Safe Mutation with Algebraic Effects

Hashan Punchihewa
Imperial College London
United Kingdom
hashan.punchihewa17@imperial.ac.uk

Nicolas Wu
Imperial College London
United Kingdom
n.wu@imperial.ac.uk

Abstract

It can be difficult to write safe concurrent programs which use shared mutable state. Subtle mistakes can lead to data races that manifest as unexpected program behaviour. The prevailing approaches to solving this dilemma are to either eschew mutable state altogether, or design bespoke languages that prevent data races by design. This article introduces a third approach by showing how safe mutation can be integrated into a mainstream functional programming language with algebraic effects. This article produces a framework that tracks the use of mutable state and guarantees data race freedom at compile time.

CCS Concepts: • **Theory of computation → Type structures**; **Program verification**; *Program analysis*; *Pre- and post-conditions.*

Keywords: parameterised algebraic effects, type-checker plugin, concurrency, data race

ACM Reference Format:
Hashan Punchihewa and Nicolas Wu. 2021. Safe Mutation with Algebraic Effects. 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/10.1145/3471874.3472988

1 Introduction

Concurrency can yield significant improvements in performance, but can also very easily lead to subtle mistakes in program correctness. As the complexity of a concurrent program increases, there is a combinatorial explosion in the number of ways for threads to be ordered, which makes reasoning about concurrent programs challenging.

This is especially true, when threads share mutable state, as this can lead to a *data race*. This is where two threads independently access the same memory location, with at least one of the accesses being a write, and no synchronisation mechanism controlling the ordering of the accesses. It is not possible for mainstream programming languages to automatically prevent the simplest of data races.

```
transfer sender recipient amount = do
  v_sender    ← read sender
  v_recipient ← read recipient
  write sender    (v_sender − amount)
  write recipient (v_recipient + amount)

racyBank = do
  alice ← alloc £10
  bob   ← alloc £10
  fork (transfer alice bob £5)
  fork (transfer alice bob £5)
```

Consider the function racyBank, which type-checks in Haskell. This function opens up bank accounts for Alice and Bob, each with an initial deposit of £10. Then two transfers are initiated from Alice to Bob, each with a value of £5. While the intended outcome of these transfers is for Alice to end up with £0 and Bob to end up with £20, there are three other outcomes, that could arise depending on the interleaving of the threads:

- If both threads perform read sender before performing the corresponding write operation, then Alice will end up with £5.
- If both threads perform read recipient before performing the corresponding write operation, then Bob will end up with £15.
- Both of these circumstances could arise, leading to Alice ending up with £5 and Bob ending up with £15.

One approach to addressing data races has been to use new forms of concurrency, which make data races impossible. In Haskell, this has led to the development of software transactional memory (STM) [19], which allows arbitrary mutations to shared state to be combined and performed atomically. In other languages, this has led to the development of the actor model, where there is no shared state at all and actors co-ordinate by passing immutable data [1, 16].

However, despite these efforts shared state remains very popular. Transactional memory is heavily dependent on the support of a runtime system, to manage the program's interactions with memory. The actor model requires writing code that is oriented around the primitives of actors and behaviours. While actors avoid data races, they are susceptible to other classes of errors [45].



This work is licensed under a Creative Commons Attribution 4.0 International License.

Haskell '21, August 26–27, 2021, Virtual, Republic of Korea
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8615-9/21/08.
https://doi.org/10.1145/3471874.3472988

An alternative approach to prevent data races has focused on improving the power of type systems [6, 9, 10]. This work predominantly focuses on object-oriented systems, and enhances the type system to track the use of mutable state. Under such a type system, the compiler verifies at compile time, that a data race cannot occur.

The goal of this paper is to show how these ideas can be implemented in a principled way, to track instances of mutable state and verify that data races cannot occur, using effect systems and type-level computation. To this end, we provide an implementation in Haskell, building on prior literature that shows how advanced effect systems can be implemented using the theory of algebraic effects [24, 41]. The key insight of this paper is to extend these effect systems, using the notion of a parameterised algebraic effect, to encode the idea of a restriction on a resource.

In the system developed here, the example above would lead to the following error.

```
Bank.lhs:86:23: error:
    • You could be writing to a resource,
      you have no access to.
    • In a stmt of a 'do' block:
        fork (transfer alice bob (pounds 5))
      In the expression:
        do alice <- alloc (pounds 10)
           bob   <- alloc (pounds 10)
           fork (transfer alice bob (pounds 5))
           fork (transfer alice bob (pounds 5))
      In an equation for ‘racyBank’:
        racyBank
            = do alice <- alloc (pounds 10)
                 ....
   |
86 |  fork (transfer alice bob (pounds 5))
   |        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Haskell is able to identify the call to transfer as the problem in the second child thread, since it conflicts with the call to the same function in the first child thread. Haskell is able to identify that, the problem does not lie in transfer per se, but in how it is used.

This work draws on a rich body of interrelated ideas. One such idea is region-based memory management [43, 44], where the lifetime of objects is analysed to control memory allocation; this is closely connected to research to track which effects a fragment of code might interact with [31]. In addition, there is a link to the idea of a capability, which is a token that grants access to a resource, developed in systems research [15, 26, 27, 30]; this is very similar to the idea of a coeffect, which places restrictions on the context in which a computation can occur [37]. A specific instance of coeffects is linear types, where values can only be used once [18, 46].

First, we introduce background material on algebraic effects (§2). Then, the specific contributions of this paper are:

- The construction of a model of concurrency, that disallows data races (§3). This is achieved by taking advantage of parameterised algebras, and type-level computation.
- A further extension of this model to support recursive array algorithms, by allowing an array to be split into two contiguous subarrays, that can be used by distinct threads (§4).
- A type-checker plugin (§5), which improves type inference, by augmenting the Haskell type-checker with tailor-made rules to handle the constraints that arise from the library developed.

We conclude with a discussion on related work (§6) and future research directions (§7).

The source code for this work is available at https://github.com/hashanp/safe-mutation.

2 Background

2.1 Algebraic Effects

In this section, algebraic effects will be introduced to encode the syntax of the mutable store effect. Algebraic effects [25, 41] are a way of encoding effects in functional programming languages.

In Haskell, an algebraic effect is encoded as a functor, whose data constructors are the operations of the effect. As an example, the Store functor is given below with the operations Alloc, Read and Write. It represents an effect for mutable data stores. The Alloc operation takes an initial value of type a, and produces as output Token a, a reference to a new mutable store containing a value of type a.

```
data Store a where
  Alloc :: a → Store (Token a)
  Read  :: Token a → Store a
  Write :: Token a → a → Store ()
```

This functor can be combined with the freer monad [23], given by Prog, to build up the syntax of the algebraic effect. The Return constructor is used to represent pure computations, and the Call constructor is used to represent impure computations.

```
data Prog f a where
  Return :: a → Prog f a
  Call   :: f b → (b → Prog f a) → Prog f a
```

The data type Prog Store a represents computations involving the store effect, and which yield a value of type a.

```
read s    = Call (Read s return)
write s v = Call (Write s v return)
```

Smart constructors for effects, such as read and write, are defined for convenience.

```
increment s = do
  v ← read s
  write s (v + 1)
```

A program that increments a mutable store is given above. This takes a mutable store s and performs the read operation to retrieve its value v. It then uses the write operation to store v + 1 in the store.

Thus, algebraic effects can be used to implement the mutable store effect. A shortcoming of ordinary algebraic effects is that it cannot model an operation such as fork which constructs a child thread. This is because of the algebraicity property that gives rise to such effects: the operations of an algebraic effect must commute with bind. This is not the case with fork, where the equivalence fails to hold for a program p and continuation k:

$$(\text{fork p}) \gg\!= \text{k} \not\equiv \text{fork (p} \gg\!= \text{k)}$$

This equivalence cannot hold. In the expression on the left-hand side, the continuation k runs in the parent thread, while in the expression on the right-hand side, the continuation k runs in the child thread.

2.2 Scoped Algebraic Effects

A noteworthy enhancement to the expressive power of algebraic effects, that addresses this shortcoming, is scoped effects (or higher-order effects). A scoped operation is an operation which changes the behaviour of subsequent operations. The precise set of operations affected by a scoped operation is called the inner scope.

Prior literature has shown that it is not enough to allow the handlers of algebraic effects to delimit the scope of an effect, as this breaks the separation between syntax and semantics, a core attribute of algebraic effects [40, 47]. Instead the scope entailed by a scoped operation should be delimited by syntax.

```
data Prog f g a where
  Return :: a → Prog f g a
  Call   :: f b → (b → Prog f g a) → Prog f g a
  Enter  :: Prog f g a → g a b → (b → Prog f g c)
            → Prog f g c
```

Scoped operations are incorporated into the syntax, through the Enter data constructor, which takes the inner scope, the scoped operation and the continuation as parameters.

The scoped operations are represented by their own functor g, indexed by two type parameters: one represents the result of the computation within the inner scope; the other represents the result of the scoped operation accessible to the outer scope.

```
data Thread a b where
  Fork :: Thread () ()
```

For the fork operation, the Thread functor will be used, with both type parameters as unit, although this will be revisited later. The syntax for the state effect will now be Prog Store Thread a. Note that smart constructors can also be defined for scoped operations.

```
fork p = Enter Fork p return
```

To conclude this section, a standard algebraic effect implementation with freer monads has been given. This provides the background to understand the racyBank example in the introduction, which despite containing a data race, can be encoded in the mutable store effect developed so far. This motivates the next section, which introduces the idea of constraining reads from and writes to mutable stores.

3 Development

3.1 Type-Level State

So far, the syntax of a mutable store effect with a scoped fork operation for concurrency has been introduced. However, this syntax does not prevent programs that lead to data races. In this section, a high-level model will be given showing how associating type-level state with mutable stores can be used to prevent data races.

A read from a mutable store is considered safe, if there are no other threads that could be writing to the store concurrently. Whereas a write to a store is considered safe, if there are no other threads accessing the store at all. In other words, there can be multiple threads that can access the store, as long as they're all reading. However, if a thread is writing, then it must be the only thread that can access the store.

We model this situation using three *access levels*:

- If a thread has access level N to a store, then a thread cannot perform any operation at all.
- If a thread has access level R to a store, it is only safe to perform reads, because there could be other threads that have access to the store.
- If a thread has an access level X, it is safe to perform read and writes to the store, as it has exclusive access to the store.

These access levels are formulated in the following type.

```
data AccessLevel = N | R | X
```

These access levels can be thought of as ordered with $\text{N} \leqslant \text{R} \leqslant \text{X}$, and this relationship can be formulated with the following typeclass.

```
type (⩽) :: AccessLevel → AccessLevel → Constraint
class a ⩽ b
instance a ⩽ X
instance R ⩽ R
instance N ⩽ R
instance N ⩽ N
```

To illustrate how this might be used, consider the increment example. Suppose the access level to the store s is given by a. The read operation generates the constraint $\text{R} \leqslant \text{a}$ and the

write operation generates the constraint X ⩽ a. Note that the constraint R ⩽ a is subsumed by the constraint X ⩽ a.

```
increment s = do
  v ← read s
  write s (v + 1)
```

In our model, new threads will be constructed using a fork operation. This means the access level can actually change after a fork operation, since a thread might have access to a resource, but then lose it to a child thread.

In the example below the parent thread begins with exclusive access to the store, but then creates a child thread. This child thread performs a write operation, meaning that it requires exclusive access to the store. This means that after the fork operation, the parent thread has no further access to the store.

```
goodState = do
  s ← alloc True
  write s False
  fork do
    write s False
  return ()
```

To capture the relationship between the access level of a parent thread before fork, the access level of a parent after fork and the access level of the child thread, a typeclass Share is used. This controls how access is shared between the parent and child after the fork operations.

```
type Share :: AccessLevel → AccessLevel → AccessLevel
  → Constraint
class Share a b c
instance Share X X N
instance Share X N X
instance Share X R R
instance Share R R R
instance Share N N N
```

The fork operation in the goodState program generates the constraint Share X X N which can be satisfied. Consider a variation of goodState, called badState.

```
badState = do
  s ← alloc True
  fork do
    write s False
  write s False
  return ()
```

Notice that the write operation is performed both in the child thread, and the parent thread after the fork. This generates the constraint Share X X X which cannot be satisfied. Note that in certain circumstances, there might be multiple possible values for Share a b c.

```
ambiguousState = do
  s ← alloc True
  fork do
    return ()
  return ()
```

Consider ambiguousState, where the mutable store s isn't used afterwards. Then Share X X N or Share X N X could be used to satisfy the Share constraint for fork. How ambiguous cases and type inference interact will be discussed later.

3.2 Parameterised Algebraic Effects

In the previous section, it was shown that data races can be prevented by associating type-level states with mutable stores. However, this was only shown at a high-level, and how to achieve this concretely in Haskell has not yet been explained. This section introduces parameterised algebraic effects, which provide the underlying infrastructure to make this possible.

Parameterised computation was introduced by [4, 5], allowing computations to be parameterised by objects, that represent the change in state that occurs as a result of computation.

First let us define a parameterised functor, with parameterising category S, as a functor $T : S^{\mathrm{op}} \times S \times C \to C$. The objects of the parameterising category can be seen as predicates on states. The first parameter is called the precondition and the second parameter is the postcondition.

The object $T(P, Q, X)$ is used to represent an effectful computation that starts in the state that satisfies P and ends in the state that satisfies predicate Q. An arrow $q : Q \to Q'$ in the parameterising category, represents a weakening of the predicate Q. Hence there is an arrow $T(P, Q, X) \to T(P, Q', X)$, that represents weakening the postcondition of the computation, given by $T(P, q, X)$. Similarly, an arrow $p : P \to P'$ in C represents a strengthening of a predicate P'. There is an arrow $T(P', Q, X) \to T(P, Q, X)$ that represents strengthening the precondition, given by $T(p, Q, X)$.

An encoding into Haskell is given with the IFunctor typeclass. Intuitively, the parameterised imap function means that if you can reach type b from type a, and you have a computation that arrives at a with precondition i and postcondition j, then you can obtain a computation that arrives at b with the same pre and postcondition.

```
type IFunctor :: (p → p → Type → Type) → Constraint
class IFunctor f where
  imap :: (a → b) → f i j a → f i j b
```

A parameterised monad T, whose parameterising category is S, is a parameterised functor T, equipped with two natural transformations, $\mathrm{ireturn}_{S,A} : A \to T(S, S, A)$ as well as $\mathrm{ijoin}_{S_1,S_2,S_3,A} : T(S_1, S_2, T(S_2, S_3, A)) \to T(S_1, S_3, A)$. The ireturn transformation allows you to lift values into computations, and the ijoin transformation allows you to sequence

computations. Since a value doesn't actually perform computation, the indices for the ireturn transformation stay the same, reflecting that there is no change in state. The ijoin transformation requires that the postcondition of the first computation and the precondition of the second computation be the same.

```
type IMonad :: (p → p → Type → Type) → Constraint
class (IFunctor m) ⇒ IMonad m where
  ireturn :: a → m i i a
  ijoin   :: m i j (m j k a) → m i k a
  ibind   :: m i j a → (a → m j k b) → m i k b
```

An encoding into Haskell is given with the IMonad typeclass[1]. In the same way the ordinary Monad typeclass has the ≫= function for composition, a IMonad equivalent, ibind can be defined in terms of ijoin and imap:

```
ibind p k = ijoin (imap k p)
```

Parameterised computation can also be extended to algebraic effects. In the same way that ordinary algebraic effects can be implemented with monads in Haskell, parameterised effects can be implemented with parameterised monads. The functor representing non-scoped operations is extended with two type parameters p and q representing the pre and postcondition of the operation.

```
data IProg f p q a where
  Return :: a → IProg f p p a
  Call   :: f p q b → (b → IProg f q r a) → IProg f p r a
```

To make IProg concrete, consider the contrived Firewall effect, whose parameterised state is either Active or Inactive, and which consists of two operations Enable and Disable to toggle this state.

```
data Status = Active | Inactive
type Firewall :: Status → Status → Type → Type
data Firewall p q x where
  Enable :: Firewall Inactive Active ()
  Disable :: Firewall Active Inactive ()
```

The program lowerDefences begins expecting the firewall to be active, and ends with the firewall inactive, and therefore has type IProg Firewall Active Inactive.

```
lowerDefences :: IProg Firewall Active Inactive
lowerDefences = do
  print "Lowering Defences"
  print "Prepare for Malware"
  disable
```

Extending this to scoped operations is more involved. There are four type parameters to consider: the state before entering the inner scope p, the state after entering the inner scope p′, the state before exiting the inner scope q′, and the state after exiting the outer scope q. This means the type representing scoped operations is extended with four additional type parameters.

```
data IProg f g p q x where
  Return :: a → IProg f p p a
  Call   :: f p q b → (b → IProg f q r a) →
            IProg f p r a
  Enter  :: IProg f g p′ q′ a → g p p′ q′ q a b →
            (b → IProg f g q r c) → IProg f g p r c
```

To conclude, this section has introduced parameterised algebraic effects, which allow us to define a richer syntax to encode computations. The type-level state associated with mutable stores can be stored as the pre and postconditions of parameterised algebraic effects, allowing the use of mutable stores to be tightly constrained to prevent data races.

3.3 Type-Level State Revisited

In this section, a parameterised version of the store effect will be introduced, that does not allow programs with data races. The parameterised state for this effect will be a list of access levels, [AccessLevel].

```
type Token :: Type → Nat → Type
```

For each mutable store, represented by a Token, there will be a corresponding entry in the parameterised state. Thus, the parameterised state functions like a context in type theory. Previously Token was indexed by one type variable, to indicate the type of the underlying data kept within the store. Now Token will be indexed by an additional type variable, a natural number, i.e. the index of the corresponding access level within the parameterised state. This is analogous to using de Bruijn indices to refer to an entry within a context in type theory.

```
type Store :: [AccessLevel] → [AccessLevel] → Type
data Store p q x where
  Alloc :: t → Store p (Append p X) (Token t (Length p))
  Write :: (X ⩽ Lookup p n) ⇒ Token t n → t
           → Store p p ()
  Read  :: (R ⩽ Lookup p n) ⇒ Token t n
           → Store p p t
```

The first operation is Alloc. This operation takes a value of type t which is used to initialise the elements of the store, and produces a reference to the store. The new type-level state associated with this mutable store is X, so the parameterised state changes from p to Append p X, with X being inserted at the end. This means the index of the type-level state is Length p, so the result of the operation is Token t (Length p).

The second operation is Write. This takes two parameters: a mutable store which has type Token t n; and a value to be put into the store of type t. The expression Lookup p n

[1] In order to use do-notation with this typeclass, the RebindableSyntax language extension is used.

retrieves the type-level state associated with the mutable store, by looking up the value at index n of the parameterised state p. The operation requires $X \leqslant p$, in other words, it must be possible to write to the mutable store.

The third operation is Read. This operation take a reference to a store with type Token t n, and produces the contents of the store with type t. Again, the type-level state for the store is found with Lookup p n and a $\leqslant$ constraint ensures that reading is safe.

```
type Length  :: [m] → Nat
type Append :: [m] → m → [m]
type Lookup :: [m] → Nat → m
```

Note that the store effect above makes use of a number of auxiliary functions, that operate on type-level lists. The signatures of these are given above.

```
type Thread :: [AccessLevel] → [AccessLevel]
  → [AccessLevel] → [AccessLevel] → Type
  → Type → Type
data Thread p p′ q′ q x x′ where
  Fork :: (ShareList p q p′) ⇒ Thread p p′ q′ q () ()
```

In addition to the parameterised version of the Store functor, there is also a parameterised version of the Thread functor. The Share typeclass has already been introduced to constrain how the access level for a single mutable store changes as a result of a Fork operation. The typeclass ShareList extends this to the entire parameterised state. This typeclass generates a Share constraint from each item in the lists p, q and p′.

```
type ShareList :: [AccessLevel] → [AccessLevel]
  → [AccessLevel] → Constraint
class ShareList as bs cs
instance ShareList [ ] [ ] [ ]
instance (Share a b c, ShareList as bs cs)
  ⇒ ShareList (a : as) (b : bs) (c : cs)
```

To conclude this section, we have shown how parameterised algebraic effects can be used to create a new version of the store effect, where data races cannot occur. There are a number of shortcomings of this basic model, which shall be addressed in the following sections.

3.4 Regaining Lost Access

Consider the program below. Once the child thread has been created, the parent thread is unable to access store. Furthermore, the result of the child thread is inaccessible to the parent thread. These problems stem from the fact, that the parent thread has no way of knowing when the child thread has finished.

```
lostAccess = do
  store ← alloc True
  fork do
    write store False
    return ()
  write store False
```

A common concurrent programming pattern is for the fork operation to return a future, representing the result of the child thread, which may not exist yet. There is an accompanying operation wait that can be used to block the thread, until this result is available. We can add this functionality to our framework, as follows.

```
data Thread p p′ q′ q x x′ where
  Fork :: (ShareList p q p′)
    ⇒ Thread p p′ q′ q x (Future x)
data Store p q x where
  ...
  Wait :: Future a → Store p p a
```

This partially improves the situation, allowing the following to be written. Here child is a future, and wait is used to pause the parent thread until the child thread is finished, and obtain its result. This means that using write afterwards will not cause a data race. However, this code would not be allowed in the framework, as wait does not have any type-level behaviour, meaning that the access bounds of store is still N at the point of the second write.

```
lostAccess′ = do
  store ← alloc True
  child ← fork do
    write store False
    return ()
  wait child
  write store False
```

To remedy this, the finish operation will be introduced. The finish operation waits for all threads that are descendants of its inner scope to be completed. This is reflected in the type-level indices. The type-level state after the scoped operation is the same as before.

The rationale for associating this type-level behaviour with a separate operation finish, as opposed to wait is that it makes type-level bookkeeping simpler, while being easier for the programmer to reason about. Further, it means that wait can be used on the same future several times.

```
data Thread p p′ q′ q x x′ where
  ...
  Finish :: Thread p p q p x x
```

This allows lostAccessRegained to be written which demonstrates the use of finish.

```
lostAccessRegained = do
  store ← alloc True
  finish do
```

```
    fork (write store False)
  write store False
```

To conclude this section, the wait and finish operations have been added to allow parent threads to interact with child threads, allowing more concurrent programs to be expressed.

4 Decomposition with Arrays

In this section, we consider how the model can be extended to accommodate arrays. We shall work with Array which resembles Store, but that allocates and manipulates arrays rather than stores. Programs will be written in Prog Array Thread now, rather than Prog State Thread.

```
data Array p q x where
  Alloc :: Int → t
    → Array p (Append p X) (Token t (Length p))
  Write :: (X ⩽ Lookup p n) ⇒ Token t n → Int → t
    → Array p p ()
  Read :: (R ⩽ Lookup p n) ⇒ Token t n → Int
    → Array p p t
  Length :: (R ⩽ Lookup p n) ⇒ Token t n
    → Array p p Int
```

The Alloc operation takes an additional integer argument for the length of the array. The operations Read and Write take an additional argument for the index to read and write from. Finally, there is a Length operation for getting the length of the array.

However this model is overly restrictive, preventing safe programs from being written. One class of programs that cannot be written is recursive array algorithms, such as quicksort [20]. We cannot currently express a concurrent version of such algorithms using our model. This is because only one thread can write to an array concurrently. However simultaneous writes to the same array by different threads are safe, if the writes are to different indices. Languages such as Rust enforce this through the concept of slices, which allow you to construct a reference into a contiguous sequence of elements within a data structure. We shall introduce a similar capability into our model.

```
data ValueType where
  Actual :: ValueType
  Slice  :: Nat → ValueType

type Token :: Type → ValueType → Nat → Type
```

This is accomplished by introducing a Split operation, that allows you to construct two slices, by splitting at a certain index. The type ValueType will be used to differentiate between actual arrays and slices. The type Token will be extended to take a ValueType as a type parameter. If a resource has the value type Slice n it means that it is a slice, that was constructed from the resource at index n, which could itself be a slice.

An example of slices is given below. The array i is split into two slices i_1 and i_2. It is split at index 4, meaning that the indices 0 to 4 of i_1 refer to indices 0 to 4 of i. Whereas the indices 0 to 4 of i_2 refer to indices 5 to 9 of i. After the slice occurs, no further operations can be performed on i, and i_1 and i_2 are treated as entirely separate resources. This means that i_1 can be written to in the child thread, and i_2 can be read from in the parent thread, while the child thread could be concurrently executing.

```
example = do
  i ← alloc 10 True
  (i₁, i₂) ← slice i 4
  fork do
    write i₁ 0 False
  b ← read i₂ 0
  return b
```

We also need to consider how slices interact with access levels. Once a slice to a resource is created, you can no longer directly access that resource. Therefore to create a slice, you need exclusive access to the array, like the write operation. However, unlike the write operation after you create the slice, the access level will become N, reflecting the fact that the resource is lost.

```
data Array p q x where
  ...
  Split :: (X ⩽ Lookup p n, k ~ Length p)
    ⇒ Token t v n → Int
    → Array p (Append (Append (Replace p n N) X) X)
        (Token t (Slice n) k, Token t (Slice n) (S k))
```

Note that this makes use of a new type family Replace xs i x which replaces the element of xs at the ith index with x. Its signature is given below.

```
type Replace :: [m] → Nat → m → [m]
```

The integer argument to a Split operation is the index at which you want to split, and the output of the operation are two slices. We might also imagine an operation to obtain the original resource, from the two slices. This would require exclusive access to both slices, and remove any further access to them.

```
data Array p q x where
  ...
  Join :: (X ⩽ Lookup p n₁, X ⩽ Lookup p n₂,
           Lookup p k ~ N)
    ⇒ Token t (Slice k) n₁ → Token t (Slice k) n₂
    → Array p
      (Replace (Replace (Replace p n₁ N) n₂ N) k X) ()
```

Thus, slices provide a way to naturally implement recursive array algorithms.

4.1 Quicksort

In this subsection, an implementation of the quicksort algorithm will be given. The quicksort algorithm works by choosing a pivot element, and elements in the array are split into two halves. The first half consists of items smaller than or equal to the pivot element. The second half consists of items greater than the pivot element. The quicksort algorithm then recurses on the two individual halves. One advantage of this algorithm is that it can be extended to a multithreaded environment, by running different recursive calls on different threads.

The recursive nature of quicksort leads to a problem, when trying to write it naturally. The problem is that the quicksort function is polymorphic, owing to the presence of the parameterised state. Type inference with polymorphic recursion where type parameters change between invocations is undecidable in general.

Haskell, in particular, makes the assumption that the type parameters are the same, i.e. the type-level state does not change between invocations. However, this is not the case, since two slices are added. Haskell makes this assumption even before the type-checker plugin is invoked. The solution is to explicitly specify a type signature that constrains the input to the function.

```
quicksort :: (X ⩽ Lookup p n)
          ⇒ Token Int r n
          → IProg Array Thread p p ()
quicksort arr = do
  len ← length arr
  if len ⩽ 2 then do
    when (len ≡ 2) do
      v₀ ← read arr 0
      v₁ ← read arr 1
      when (v₀ > v₁) do
        write arr 0 v₁
        write arr 1 v₀
  else finish do
    i ← partition len arr
    (i₁, i₂) ← slice arr (max (i − 1) 0)
    fork do
      quicksort i₁
    quicksort i₂
    return ()
```

Quicksort is given above. First it checks the length of the array. If the length is less than two, it exits. If it is equal to two, then it reads the two values of the array and swaps them if necessary. Otherwise, the function will call another function partition, that splits the function into two halves, returning i as the index of the pivot. Then split is called to divide the array, based on the pivot. Then a child thread is created to sort the left slice, while the parent thread sorts the right slice.

4.2 Convolution

Another example of a concurrent algorithm involving mutable arrays is discrete convolution, which has applications in signal and image processing. For simplicity, the case of convolution with a one-dimensional input array and a three element kernel, constituting the weights, will be considered. In order to handle resources efficiently, the operation will be performed in-place. If the output is o, and the input is i, then the kth element of the output, o_k, is obtained by taking the dot product of $[i_{k-1}, i_k, i_{k+1}]$ with the weights.

```
serialConvolve before after inputs weights = do
  len ← length inputs
  foldM [0 .. (len − 1)] before λi prevEl → do
    let j = i + 1
    currEl ← read inputs i
    nextEl ← if j ≡ len
      then return after
      else read inputs j
    let sum = [prevEl, currEl, nextEl] · weights
    write inputs i sum
    return currEl
  return ()
```

The function serialConvolve performs convolution serially, with before and after being the boundary elements to the left and right respectively. It performs a fold on an array of indices, with the value of the previous element being carried over in each iteration. It is necessary to carry over the previous element, as it will have been overwritten by the output element, as the algorithm is performed in-place.

```
parallelConvolve :: (X ⩽ Lookup p n)
                 ⇒ Int → Int → Token Int r n → [Int]
                 → IProg Array Thread p p ()
parallelConvolve before after inputs weights = do
  len ← length inputs
  if len < 10 then do
    serialConvolve before after inputs weights
  else finish do
    let middle = len/2
    middle₁ ← read inputs middle
    middle₂ ← read inputs (middle + 1)
    (i₁, i₂) ← slice inputs middle
    fork do
      parallelConvolve before middle₂ i₁ weights
    parallelConvolve middle₁ after i₂ weights
    return ()
```

The function parallelConvolve performs convolution using multiple threads, in a similar way to quicksort. When

the array is smaller than a certain size, it will delegate the operation to serialConvolve. Otherwise, it will split the array into two slices and recurse. It is necessary to read the two middle elements before splitting the array, to obtain the boundary elements for the two slices.

4.3 Discrete Fourier Transform

The final example to be presented is an implementation of the fast Fourier transform (FFT) algorithm, that computes the discrete Fourier transform (DFT) of a sequence. It is typically used to convert a signal from the time or space domain to the frequency domain. The FFT algorithm is an efficient algorithm to perform this transformation with recursion.

```
fft :: (R ⩽ Lookup p n1, X ⩽ Lookup p n2, n1 ≁ n2)
  ⇒ Token (Complex Double) r1 n1
  → Token (Complex Double) r2 n2
  → (Int, Int, Int)
  → IProg Array Thread p p ()
fft input output (start, gap, len) = do
  if len ≡ 1 then do
    v ← read input start
    write output 0 v
  else do
    let offset = len/2
    finish do
      (output1, output2) ← slice output (offset − 1)
      fork do
        fft input output1 (start, gap ∗ 2, offset)
      fft input output2 (start + gap, gap ∗ 2, offset)
    forM [0 . . (offset − 1)] λj1 → do
      let j2 = j1 + offset
      x1 ← read output j1
      x2 ← read output j2
      let factor1 = exp (−2i ∗ π ∗ j1) / len
      let factor2 = exp (−2i ∗ π ∗ j2) / len
      write output j1 (x1 + factor1 ∗ x2)
      write output j2 (x1 + factor2 ∗ x2)
    return ()
```

Unlike the previous examples, this algorithm is not performed in-place, and this implementation uses two arrays: one containing the input array; and another containing the output array. The input is assumed to be a power of 2, which allows the algorithm to be simpler.

Slices formed from the output array are passed to recursive calls. However, this is not the case for the input array, which is passed in its entirety to recursive calls. Instead, an auxiliary argument is passed to represent the subarray within the overall input array to perform DFT on. This is because the recursive calls operate on discontiguous subarrays within the input array.

In the base case, the input is transferred unchanged to the output array. In the recursive case, the DFT of the even indices and the odd indices are handled by separate recursive calls. The output array is split into two slices, but the input array remains the same, and instead the auxiliary argument is changed to reflect the subarrays passed to these recursive calls. After the subcomputations are complete, the results are combined, by the for-loop.

Note that this is data race free, as the input array is only being read so can be shared between recursive calls. However, the output array is being written to, and must be split between recursive calls. This is reflected in the type signature, where read access to the input array is needed, but exclusive access to the output array is needed.

A subtle aspect of the type signature is the disequality constraint $n_1 \nsim n_2$, which states that the input array and the output array are different. If the input array and the output array were the same, then this could result in a data race. Naturally, this means the absence of the constraint would lead to the program failing to type-check.

To conclude this section, we've seen a conservative extension of the framework that makes it straightforward to safely encode concurrent divide-and-conquer algorithms.

5 Type-Checker Plugin

One issue with the library, that has hitherto been avoided, is the issue of ambiguous type variables. This is where ambiguous type variables prevent the Haskell type-checker plugin to make progress solving constraints, even if enough information is already present to do so. One example encountered earlier was in the ambiguousState example where there were two possible ways that the Share typeclass could be satisfied. One approach to this problem would be to extend GHC with custom-tailored rules to handle these specific cases. However, this heavyweight approach would demand that users of the library depend on a customised implementation of GHC, which is of course an unrealistic expectation.

An alternative approach which is considerably more practical is the use of a compiler plugin. GHC offers different interfaces to modify its internal behaviour [39]. A recent addition is the type-checker plugin interface, which allows plugins to modify the constraint solver.

```
data TcPlugin = ∀s.TcPlugin
  {tcPluginInit  :: TcPluginM s
  , tcPluginSolve :: s → TcPluginSolver
  , tcPluginStop  :: s → TcPluginM ()
  }
```

The type TcPlugin that defines a type-checker plugin is given above. Most of the work occurs in the tcPluginSolve function that is called by GHC. GHC provides the function with a list of given constraints, which are constraints that are known to hold and a list of wanted constraints, that GHC

has yet to prove. The plugin improves type inference significantly, meaning that the end-user has to provide minimal type annotations, which considerably reduces the burden of the library.

5.1 Architecture

The first step of the type-checker plugin is taking the constraints given by GHC, and converting them into its own internal representation.

```
data OpType
  = OpApp OpFun [OpType] | OpVar Var | OpLift Type
  | OpMultiAppend OpType [OpType] | ...
data OpFun
  = OpReplace | OpLookup | OpLength | OpAppend | ...
data OpClass
  = OpEquality | OpShare | OpShareList | OpLeq | ...
data OpConstraint
  = OpConstraint OpClass [OpType]
data OpConstraintSet
  = OpConstraintSet [OpConstraint] Bool
```

The key data types are given above:

- OpFun is used to encode type constructors.
- OpType is used to encode types.
- OpClass is used to encode the different types of constraints, either different typeclasses or an equality constraint.
- OpConstraint is used to encode a constraint, i.e. an OpClass value and its arguments in the form of OpType values.
- OpConstraintSet is used to represent a series of constraints. The Bool represents a change flag, to record if a modification has been made to the constraint set.

Each OpConstraintSet starts out representing a single constraint, but might be broken down into a series of constraints during processing. Grouping these constraints together and storing a change flag is useful for bookkeeping purposes.

Using this bespoke representation with dedicated constructors for type-level utilities used in the library, makes it far simpler to write code to perform transformations on and extract information from the constraints.

After this representation is constructed, constraint solving can begin. Constraint solving consists of two phases. In the analysis phase, the constraints are inspected to extract useful information. In the simplification phase, the constraints are refined, with the help of the information retrieved in the analysis phase.

As might be expected, several iterations of these two phases might be applied before the constraints are fully solved. This is because refinements will inevitably beget further refinements.

5.2 Constraint Solving

To understand some of the most important simplifications, that the plugin performs, it is illustrative to look at some examples. The following function is an embarrassingly simple example of where the plugin will ensure unnecessary constraints are avoided.

```
allocThenPut = do
  s ← alloc False
  write s True
```

The function allocThenPut, as the name suggests, allocates a mutable store s, and writes a value s to it. The reason is that the parameterised state upon entry to the function is unknown, and so treated by the Haskell type-checker as an ambiguous type variable a. When the mutable store is created, this type-level state becomes Append a X. Then the write operation generates the following constraint:

```
Leq X (Lookup (Append a X) (Length a))
```

It might be clear to us that the Lookup constraint can be simplified to X, since the element at index Length a of Append a X is going to be X. However, the Haskell type-checker cannot deduce this, unless Append a X and Length a are concrete values, which is of course not possible if a is ambiguous. The plugin, however, is equipped with a rule that can handle this. The plugin can also handle more complex cases such as the expression:

```
Lookup (Append (Append a R) X) (S (Length a))
```

Here, the Lookup is performed with multiple items appended to the base list a, and the index to look up is given as an offset relative to the length of the base list. This simplifies to X. This type of simplification is an example of one that does not require any information from the analysis phase.

Another example of how ambiguous type variables might arise is given with fork.

```
allocThenPutTwo = do
  s₁ ← alloc False
  s₂ ← alloc False
  fork do
    write s₁ True
  write s₂ True
```

Here two mutable stores are allocated: s_1 and s_2. The child thread created from fork writes to s_1, while the parent thread writes to s_2. The fork operation leads to the following constraint being generated:

```
ShareList (Append (Append a X) X) b c
```

The variable b refers to the parameterised state after the fork operation in the parent thread, and the variable c refers to the parameterised state after the fork operation in the child thread.

Note that a is the parameterised state upon entry to the function, and that since it is unknown the constraint cannot be solved. The plugin, however, is able to identify the instances where b and c are referred to in other constraints, in the analysis phase. With this information, it replaces the ShareList constraint with two Share constraints:

- Share (Lookup (Append (Append a X) X) (Length a)) (Lookup b (Length a)) (Lookup c (Length a))
- Share (Lookup (Append (Append a X) X) (S (Length a))) (Lookup b (S (Length a))) (Lookup c (S (Length a)))

The first constraint is on the use of s_1, and the second constraint is on the use of s_2. By applying the rules on Lookup discussed earlier, these can be further simplified to:

- Share X (Lookup b (Length a)) (Lookup c (Length a))
- Share X (Lookup b (S (Length a))) (Lookup c (S (Length a)))

Note that the use of the write operation generates the following two constraints:

- X $\leqslant$ Lookup b (S (Length a))
- X $\leqslant$ Lookup c (Length a)

The first constraint is from the write operation in the parent thread, and the second is from the write operation in the child thread. Information about the Leq constraints and the Share constraints, obtained in the analysis phase, can be combined to jointly solve both groups of constraints with the following equations:

- Lookup b (Length a) ~ N
- Lookup b (S (Length a)) ~ X
- Lookup c (Length a) ~ X
- Lookup c (S (Length a)) ~ N

The Share constraints are solved as Share X N X and Share X X N respectively. The $\leqslant$ constraints are solved as X $\leqslant$ X and X $\leqslant$ X. This approach scales up to more complex situations, where the mutable stores are not allocated within the function, but passed in as arguments. Further, there are similar rules for solving situations where there are read operations.

To sum up, this section has presented the type-checker plugin, a core component of the library, demonstrating some of the ways that the plugin works to improve type inference.

6 Related Work

The work in this paper draws from and has similarities with a broad range of interconnected ideas.

Behavioural types. This field of study arose from process calculi, formal models of concurrency based on communication channels, which are an alternative way to co-ordinate between processes [21]. Systems based on behavioural types have been used to avoid data races and ensure safety and liveness in concurrent programs.

A particular class of behavioural types is session types. In this typing discipline, the types of communication channels encode the protocol, describing the interaction between two processes. There are also multiparty extensions, which account for communication between many processes. There have been several encodings of session types in Haskell [28, 35, 42].

Linear types. The notion of constraints of resources has close links to linearity [18, 46]. In particular, the work here can be emulated through a combination of linear types and indexed types. Rather than storing type-level state within a parameterised algebra, type-level state could be stored in an indexed type.

There is currently on-going work to integrate linear types natively into Haskell [7]. This has required significant changes to GHC, since linearity has to be 'retrofitted' onto Haskell. In contrast, the work presented here takes advantage of the type-level programming features already present in GHC.

An embedding of linearity was presented in Paykin and Zdancewic [36]. Their system was also written in Haskell and used type-level constructs in Haskell to encode constraints. In particular, this system was also used to ensure safe access to mutable arrays, with a similar split operation.

This implementation was done using higher-order abstract syntax [38], a technique for embedding a language in a host language while using the variable bindings of the host language. A disadvantage of this approach is the presence of combinators, whenever there is function abstraction and application, which makes the code difficult to read.

Ownership types. Ownership types are a closely related idea to linearity [13]. While linear types are typically found in functional languages, ownership types are typically found in imperative languages, where an object has a guarantee that it has the only reference to another object.

In Boyapati et al. [9], ownership types are used to implement a type system that is not susceptible to data races. In this system, accesses to an object are either carried out with synchronisation or there is a guarantee that the access is safe.

The Rust programming language has a type system based on ownership and borrowing that allows for safe mutation with mutable state [29]. The type system will not allow potentially unsafe accesses to memory, unless there is synchronisation.

Effect systems. Another branch of research has focussed on using type and effect systems for ensuring the correctness

of concurrent programs [31]. In [8], this is accomplished in Java, with a region-based system, which is used to partition the heap. This paper also explores divide-and-conquer algorithms, by partitioning an array into two disjoint subarrays. In [17], a type and effect system is also added to Java, but this approach focusses on the interaction between locks and the type system.

Coeffects. A generalisation of linearity is the coeffect [37]. While an effect corresponds to how a function changes its execution context, a coeffect is a restriction on a function's execution context. Coeffects correspond to graded comonads, and this has been incorporated into the type system of the Granule programming language [32]. In our system, the type-level state corresponding to each instance of mutable state can be seen as a coeffect.

Typestate. Typestate is an old idea found in object-oriented languages of associating state with variables of a given type [3]. Method invocations will lead to changes in type-level state. This could be modelled as a finite state machine. This is very similar to the idea of type-level state discussed in this paper, where operations lead to a change in type-level state.

Capabilities. A capability is a token that grants access to an object [26, 27]. In Miller [30], a programming language is developed that combines the object-oriented paradigm with capabilities to allow objects to cooperate concurrently while avoiding 'destructive interference', that might prevent safety or liveliness. This system is largely dynamic, whereas the system developed here is static.

In Boyland [11], a type system is developed for a variant of capabilities, that allow different views of data, based on fractions. In particular, a whole fraction is needed for write access, however a capability can be split into smaller fractions, which only give read access. Fractions can be combined together again to retrieve write access. This bears a resemblance to the model developed here, in that different threads can have different levels of access to a resource.

In Castegren and Wrigstad [12], a language is introduced that encodes capabilities using traits, reusable structures that can be combined to construct classes. An interesting feature is that there can be concurrent access to disjoint parts of the same object from different threads. An extension to this work has been presented which extends this idea of disjoint access to arrays. This is similar to the split operation presented here [2].

Actor model. Like process calculi, the actor model is based on communication, with the primitives of actors and behaviours [1, 16]. Actors send messages directly to each other, and handle messages with behaviours. In Clebsch et al. [14], a type system is presented that combines the actor model with mutable state, while avoiding data races with capabilities. This type system has been implemented in the Pony programming language.

Notions of computation. In this work, parameterised computations have been used to ensure constraints on resources. An alternative generalisation of a computation is a graded computation, where the computation is indexed by only one index, with a monoidal structure [33]. In Ivašković and Mycroft [22], it is shown how graded monads can be used to prevent deadlocks, in a concurrent system. Unlike the system presented here, this system does not attempt to prevent data races without synchronisation, and relies on a fixed number of global locks. The connection between parameterisation and grading is explored in Orchard et al. [34].

7 Future Work

One direction for future research would be to explore whether the decomposition that is possible with arrays, can be applied to more complex data structures such as binary trees. This would make it possible to implement a recursive algorithm that creates new threads as it traverses a tree. Another potential development would be to model a system that integrates message-passing with mutable state, similar to the system found in Pony. A further direction for future work would be to apply the same techniques described here to other kinds of resources, where there are constraints are how operations can be performed. Examples of such resources might be network sockets and graphics contexts.

8 Conclusion

This paper adds a novel approach to a growing body of work on safe concurrent programming. Other strands of research focus on either avoiding mutable state altogether, or creating entirely new languages tailored to allow safe mutation. However, this paper demonstrates that advanced language features enable safe mutation to be integrated with a mainstream functional programming language in a principled way. In this way, the paper also advances research into the applications of advanced language features such as type-checker plugins and type-level computation.

Acknowledgements

This work has been supported by EPSRC grant number EP/S028129/1 on "SCOPE: Scoped Contextual Operations and Effects".

References

[1] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[2] Beatrice Åkerblom, Elias Castegren, and Tobias Wrigstad. 2019. Reference Capabilities for Safe Parallel Array Programming. *CoRR* abs/1905.13716 (2019). https://doi.org/10.22152/programming-journal.org/2020/4/1 arXiv:1905.13716

[3] Jonathan Aldrich, Joshua Sunshine, Darpan Saini, and Zachary Sparks. 2009. Typestate-Oriented Programming. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications (OOPSLA '09).* Association for

Computing Machinery, New York, NY, USA, 1015–1022. https://doi.org/10.1145/1639950.1640073

[4] Robert Atkey. 2009. Algebras for Parameterised Monads. In *Algebra and Coalgebra in Computer Science*, Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–17.

[5] Robert Atkey. 2009. Parameterised Notions of Computation. *J. Funct. Program.* 19, 3–4 (July 2009), 335–376. https://doi.org/10.1017/S095679680900728X

[6] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. 2000. Guava: A Dialect of Java without Data Races. *SIGPLAN Not.* 35, 10 (Oct. 2000), 382–400. https://doi.org/10.1145/354222.353197

[7] Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-Order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

[8] Robert L. Bocchino, Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. 2009. A Type and Effect System for Deterministic Parallel Java. *SIGPLAN Not.* 44, 10 (Oct. 2009), 97–116. https://doi.org/10.1145/1639949.1640097

[9] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. 2002. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*. Association for Computing Machinery, New York, NY, USA, 211–230. https://doi.org/10.1145/582419.582440

[10] John Boyland. 2001. Alias Burying: Unique Variables without Destructive Reads. *Softw. Pract. Exper.* 31, 6 (May 2001), 533–553. https://doi.org/10.1002/spe.370

[11] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72.

[12] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs))*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Vol. 56. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:26. https://doi.org/10.4230/LIPIcs.ECOOP.2016.5

[13] Dave Clarke, Johan Östlund, Ilya Sergey, and Tobias Wrigstad. 2013. *Ownership Types: A Survey*. Springer Berlin Heidelberg, Berlin, Heidelberg, 15–58. https://doi.org/10.1007/978-3-642-36946-9_3

[14] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE! 2015)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2824815.2824816

[15] 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 (POPL '99)*. Association for Computing Machinery, New York, NY, USA, 262–275. https://doi.org/10.1145/292540.292564

[16] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016)*. Association for Computing Machinery, New York, NY, USA, 31–40. https://doi.org/10.1145/3001886.3001890

[17] Cormac Flanagan and Shaz Qadeer. 2003. A Type and Effect System for Atomicity. *SIGPLAN Not.* 38, 5 (May 2003), 338–349. https://doi.org/10.1145/780822.781169

[18] Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50, 1 (Jan. 1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

[19] 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 (PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 48–60. https://doi.org/10.1145/1065944.1065952

[20] C. A. R. Hoare. 1961. Algorithm 64: Quicksort. *Commun. ACM* 4, 7 (July 1961), 321. https://doi.org/10.1145/366622.366644

[21] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (April 2016), 36 pages. https://doi.org/10.1145/2873052

[22] 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 (Haskell 2020)*. Association for Computing Machinery, New York, NY, USA, 17–30. https://doi.org/10.1145/3406088.3409024

[23] Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. *SIGPLAN Not.* 50, 12 (Aug. 2015), 94–105. https://doi.org/10.1145/2887747.2804319

[24] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

[25] Oleg Kiselyov, Amr Sabry, and Cameron Swords. 2013. Extensible Effects: An Alternative to Monad Transformers. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell (Haskell '13)*. Association for Computing Machinery, New York, NY, USA, 59–70. https://doi.org/10.1145/2503778.2503791

[26] Hugh C. Lauer and Roger M. Needham. 1979. On the Duality of Operating System Structures. *SIGOPS Oper. Syst. Rev.* 13, 2 (April 1979), 3–19. https://doi.org/10.1145/850657.850658

[27] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann, USA.

[28] Sam Lindley and J. Garrett Morris. 2016. Embedding Session Types in Haskell. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016)*. Association for Computing Machinery, New York, NY, USA, 133–145. https://doi.org/10.1145/2976002.2976018

[29] Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *Ada Lett.* 34, 3 (Oct. 2014), 103–104. https://doi.org/10.1145/2692956.2663188

[30] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph.D. Dissertation. USA. Advisor(s) Shapiro, Jonathan S. AAI3245526.

[31] Flemming Nielson and Hanne Riis Nielson. 1999. *Type and Effect Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 114–136. https://doi.org/10.1007/3-540-48092-7_6

[32] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative Program Reasoning with Graded Modal Types. *Proc. ACM Program. Lang.* 3, ICFP, Article 110 (July 2019), 30 pages. https://doi.org/10.1145/3341714

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

[34] Dominic Orchard, Philip Wadler, and Harley Eades. 2020. Unifying graded and parameterised monads. *Electronic Proceedings in Theoretical Computer Science* 317 (05 2020), 18–38. https://doi.org/10.4204/EPTCS.317.2

[35] Dominic Orchard and Nobuko Yoshida. 2016. Effects as Sessions, Sessions as Effects. *SIGPLAN Not.* 51, 1 (Jan. 2016), 568–581. https://doi.org/10.1145/2914770.2837634

[36] Jennifer Paykin and Steve Zdancewic. 2017. The Linearity Monad. *SIGPLAN Not.* 52, 10 (Sept. 2017), 117–132. https://doi.org/10.1145/3156695.3122965

[37] Tomas Petricek, Dominic Orchard, and Alan Mycroft. 2014. Coeffects: A Calculus of Context-Dependent Computation. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. Association for Computing Machinery, New York, NY, USA, 123–135. https://doi.org/10.1145/2628136.2628160

[38] Frank Pfenning and Conal Elliott. 1988. Higher-Order Abstract Syntax. *Sigplan Notices - SIGPLAN* 23, 199–208. https://doi.org/10.1145/960116.54010

[39] Matthew Pickering, Nicolas Wu, and Boldizsár Németh. 2019. Working with Source Plugins. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell (Haskell 2019)*. Association for Computing Machinery, New York, NY, USA, 85–97. https://doi.org/10.1145/3331545.3342599

[40] Maciej Piróg, Tom Schrijvers, Nicolas Wu, and Mauro Jaskelioff. 2018. Syntax and Semantics for Operations with Scopes. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. Association for Computing Machinery, New York, NY, USA, 809–818. https://doi.org/10.1145/3209108.3209166

[41] Gordon Plotkin and John Power. 2004. Computational Effects and Operations: An Overview. *Electron. Notes Theor. Comput. Sci.* 73 (Oct. 2004), 149–163. https://doi.org/10.1016/j.entcs.2004.08.008

[42] Riccardo Pucella and Jesse A. Tov. 2008. Haskell Session Types with (Almost) No Class. *SIGPLAN Not.* 44, 2 (Sept. 2008), 25–36. https://doi.org/10.1145/1543134.1411290

[43] C. Ruggieri and T. P. Murtagh. 1988. Lifetime Analysis of Dynamically Allocated Objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '88)*. Association for Computing Machinery, New York, NY, USA, 285–293. https://doi.org/10.1145/73560.73585

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

[45] Carmen Torres Lopez, Stefan Marr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2018. *A Study of Concurrency Bugs and Advanced Development Support for Actor-based Programs*. Springer International Publishing, Cham, 155–185. https://doi.org/10.1007/978-3-030-00302-9_6

[46] Philip Wadler. 1990. Linear Types Can Change the World!. In *Programming Concepts and Methods*. North.

[47] Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect Handlers in Scope. *SIGPLAN Not.* 49, 12 (Sept. 2014), 1–12. https://doi.org/10.1145/2775050.2633358