints – and furthermore we are looking for the *least upper bound* of the two grading annotations.

Another attempt at a solution redefines the if-then-else construct using the underlying ifThenElse three argument function (relying on the fact the RebindableSyntax extension is used):

```
ifThenElse :: Bool → Sync r a → Sync s a
  → Sync (r :∨: s) a
```

This redefines the usual setting in which `if` is used, so just redefining `ifThenElse` makes it impossible to write:

```
if True then 2 else 3
```

A possible solution in Haskell is to introduce a new typeclass containing `ifThenElse`, distinguishing whether this particular instance of using the `if` statement combines the lock information or whether it is the conventional setting. This requires using incoherent typeclass instances.

We opt for a simpler approach. We introduce an alternation function, which is the `Sync`-level version of if-then-else that infers the right types:

```
alt :: Bool → Sync r a → Sync s a
  → Sync (r :∨: s) a
alt True x _  = Sync $ unSync x
alt False _ y = Sync $ unSync y
```

The code from before is now written as:

```
alt True (syncX $ syncPutStrLn "a")
         (syncY $ syncPutStrLn "b")
```

Recursion. The interaction of the grading algebra with recursive programs gives rise to problems – the type checker rejects most non-trivial recursive programs. The following simple function `foo` type-checks:

```
foo x = alt (x > 0)
            (foo (x - 1))
            (syncY $ syncPutStrLn "works?")
```

However, the inferred grading is not `["Y"]` – instead, it is some `s` satisfying `(["Y"] :∨: s) ~ s`, where the tilde symbol `~` represents type equality:

```
foo :: (Num t, Ord t, (r :∨: '["Y"]) ~ r) ⟹
  t → Sync r ()
```

What we are looking for is a least solution – that is, the *least fixed point* of the function mapping some lock list `s` to `["Y"] :∨: s`. Since Haskell is unable to perform least fixed point calculation at the type level (it does not support *equirecursive types*), the type of `foo` has to be provided by the programmer.

However, some recursive programs do not even type check. The problems arise because the textual form of the definition of `:∨:` does not enable the type checker to infer that it is associative or idempotent. Consider the following 'repeated bind' operation:

```
repeatBind n m =
  foldl (≫) m (replicate (n - 1) m)
```

The purpose of this function is to produce a computation `m ≫ m ≫ ... ≫ m`, where `m` appears `n` times. However, the type inference algorithm cannot come to the conclusion that `r :∨: r :∨: ... :∨: r` simplifies to `r`, no matter how many occurrences of `r` there are. Even explicitly writing the type of `repeatBind` in the source code cannot make this inference.

The easiest fix is to introduce a new type, which can be used as a constraint:

```
type Idempotent r = (r :∨: r) ~ r
```

Then idempotence of `:∨:` is used as an assumption:

```
repeatBind :: Idempotent r
  ⇒ Int → Sync r a → Sync r a
repeatBind n m =
  foldl (≫) m (replicate (n - 1) m)
```

This allows us to write expressions such as

```
repeatBind 5 philosopher3
```

and Haskell's type system is able to infer that its type is `Sync '["X", "Y"] ()`.

Most such useful properties can be wrapped into types similar to `Idempotent r`, though they might require multiple arguments. For example, a function might rely on associativity of `:∨:`, and such an assumption can be stated using a new type again:

```
type Associative r s t =
  ((r :∨: s) :∨: t) ~ (r :∨: (s :∨: t))
```

Generic code. A more fundamental problem is seen in the dining philosophers example. Since all synchronization functions are defined separately, it is impossible to use them in a 'generic' way. This means that, even if all five philosophers have analogous behaviour, the code has to be written five times. It is not possible to write the following using our definitions so far:

```
philosopher x l1 l2 =
  sync (min l1 l2) $ do
    delay 100000
    sync (max l1 l2) $ do
      delay 100000
      syncPutStrLn $ "philosopher␣" ++ show x
philosopher1 = philosopher 1 "V" "W"
```

Ideally, we want to be able to pass references to locks and use only a single `sync` function. This looks attractive, but requires a dependently typed language – the type of `sync l` depends on the value of `l`. This is exactly what we are trying to avoid with this Haskell implementation.

One way to address this issue is by passing not the region or lock on which to synchronize, but the synchronization function itself. This does not allow the programmer to 'choose the minimum of two locks', but it resolves code duplication. The resulting code is much shorter and requires no additional type annotations:

```
philosopher x s1 s2 = s1 $ do
  delay 100000
  s2 $ do
    delay 100000
    syncPutStrLn $ "philosopher␣" ++ show x
philosopher1 = philosopher 1 syncV syncW
philosopher2 = philosopher 2 syncW syncX
philosopher3 = philosopher 3 syncX syncY
philosopher4 = philosopher 4 syncY syncZ
philosopher5 = philosopher 5 syncV syncX
```

The type system manages to infer the following type for `philosopher`:

```
philosopher :: Show a  ⟹
  a → (Sync s b → t) →
  (Sync '[] () → Sync s b) → t
```

Notice that the programmer has to explicitly encode the order between the locks in the call to `philosopher`. Implementing the 'minimum of two locks' and 'maximum of two locks' is possible and this solution can be taken further. This is done by defining functions `minSync` and `maxSync`, both of which take two synchronization functions and return the appropriate one – for example, `minSync syncX syncY`. Assuming the existence of a type-level function giving the minimum of two locks represented as `Symbol`s, the type signatures of `minSync` is:

```
minSync :: (Sync r a → Sync s a) →
    (Sync r a → Sync t a) →
    Sync r a → Sync (Min s t) a
```

We omit this implementation from this functional pearl, as it comes with its own details and difficulties.

5 Theoretical Foundations

In this section we briefly outline the theoretical background of the implementation described in Section 3. First we formally define the grading algebra (Section 5.1). We then give the necessary theoretical background on graded monads and explain why `Sync` is a graded monad (Section 5.2). Finally we provide some comments on type inference and why it can be performed in polynomial time (Section 5.3).

5.1 The Lattice of Subsets with *error*

Definition 5.1. A partial order $(E, \sqsubseteq)$ is a *lattice* if:

- for any two $x, y \in E$ there exists a (necessarily unique) $z \in E$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$, and for any t such that $x \sqsubseteq t$ and $y \sqsubseteq t$ it is the case that $z \sqsubseteq t$ – this z is the *join* (or *least upper bound*) of x and y and denoted $x \sqcup y$;
- for any two $x, y \in E$ there exists a (necessarily unique) $z \in E$ such that $z \sqsubseteq x$ and $z \sqsubseteq y$, and for any t such that $t \sqsubseteq x$ and $t \sqsubseteq y$ it is the case that $t \sqsubseteq z$ – this z is the *meet* (or *greatest lower bound*) of x and y and denoted $x \sqcap y$.

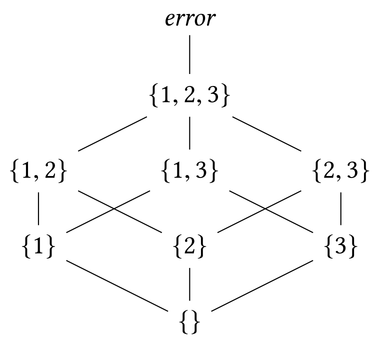

Figure 2. The lock set lattice. Note the *error* top element: every subset of $\{1, 2, 3\}$ is less than *error* in this lattice.

A simple example is the *subset lattice*: for any set U, the partial order $(\mathcal{P}(U), \subseteq)$ is a lattice, where $\mathcal{P}(U)$ is the powerset (set of all subsets) of U. In this case the join is $\cup$ and the meet is $\cap$.

Any lattice $(E, \sqsubseteq)$ with a least element $\bot$ (satisfying $\bot \sqsubseteq x$ for all $x \in E$) gives rise to a monoid with the same carrier set E, whose operator is the lattice join and whose identity is $\bot$ – in other words, $(E, \sqcup, \bot)$ is a monoid. The fact this is a monoid is of special interest to us.[3]

We introduce the 'subsets with *error*' lattice. Given a set U and some object $error \notin U$, define the set $\mathcal{P}_{error}(U)$ as $\mathcal{P}(U) \cup \{error\}$. Furthermore, define the partial order $\sqsubseteq$ with just two rules:

- $x \sqsubseteq error$ for all $x \in \mathcal{P}_{error}(U)$;
- $x \sqsubseteq y$ for all $x, y \in \mathcal{P}(U)$ such that $x \subseteq y$.

Then $(\mathcal{P}_{error}(U), \sqsubseteq)$ is a lattice. This is not difficult to establish. It is known that $(\mathcal{P}(U), \sqsubseteq)$ is a lattice, and furthermore adding a top ($\top$) element to a lattice yields a lattice. Figure 2 shows the Hasse diagram of this lattice when $U = \{1, 2, 3\}$.

The reason for introducing *error* is to introduce a total function that is analogous to `PartialCons`: adding an item that violates an assumption about the order in which they need to be acquired should result in failure. While in Haskell we aim to make the type checker explicitly reject the expression, here we use *error* to allow all functions to be total. Assume $(U, <)$ is a total order, e.g. locks with their ordering. Then we define the function $add : U \times \mathcal{P}_{error}(U) \longrightarrow \mathcal{P}_{error}(U)$ as follows:

$$add(x, S) = \begin{cases} \{x\} \cup S, & S \neq error \text{ and } \forall y \in S. x < y \\ error, & \text{otherwise} \end{cases}$$

For example, $add(1, \{2\}) = \{1, 2\}$ and $add(2, \{1, 3\}) = error$.

All of this could be alternatively presented via an algebra of subsequences of $[1, 2, \ldots, n]$, where union can instead be represented as a *smallest common supersequence* operator.

[3]This is also the reason why we use the letter E – it is commonly used in literature when grading algebras are discussed. By contrast, lattice literature typically uses letters D and P.

5.2 Graded Monads

We define the notion of graded monads,[4] and then show that `Sync` is a graded monad.

Definition 5.2. For a partially ordered monoid $((E, \sqsubseteq), \star, i)$ such that $\star$ is monotone with respect to $\sqsubseteq$, a *graded monad* T is given by:

- a set of type constructors T^r, where $r \in E$;
- a polymorphic function $\mathsf{return} : \alpha \to T^i \alpha$;
- for any $r, s \in E$, a left-associative polymorphic function $\gg\!=_{r,s} : T^r \alpha \to (\alpha \to T^s \beta) \to T^{r \star s} \beta$
- for any $r, r' \in E$ satisfying $r \sqsubseteq r'$, a polymorphic function $\mathsf{sub}_{r,r'} : T^r \alpha \to T^{r'} \alpha$

satisfying the following properties:

- $m \gg\!=_{r,i} \mathsf{return} \equiv m$ for all m of type $T^r A$;
- $\mathsf{return}\ x \gg\!=_{i,r} k \equiv k\ x$ for x of type A and k of type $A \to T^r B$;
- $(m \gg\!=_{r,s} k) \gg\!=_{s,t} h \equiv m \gg\!=_{r,s} \lambda x.(k\ x \gg\!=_{s,t} h)$ for all m of type $T^r A$, k of type $A \to T^s B$ and h of type $B \to T^t C$;
- $\mathsf{sub}_{r,r} \equiv \lambda x.x$ for all $r \in E$;
- $\mathsf{sub}_{s,t} \circ \mathsf{sub}_{r,s} \equiv \mathsf{sub}_{r,t}$ for all $r, s, t \in E$ such that $r \sqsubseteq s \sqsubseteq t$;
- $\mathsf{sub}_{r\star s, r'\star s'}(m \gg\!=_{r,s} k) \equiv \mathsf{sub}_{r,r'}\ m \gg\!=_{r',s'} (\mathsf{sub}_{s,s'} k)$ for any $r, r', s, s' \in E$ satisfying $r \sqsubseteq r'$ and $s \sqsubseteq s'$.

The first two properties are very similar to the three monad laws, but with the addition of the associativity law (third property) relying on the associativity of the $\star$ operator of the monoid. The latter three properties show how reflexivity of $\sqsubseteq$, transitivity of $\sqsubseteq$ and monotonicity of $\star$ with respect to $\sqsubseteq$ are 'lifted' and form the basis of subtyping. The $\mathsf{sub}_{(-),(=)}$ family of functions can be seen as explicit casting or type coercion operations.

The `Sync` structure introduced earlier almost matches the Haskell definition of graded monad, but its sequencing operator is not total. To model this partiality, we use the grading algebra defined in Section 5.1 (with *error* representing 'undefined result'). Instead we focus on an almost identical Sync graded monad – the underlying data representation is the same, the change is that the grading algebra is well-defined (that is, the $\star$ operator is total). Replacing the list representation with the $\mathcal{P}_{error}(U)$ (with $\sqcup$ as the monoid operator $\star$) one only requires one main change: the types of synchronization functions. Now the type of `syncX` is $\mathsf{Sync}^r \alpha \to \mathsf{Sync}^{add(x,r)} \alpha$ for all r, and likewise for all the other synchronization functions. It can be shown that Sync satisfies all of the axioms of a graded monad.

The grading algebra we use is special and simple in several ways. Firstly, the $\star$ (here $\sqcup$) operator is commutative, which is not generally the case for effect systems. Commutativity is expected due to the fact that the locking primitives are structured – for example, `(syncX e1) ≫ (syncX e2)` has the same type as `(syncX e2) ≫ (syncX e1)`, as lock `X` is released before the second statement acquires it. Secondly, sequencing happens to coincide with the join operator, which resembles Lucassen-Gifford effect systems, but is not a requirement on grading algebras. Neither of these assumptions hold if the locking primitives are richer. The grading algebra is simple, but limits the kinds of programs we can write.

[4]The definition is not the most general: these are graded monads in the context of Haskell, not for arbitrary categories.

5.3 Type Inference

We hinted previously that the type-inference algorithms of Haskell cannot compute the types of all values – particularly the recursive ones. However, due to monotonicity of $\sqcup$ and *add* (Theorems 5.6 and 5.7), it is in principle possible to infer types with a different algorithm.

One perspective is that each expression in a program generates typing constraints involving it and its possible subexpressions. For example:

- if m has type $\mathsf{Sync}^r A$ and k has type $A \to \mathsf{Sync}^s B$, then the expression $m \gg\!= k$ has type $\mathsf{Sync}^{r \sqcup s} B$ (recall that $r \sqcup s = r \star s$);
- if m has type $\mathsf{Sync}^r A$, n has type $\mathsf{Sync}^s A$, then the expression $\mathsf{if}\ b\ \mathsf{then}\ m\ \mathsf{else}\ n$ has type $\mathsf{Sync}^{r \sqcup s} A$;
- if m has type $\mathsf{Sync}^r A$, then the expression `syncX` m has type $\mathsf{Sync}^{add(x,r)} A$.

Thus, if every expression wrapped inside the Sync monad is assigned a grade from $r_1, \ldots, r_k$, we end up with constraints such as:

$$r_1 = \{\}, \quad r_2 = r_1, \quad r_3 = r_1 \sqcup r_4, \quad r_4 = add(x, r_2)$$

Every grade has an associated equation where it appears on the left-hand side. All of these constraints can be seen as representing a single equation of the form:

$$(r_1, \ldots, r_k) = g(r_1, \ldots, r_k)$$

for some function $g : \mathcal{P}_{error}(U)^k \to \mathcal{P}_{error}(U)^k$ based on the generated constraints. The solution of this equation is a fixed point of g. Moreover, since we are looking for the 'most precise' grade, it is the *least fixed point* of f with respect to the partial order $\sqsubseteq^k$.

For example, consider the following expression:

```
bar x = alt (x > 0)
  ((syncX $ syncPutStrLn (show x)) ≫ bar (x - 1))
  (syncY $ syncPutStrLn "?")
```

Suppose the grade associated with `bar x` is r_1, the grade associated with the recursive case is r_2, and the grade associated with the base case of the recursion (or the 'else branch') is r_3. The generated constraints are:

$$r_1 = r_2 \sqcup r_3, \quad r_2 = \{X\} \sqcup r_1, \quad r_3 = \{Y\}$$

The least solution of this equation is $r_1 = r_2 = \{X, Y\}$, $r_3 = \{Y\}$.

Finding the least fixed point is not difficult. It relies on a few key properties of the lattice and of the operations introduced in Section 5.1.

Definition 5.3. A lattice $(P, \sqsubseteq_P)$ is *complete* if every ascending chain $p_0 \sqsubseteq_P p_1 \sqsubseteq_P \dots$ has a least upper bound $\bigsqcup_i p_i$ that is in P.

Definition 5.4. Given two complete lattices $(P, \sqsubseteq_P)$ and $(Q, \sqsubseteq_Q)$, denote least upper bounds of ascending chains $p_0 \sqsubseteq_P p_1 \sqsubseteq_P \dots$ in P and $q_0 \sqsubseteq_Q q_1 \sqsubseteq_Q \dots$ in Q as $\bigsqcup_i p_i$ and $\bigsqcup_i q_i$, respectively. We say that a function $f : P \longrightarrow Q$ is *Scott-continuous* if for every ascending chain $p_0 \sqsubseteq_P p_1 \sqsubseteq_P \dots$ in P, we have

$$f\left(\bigsqcup_i p_i\right) = \bigsqcup_i f(p_i)$$

Theorem 5.5. *$(\mathcal{P}_{error}(U), \sqsubseteq)$ is a complete lattice.*

Theorem 5.6. *The join operator $\sqcup$ is Scott-continuous (in both of its arguments).*

Theorem 5.7. *The add function is Scott-continuous in its second argument.*

A consequence of this is that the function g introduced above is also Scott-continuous.

Finally, the following theorem provides a way to compute the least fixed point:

Theorem 5.8 (Kleene). *Let $(P, \sqsubseteq_P)$ be a lattice with a least element $\bot$ and let $f : P \longrightarrow P$ be Scott-continuous function. Then the least fixed point of g is given by $\bigsqcup_i f^i(\bot)$. [10]*

The algorithm that can be used to compute the grades of each expression is a simple iteration:

1. Initialise $r_1 = r_2 = \dots = r_k = \{\}$.
2. Compute $(s_1, \dots, s_k) = g(r_1, \dots, r_k)$.
3. If $(s_1, \dots, s_k) = (r_1, \dots, r_k)$, return $(r_1, \dots, r_k)$.
4. Otherwise, assign $(s_1, \dots, s_k)$ to $(r_1, \dots, r_k)$ and repeat 2.

This algorithm necessarily terminates because g, being Scott-continuous, also is monotonic in every argument, and the $(\mathcal{P}_{error}(U)^k, \sqsubseteq^k)$ lattice has a finite height – so there will be a finite number of iterations before a fixed point is reached. By Theorem 5.8, the returned tuple $(r_1, \dots, r_k)$ is the least fixed point of g.

The worst-case running time of this algorithm is polynomial in the number of locks l. Updating each r_i takes constant time, since it is either an application of *add* or $\sqcup$ or the identity function to one or two of the other grades being inferred. Hence each update iteration takes $O(k)$ time. Consider any chain in the $(\mathcal{P}_{error}(U)^k, \sqsubseteq^k)$ lattice from the bottom element $(\{\}, \{\}, \dots, \{\})$ to the top element $(\top, \top, \dots, \top)$. For two successive lattice elements on that path, lattice elements, $(S_1, \dots, S_k) \sqsubseteq^k (S'_1, \dots, S'_k)$, there is least one i $(1 \le i \le k)$ such that $S_i \neq S'_i$. Since every step in $\mathcal{P}_{error}(U)^k$ corresponds to at least one step in one of the k $\mathcal{P}_{error}(U)$ lattices, the height[5] of $(\mathcal{P}_{error}(U)^k, \sqsubseteq^k)$ does not exceed $k(l+1)$. This is equal to the worst-case number of iterations performed. Thus the running time of this algorithm is $O(k^2(l+1))$.

6 Related Work

Session types. Honda et al. [8] introduced *session types* for π-calculus, which constrain the kinds of interaction and communication allowed between concurrent threads. A well-typed program ensures that two potentially communicating processes have compatible communication patterns. There have been several implementations of session types in Haskell as monad generalisations, for example by Orchard and Yoshida [16], Sackman and Eisenbach [19], Pucella and Tov [17]. Our approach lacks the message-passing capabilities of π-calculus, but it makes thread-safety analysis much easier.

Regions and capabilities. One of the earliest applications of effect systems was for concurrency and memory regions – see Tofte and Talpin [21]. Crary et al. [2] introduced the *Capability Calculus*, a compiler intermediate language with region-based memory management. This language features first-class memory regions – a new region can be allocated using a form of `let`. These regions are deallocated after the context which created them finishes, meaning that this is a paradigm that is an alternative to garbage collection. The Capability Calculus is type safe in the sense that there are no memory leaks. The inference system holds information about which regions are available to an expression as part of the context. Similar capability-based systems for concurrency exist – see Castegren's thesis [1] for an overview.

Gerakios et al. [4] consider a more expressive capability-based type system in which locks are lexically scoped and there is no assumed order between them. The deadlock avoidance strategy is based on reference counting and tracking the order of synchronization operations. The language they introduce uses `lock` and `unlock` primitives, as well as `share` and `release` which track reference counts. The typing relation uses type-and-effect judgements of the form $M; \Delta; \Gamma \vdash e : \tau \& (\gamma; \gamma')$, where $M; \Delta; \Gamma$ is the typing context (with additional information about permissions), e is an expression in the language, τ is the type attributed to e, and γ and γ' are *input* and *output effects*. The effects are drawn from the set of ordered lists that represent sequences of operations (such as locking) on references. It is unknown whether their type system can be embedded in Haskell through graded monads, and perhaps an interesting avenue of future work.

There are other approaches that do not assume a fixed order. For example, Kobayashi [11] considers a more expressive type system for π-calculus. Suenaga [20] uses the idea of *lock levels* without an explicit lock order. Pun et al. [18] use a

[5] By 'height' we mean the maximum number of steps of an increasing chain in a lattice, not the number of elements of this chain.

two-step process: a effect system yields a transition system representing program execution, after which the system is explored at the global level to detect whether the program might deadlock.

Gordon et al. [6] consider a problem and framework similar to ours. Their approach is based on *lock capabilities* – static capabilities that permit acquiring additional locks and are *inherited*. They do not assume a fixed total order – instead, they generalise to a forest-shaped partial order.[6] The approach is type-directed and uses type judgements of the form $\Upsilon; \Gamma; L \vdash e : \tau; \Upsilon'$, where Υ and Υ' represent 'tree disjointness' in the partial order, Γ is the typing context for local variables, and L is the set of held locks. By contrast, our approach does not add this information to the context and all information is wrapped inside the annotation of a graded monad.

Furthermore, Gordon [5] also uses a lattice with $\top$ representing an error value. The same paper also considers a synchronization framework that more resembles mutual exclusion locks with explicit locking and unlocking primitives.

7 Conclusions

We introduced a set of simple concurrency primitives that allow the programmer to write code inside a graded monad that is guaranteed to be deadlock-free. The simplest implementation, which allows us to write a safe dining philosophers solution, uses a list of mutual exclusion locks stored as `MVars` holding unit values (where empty means that the lock is not acquired, and full means it is). An extension that introduces support for memory regions is based on monad transformers and it generalises the above use of `MVars` to protect data more general than `()`. The Haskell type system is able to infer the types of a significant number of applications, but requires programmer annotations in cases of most recursive values and code that relies on properties of the grading algebra. We have shown that, with added support for equirecursive types, a Kleene-style iteration can perform full inference.

Further work. The main limitation of our approach is that it assumes a fixed pre-existing total order on the set of locks. This is sufficient, but not necessary for deadlock detection. All that is needed is for different parts of the program, those guaranteed not to be run concurrently, assume a partial order between locks (this makes the example in Section 2.2 type check). A way to resolve this is by inferring a *happens-before* relation on the lock set during type inference and type-checking, and then rejecting if the inferred constraints are inconsistent. This happens-before relation can be represented as a graph, and it is inconsistent if and only if the graph has a cycle. We will explore various applications of graph effect algebras, including concurrency, in a future publication.

[6]This means that the partial order $\sqsubseteq$, when visualised as a Hasse diagram, looks like a forest of trees – there is a 'unique path to root' for every element of the set.

The structured locking primitive we used (`syncX`, `syncY`, ...) is not nearly general – most programs cannot be translated into this framework (for example, producer-consumer programs). However, the provided constructs can be seen as an intermediate step towards deadlock-free Haskell monitors, which provide more expressive concurrency primitives. We intend to investigate the degree to which the implementation in this pearl can be generalised to implement monitors.

There are other limitations of our approach that provide opportunities for further work. For example, there is no abstraction over locks, which we discussed in Section 4. More significantly, fine-grained locking is impossible – for example, a thread-safe binary search tree implementation might require associating a lock with each node or subtree. It is not clear how these issues can be resolved in Haskell without dependent types.

Acknowledgments

The first author is funded by Trinity College, University of Cambridge, through its Internal Graduate Scholarship.

We thank Michael Gale for his encylopaedic knowledge of Haskell syntax extensions.

References

[1] Elias Castegren. 2018. *Capability-Based Type Systems for Concurrency Control.* Ph.D. Dissertation. Uppsala UniversityUppsala University, Division of Computing Science, Computing Science.

[2] Karl Crary, David Walker, and Greg Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275. https://doi.org/10.1145/292540.292564

[3] Cormac Flanagan and Stephen N. Freund. 2000. Type-Based Race Detection for Java. *SIGPLAN Not.* 35, 5 (May 2000), 219–232. https://doi.org/10.1145/358438.349328

[4] Prodromos Gerakios, Nikolaos Papaspyrou, and Konstantinos Sagonas. 2010. A Type System for Unstructured Locking that Guarantees Deadlock Freedom without Imposing a Lock Ordering. In *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010 (EPTCS, Vol. 69)*, Kohei Honda and Alan Mycroft (Eds.). 44–58. https://doi.org/10.4204/EPTCS.69.4

[5] Colin S. Gordon. 2017. A Generic Approach to Flow-Sensitive Polymorphic Effects. In *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain (LIPIcs, Vol. 74)*, Peter Müller (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 13:1–13:31. https://doi.org/10.4230/LIPIcs.ECOOP.2017.13

[6] Colin S Gordon, Michael D Ernst, and Dan Grossman. 2012. Static lock capabilities for deadlock freedom. In *Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation.* 67–78.

[7] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 48–60. https://doi.org/10.1145/

1065944.1065952

[8] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. 1998. Language primitives and type discipline for structured communication-based programming. In *Programming Languages and Systems*, Chris Hankin (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 122–138.

[9] 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

[10] Stephen Cole Kleene, NG De Bruijn, J de Groot, and Adriaan Cornelis Zaanen. 1952. *Introduction to metamathematics*. Vol. 483. van Nostrand New York.

[11] Naoki Kobayashi. 2006. A New Type System for Deadlock-Free Processes. In *CONCUR 2006 – Concurrency Theory*, Christel Baier and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–247.

[12] J. M. Lucassen and D. K. Gifford. 1988. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 47–57. https://doi.org/10.1145/73560.73564

[13] R. Milner. 1982. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg.

[14] Robin Milner, Joachim Parrow, and David Walker. 1992. A calculus of mobile processes, I. *Information and Computation* 100, 1 (1992), 1–40. https://doi.org/10.1016/0890-5401(92)90008-4

[15] Dominic Orchard and Tomas Petricek. 2014. Embedding effect systems in Haskell. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell*. 13–24.

[16] Dominic Orchard and Nobuko Yoshida. 2016. Effects as Sessions, Sessions as Effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 568–581. https://doi.org/10.1145/2837614.2837634

[17] Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 25–36. https://doi.org/10.1145/1411286.1411290

[18] Ka I. Pun, Martin Steffen, and Volker Stolz. 2012. Deadlock checking by a behavioral effect system for lock handling. *The Journal of Logic and Algebraic Programming* 81, 3 (2012), 331 – 354. https://doi.org/10.1016/j.jlap.2011.11.001 The 22nd Nordic Workshop on Programming Theory (NWPT 2010).

[19] Matthew Sackman and Susan Eisenbach. 2008. Session types in Haskell: Updating message passing for the 21st century. (2008). https://spiral.imperial.ac.uk/bitstream/10044/1/5918/1/session-types-in-haskell.pdf

[20] Kohei Suenaga. 2008. Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References. In *Programming Languages and Systems*, G. Ramalingam (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–170.

[21] Mads Tofte and Jean-Pierre Talpin. 1997. Region-Based Memory Management. *Inf. Comput.* 132, 2 (Feb. 1997), 109–176. https://doi.org/10.1006/inco.1996.2613

[22] Philip Wadler and Peter Thiemann. 1999. The Marriage of Effects and Monads. *ACM Transactions on Computational Logic* 4 (12 1999). https://doi.org/10.1145/601775.601776