AN ABSTRACT OF THE THESIS OF

Michael McGirr for the degree of Master of Science in Computer Science presented on TODO submit date.

Title: The Ownership Monad

Abstract approved: _____

Eric Walkingshaw

TODO abstract statement.

The Ownership Monad

by

Michael McGirr

A THESIS

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Master of Science

Presented TODO submit date
Commencement June TODO commencement year

Master of Science thesis of Michael McGirr presented on TODO submit date.

APPROVED:

Major Professor, representing Computer Science

Director of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my thesis will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my thesis to any reader upon request.

Michael McGirr, Author

ACKNOWLEDGEMENTS

TODO

TABLE OF CONTENTS

LIST OF ALGORITHMS

Algorithm Page

Chapter 1: Introduction

This project report presents a library[1] for Haskell that implements an *ownership system* for resource aware programming. This ownership system introduces a set of rules that govern how, when, and by whom a resource within what is called an *Owned Reference* (an `ORef`) can be used.

When an owned reference is created it is bound to a resource. The resource is placed within an entry which is then stored in the state of the ownership monad. At the same time this occurs, the entry becomes the referent to a uniquely identifiable variable ID. This variable becomes the way to refer to and access the contents of the resource it is bound to inside the ownership monad. Each referent can only have, at most, one variable binding at a time.

While the restrictions this implementation imposes may seem to increase the complexity of writing programs, the resulting guarantees offer improvements in specific areas. Furthermore applying this system provides an example for how monadic based approaches can offer abilities to reason about resource usage.

Adding a way to keep track of resources in a pure language like Haskell may at the onset seem unnecessary since in a pure language, by definition, the data making up the resources bound to variables is immutable. Because of this there is no inherent state. Haskell's purity therefore allows for referential transparency where values and variables in closed expressions can be thought of as being interchangeable.

Referential transparency is a desirable property which allows for a greater ability to reason about the behavior and correctness of a program. Even in a pure language like Haskell, this property breaks down in the context of concurrency. Under these circumstances, Haskell's usual approach of segregating side-effects into monadic computations does not resolve every issue that can exist with shared-state concurrency.

[1] The library presented in this project report is available at: https://github.com/lambda-land/OwnershipMonad

A simple example of this is the race condition that can be created when a mutable reference is shared across a channel between two threads:

```
1  import Data.IORef
2  import Control.Concurrent
3  import Data.Char (toLower, toUpper)
4
5  introChan :: IO ()
6  introChan = do
7    ref <- newIORef "resource"
8    ch <- newChan
9    writeChan ch ref
10
11   _ <- forkIO $ do
12     ref' <- readChan ch
13     modifyIORef ref' (map toLower)
14
15   val <- readIORef ref
16   putStrLn val
17
18   modifyIORef ref (map toUpper)
19   newVal <- readIORef ref
20   putStrLn newVal
```

Here the `forkIO` function takes an expression of type `IO ()` and executes it in a new thread. This new thread will run concurrently with the original parent thread.

This creates a clear race condition between the two threads and the mutable contents of the `IORef` shared between them. Both threads have access to the same `IORef` and it could be unclear which will access the resource first. Under these conditions the issue is not only that we have shared mutable state but that we have shared access to that state.

This kind of race condition would be difficult to detect. The easy culprit to blame in the previous example would be the mutable `IORef` shared between the two threads. Immutable data structures simplify concurrency and are preferable to mutable structures.

However categories of problems still exist where a mutable structure is required.

The `OChan` and `ORef` operations provided by this report's library allow a user to detect situations, similar to the one created by the previous example, which would result in a race condition. A comparable program using the abstractions provided by the ownership system would be able to detect the race condition before it occurred:

```
1  import Data.ORef
2  import Control.Concurrent.OChan
3
4  introOChan :: IO (Either String ())
5  introOChan = startOwn $ do
6    ref <- newORef "resource"
7    let ch = newOChan
8    writeOChan ch ref
9
10   let down :: String -> Own String
11       down x = return (map toLower x)
12
13       up :: String -> Own String
14       up x = return (map toUpper x)
15
16   _ <- liftIO $ forkIO $ do
17     _childResult <- startOwn $ do
18       ref' <- readOChan ch
19       borrowORef' ref' down
20       writeOChan ch ref'
21     return ()
22
23   borrowORef' ref up
24   return ()
```

Here a resource is created and placed within an `ORef`. The parent thread proceeds to give up ownership of the `ORef` when it writes it to the owned channel on line 5. The child thread can read the `ORef` from the `OChan` and claim ownership over its contents.

The child thread can then mutate the resource in a borrow operation before writing it back to the channel.

Each of the ownership operations used in this example will be defined and discussed at greater length later on. This example demonstrates a sequence of actions that result in an ownership violation. This violation is occurs when parent thread tries to use a reference that it has already given up ownership over by writing the reference to a channel.

When the overall example is evaluated it will result in a `Left` value which will indicate the kind of ownership violation which took place. A simple remedy for the violation can then be found. For this example the parent thread could take back ownership of the reference by reading it from the owned channel.

1.1 Contributions

This project draws from the use of affine types in Rust for ownership typing to define a similar method for tracking resources with Haskell. This is combined with the approach of defining embedded languages in monadic programming to allow for dynamic ownership checking without language level support for affine types in Haskell.

Concurrent Haskell programs can still fall prey to some of the same fundamental problems that other impure languages can, namely deadlock and data races. This project report will demonstrate a motivation for adding the kind of resource tracking that ownership typing provides by looking at what some of these problems look like and how resource tracking acts to mitigate these problems. This will use the basic concurrency tools available in Haskell such as shared state with IORef's and message passing with channels. It will then demonstrate and explain the benefits of tracking resource usage with a set of rules similar to affine types.

Chapter 2: Background

2.1 Linear and Affine Types

A linear type system enforces the rule that resources are used exactly once. Linear types are often suggested as a possible solution to limit the issues that result from mutable state [11] and concurrency [2].

Language level support for linear types has been proposed for the Glasgow Haskell Compiler. [1] Ways to add linear types to Haskell without language level support have also been demonstrated using embedded domain specific languages within a monadic context. [10]

Other less restrictive forms of logic have been used in type systems for similar resource tracking. Affine type systems weaken the restrictions imposed by linear type systems. Instead of requiring every variable to be used exactly once - as is the case with linear types - every variable must be used at most once. The language level ownership typing in Rust has been directly inspired by affine type systems.

2.2 Uniqueness Types

Idris, a dependently typed language, treats uniqueness types as a subkind of regular types. Uniqueness types in Idris are also inspired by ownership types and borrowed pointers in the Rust programming language, as well as Uniqueness Types in the Clean programming language.

In Idris a value that has a type that is made up of a `UniqueType` will only be able to have at most one reference to it at run-time. In order to make this guarantee, any value of a unique type is kept separate from normally typed values - which have the type `Type`.

Polymorphic functions which take either unique types or regular types can exist. The type `Type*` is used to denote when a type which can either be a unique type or a regular type (but not both) may be used in a polymorphic function.

```
head : {a : Type*} -> List a -> a
```

A polymorphic function that takes either unique types or regular types will also have to obey the rules for how values that are uniquely typed can be used on the right hand side of a function.

Idris allows for unique typed variable to be used more than once on the right side of a function as long as they are being read and the value is not updated. This means that a uniquely typed variable can have its value inspected in a function and this will not count towards that variable being used. The variable will not be able to be mutated or used in any way which would mutate the underlying resource.

To do this, Idris creates a new kind of type from the unique type. This new type is called a `BorrowedType` and at the type level exists alongside `UniqueType` and `Type`.

Idris treats a borrowed type as a different kind of type from unique types. Using a variable of the borrowed type adds in additional requirements and checks at the type level to ensure that the borrowed type will not mutate while its resource is being read. While a borrow is occurring the internals will exclude other functions or operations from mutating the resource.

In Idris to convert a unique type into a borrow type the `Borrowed` dependent data type is used:

```
data Borrowed : UniqueType -> BorrowedType where
     Read : {a : UniqueType} -> a -> Borrowed a

implicit
lend : {a : UniqueType} -> a -> Borrowed a
lend x = Read x
```

The `lend` function can then be invoked to create a `BorrowedType`. Doing so will indicate that while the underlying `UniqueType` variable is being used new references to it should not be tracked as additional references to a unique value. This allows the borrowed variable to be used as many times as it needs to be instead of at most once.

2.3 Ownership Types in Rust

A central feature of the Rust programming language is its use of an ownership-based type system. The ownership model used in Rust and the safety claims of using such a system have been formally proven and machine-checked in Coq [8].

The Rust language has inherent state and additionally all variables must obey the ownership rules. These rules are checked at compile time and are enforced over all variables in a program. Doing so allows Rust to forgo a runtime system or a garbage collector.

In Rust, memory allocation is handled through the ownership system. The ownership system is a language level abstraction which makes use of affine types for resource management. Additionally Rust uses *lifetime* analysis of resource use for fast deallocation. Lifetimes in Rust are beyond the scope of this project report and in some ways are more similar to a linearly typed language than an affine one.

When a resource is created and bound to a variable, such as `v1` in the example below, that variable is the *owner* of the resource.

```
let v1 = vec!["Vector", "of", "Strings"];
```

Ownership may be transferred to a different variable with a new assignment. The variable `v2` in the next example takes ownership over the vector which belonged to `v1`.

```
let v2 = v1;
```

Any attempts to access the values in the vector by referring to them through the old owner `v1` will result in a compile time error. When the variable which owns the resource goes out of scope at the end of a code block the resource will be freed.

There may be situations in which one variable needs to temporarily use the resource owned by another variable without taking ownership over it. To do this a borrow of that variable is created:

```
let v1 = vec!["Vector", "of", "Strings"];
let v2 = &v1;
```

Ampersands are used to create references to the resources owned by other variables. In the last example the variable `v2` was instantiated as a borrow of the resource owned by `v1`.

Borrows in Rust come in two flavors. We can either lend a resource to many borrowers as long as the borrowers never mutate the underlying resource or we can lend it to a single borrower that will be able to mutate the resource.[4] It should be clear why giving multiple variables mutable access to the same resource could create data races which is why mutable borrows to multiple variables (or functions) are not allowed.

A function in Rust depending on the type signature will either copy the value it is passed, take ownership of the value, or it will borrow the value - in which case ownership of the value is automatically returned when the function has finished execution.[7] A function in Rust that takes a borrowed value as an argument is syntactic sugar over that function first taking ownership of the value and then returning ownership over the value by placing it within the expression that is returned. Instead of having to do these steps explicitly a value can be passed to a function as a borrowed value. When a value is borrowed, the function will take a reference to that the value from the original owner and eventually the ownership of the resource will be handed back when the function returns. The Rust compiler will track the borrows (with the borrow checker) statically at compile time.

Chapter 3: The Ownership Monad

The term *Ownership System* is used to describe the system for how resources are tracked and how they can be used once they are created. This system operates within the context of the ownership monad. The ownership system this library implements is inspired by ownership typing in Rust [7] as well as uniqueness types from Idris [3].

The ownership system described by this paper approximates some of the features from ownership typing in the Rust language. Differences between the two result from the different language paradigms and the different use cases.

Uniqueness types in Idris, ownership typing in Rust, and the ownership system in this report make use of the idea that by tracking resources and applying rules to how resources are used, certain properties can be enforced. One of the primary motivations for using the ownership monad is as a safer way to introduce mutability, when it is needed, into a Haskell program.

3.1 The Ownership System

From the perspective of the library user: owned references are abstract data types defined by the operations which can act on references in the ownership monad. The semantics of how owned references exist and operate correspond to the outcomes of using these operations.

The functions to perform operations are made available in the public facing API module of the library for this implementation.

```
newORef    :: Typeable a => a -> Own (ORef a)
dropORef   :: Typeable a => ORef a -> Own ()
copyORef   ::               ORef a -> Own (ORef a)
moveORef   :: Typeable a => ORef a -> Own (ORef a)
writeORef  :: Typeable a => ORef a -> a -> Own ()
borrowORef :: Typeable a => ORef a -> (a -> Own b) -> Own b
readORef   :: Typeable a => ORef a -> Own a
```

These functions provide the operations to create owned references, drop a reference from the scope, copy them, move the resource from one reference to another, and write a value to a reference. Additionally there are operations for borrowing and reading the resource inside a reference. Borrowing allows a function to temporarily use the resource without taking ownership over it. Reading a reference which retrieves a copy of the value inside a reference.

3.1.1 Owned References

Owned references represent variables which provide both the symbolic entity which owns a resources as well as the mechanism to access the underlying resource. Resources are bound to a variable when they are created inside the ownership monad. In the library implementation these variables are called ORef's or *Owned References*.

The type of an `ORef` is a thin wrapper around a way to tag and identify resources stored in entries.

```
newtype ORef a = ORef {getID :: ID}
```

An `ORef` is a parameterized abstract data type. Because there is only one constructor and one field associated with this datatype it is possible to use newtype to eliminate some of the runtime overhead.

A phantom type is used to add a type variable to each `ORef`. This ensures the type safety of the code by requiring that a type be embedded with each `ORef`. This prevents references of different types from being mixed by invoking a type error at compile time. The polymorphic `a` allows any type to be stored in an `ORef`. This `ORef` datatype primarily serves to provide a handle on each resource for the ownership monad to use as it enforces ownership typing rules and tracks each resource.

When the operations inside an ownership monad are complete the references will no longer exist and the resources associated with each one will be marked as free. The information inside the resource can only be accessed by the provided operations for references within the ownership monad. The operations which act on owned references will verify whether the ownership rules are being followed and detect violations.

3.1.2 Reference Creation

References need to be introduced into the ownership monad in order to be used. To do this a library user would create a new reference and bind it with a resource. Both the instantiation of the new reference as well as the resource binding occur in the same step. A simple example of this is the creation of a reference `x` which is bound to the resource $[1, 2, 3]$. In the following code the owned reference `x` is created. The reference can now be used by other operations within the monad in which it resides.

```
x <- newORef [1,2,3]
```

The `newORef` function is part of the user facing API of the library and. As the example has shown, this function creates a new owned reference within the ownership monad and places the value provided to the function inside the reference. This value is internally stored within an `IORef` in the `Entry` datatype.

```
newORef :: Typeable a => a -> Own (ORef a)
newORef a = do
    (new, store) <- get
    thrId <- liftIO $ myThreadId
    v <- liftIO $ newIORef a
    let entry = (Entry Writable thrId (Just v))
    put (new + 1, insert new entry store)
    return (ORef new)
```

The current state of the monad is needed in order to produce the next state which will include the new owned reference. The new owned reference is a mapping of an ID and the entry. The entry contains the value being stored as well as the state of the owned reference. Performing `get` will return the ID to use next as well as the current mapping of owned reference ID's and their entries.

The ID of the thread which created the owned reference must be stored. This is done in order to prevent child threads from using the owned references that were potentially inherited through the name-spacing scope of their parent threads. The thread ID field of the entry is set when the owned reference is initially created. This requires an `IO`

operation to be performed in order to get the current thread ID of the thread creating the new `ORef`.

The entry that will be inserted into the new mapping will have a `Flag` which is set to `Writable`. The new store of entries, as well as the incremented ID, are put back into the state. The final step is to return the new `ORef` so that it can be used by other ownership operations.

3.1.3 Dropping an Owned Reference

Owned references can be removed from the context of the ownership monad. A user may want to explicitly remove a reference that is no longer needed but the ability to destroy an owned reference is also needed by other operations.

A user can destroy an owned reference using the drop operation on the owned reference. For a reference `x` that already exists in the same context as the drop operation this can be done as follows:

```
dropORef x
```

Here the *drop* operation will explicitly remove an owned reference from the ownership monad it previously existed in. This will destroy the resource from the point of view of the other operations in that ownership context.

```
dropORef :: Typeable a => ORef a -> Own ()
dropORef oref = do
    ok <- checkORef oref
    case ok of
      False -> lift $
        left "Error during drop operation.\
             \ Make sure ORef intended to be dropped is writable\
             \ and within this thread."
      True -> do
        setORefLocked oref
        setValueEmpty oref
```

Any further operations that try to use the dropped reference will be prevented from occurring:

```
1 startOwn $ do
2   ref <- newORef "hello"
3   dropORef ref
4   _copy <- copyORef ref
5   return ()
```

The copy operation in this example expression will not create a copy of a dropeed reference. Instead, because of the earlier drop operation, the copy operation will cause this expression to evaluate to a `Left` value: `Left "Error during copy operation"`.

For a drop operation to occur the reference must be in the same thread as the drop operation and the owned reference must have a `Writable` flag. In the previous example the reference was writable prior to being dropped. These two conditions are internally checked by the `checkORef` helper function:

```
checkORef :: ORef a -> Own Bool
checkORef oref  = do
  entry@(Entry _f thrId _v) <- getEntry oref
  liftIO $ do
    threadId <- myThreadId
    return $ (threadId == thrId) && writable entry
```

If the ownership system rules are not being violated then the ORef will be dropped by having it's flag set to `Locked` and the value in the entry set to `Nothing`. The entry is not deleted from the internal mapping of ID's to entries. Doing so allows the ownership system to determine if an ORef has been dropped or if it never existed.

The resource space inside the entry of a dropped reference can be freed since the Haskell runtime will garbage collect the previous value that has now been set to the `Nothing` case. The ability to drop an owned reference from a context will be used later within larger abstractions.

3.1.4 Copying a Reference

A user may want to duplicate a reference in order to have two references that at one point were copies of each other. This would be trivial to do with regular variables but the ownership rules which do not allow more than one binding to a resource complicate the task for owned references.

A simple assignment of a new reference to the old reference would mean that the two references were now referring to the same underlying resource. While it's possible to make a copy of a variable this way we would not be able to make any of the claims of resource use or safety.

To create a new owned reference that is a copy of an existing reference there is the copy operation. For an owned reference `x` we can make a copy `y` using `copyORef` as follows:

```
y <- copyORef x
```

This introduces the new reference `y` into the monadic context. Initially `y` is a copy of `x` but since the references are now separate entities they are able to diverge in terms of what their internal values are following the completion of the copy operation.

The copy operation allows the underlying resource owned by a reference to be copied by other references if they are within the scope of the same ownership monad. When this occurs a new reference is created and then given ownership over a copy of the resource. After a copy operation is performed the two references will each own what are now two separate and different resources.

This property can be observed when a reference is copied (creating another reference) and then the original reference is mutated:

```
1 startOwn $ do
2   x <- newORef (1 :: Int)
3   y <- copyORef x
4   let f :: Int -> Own Int
5       f i = return (i+1)
6   borrowORef' x f
7   xContents <- readORef x
8   yContents <- readORef y
```

```
9   return (xContents, yContents)
```

This expression will evaluate to `Right (2,1)`. After the reference `y` was created by making a copy of reference `x`, the contents of reference `x` was mutated in a borrow operation (borrow operations will be explained in depth later in the report.) Because of this mutable action the contents inside the copy and the original reference diverged.

```
copyORef :: ORef a -> Own (ORef a)
copyORef oref = do
    (new, store) <- get
    entry <- getEntry oref
    ok <- inThreadAndReadable oref
    case ok of
      False -> lift $ left "Error during copy operation"
      True -> do
        let newEntry = setEntryWritable entry
        put (new + 1, insert new newEntry store)
        return (ORef new)
```

A copy can be made if the underlying owned reference is at least readable and is in the same thread as the copy operation. To perform a thread ID check the `inThreadAndReadable` function needs to be lifted from the `IO` monad. The `inThreadAndReadable` function will return true if the owned reference is at least readable and in the same thread.

```
inThreadAndReadable :: ORef a-> Own Bool
inThreadAndReadable oref = do
  entry@(Entry _f thrId _v) <- getEntry oref
  liftIO $ do
    threadId <- myThreadId
    return $ (threadId == thrId) && readable entry
```

An owned reference which is set to readable signals that there may be other operations using the resource but in a way that is immutable. If an ORef is writable this indicates that no function is currently using the resource in a way that may mutate the value.

If the copy operation is able to occur the new owned reference is inserted into the internal store for that monad. The new owned reference created from the copy operation is set as writable. The `setEntryWritable` function creates a duplicate entry with its flag set to writable. This action is performed even if the original reference was only readable.

For those familiar with the terminology from the Rust programming language, the term *copy* here is not the same as a copy in Rust. Rust makes a special distinction between making a copy of resources that are fixed in size [1] and making a copy of resources which are more complex and not fixed in size. For the latter case it is still possible to copy these kinds of resources but these need to be cloned (using the clone function) otherwise Rust will consider these values to have been moved. [7] With this library there is only one version of a copy and it creates a new resource identical to the original; there is no distinction given to the kinds of resources that are being copied.

3.1.5 Moving Ownership

It may be necessary to transfer a resource from one reference to another. Under these circumstances the old reference would need to be dropped after the resource has been transferred, otherwise the old reference and new reference would be referring to the same resource.

Here the resource owned by the reference `old` is transferred to the reference `new` and in the process the reference `old` is dropped:

```
new <- moveORef old
```

A move operation transfers a resource owned by one reference to a new reference. An existing reference can also be overwritten by the moved contents of another reference using `moveORef'`. After a move operation is performed it will no longer be possible to refer to the underlying resource through the old reference. This operation removes the old reference from the scope of the ownership monad it previously existed in and the new reference is now the sole owner of the resource.

```
moveORef :: Typeable a => ORef a -> Own (ORef a)
```

[1] Rust will also consider an assignment operation to be a copy instead of a move if the **Copy** trait or the **Clone** trait is implemented for that type of resource. [6] [5]

```
moveORef oldORef = do
    ok <- checkORef oldORef
    case ok of
      False -> lift $
        left "Error during move from an oref to a new oref\
             \ check entry failed for the existing (old) oref."
      True -> do
        new <- copyORef oldORef
        dropORef oldORef
        return new
```

The implementation of `moveORef` verifies that the old ORef is writable. The old ORef must also exist within the same thread as the ownership monad in which the operations are being performed in.

The ownership system needs to enforce that the old ORef is writable in order for this operation to occur because this indicates that the resource has not been dropped and is still valid to use in this context. The ORef must also be writable in order to ensure that another operation is not currently using the resource in a mutable way.

If the old ORef is valid and can be used in the operation, a copy of it is made with the `copyORef` function. After the copy is complete during the move operation, the old ORef is dropped using the `dropORef` function. This removes the old ORef from that monad and prevents other operations from referring to the resource through the old ORef. The old ORef and its entry have been dropped and internally within the state of the monad are set to `Nothing`. The resource space inside the entry corresponding to the old ORef can be safely freed.

There is a key difference between moving a resource from an existing reference and copying it to a new one. Functionally a resource that is copied is cloned and duplicated; doing this doubles the space and creates a new resource. A moved resource by comparison doesn't change. Instead what is altered is the record of who owns that resource. Neither operation, moving and copying, creates a situation where more than one reference owns a resource.

```
1 startOwn $ do
2   x <- newORef "greeting"
```

```
3   y <- moveORef x
4   yContents <- readORef y
5   return yContents
```

This expression does not violate the ownership rules and will evaluate to `Right ‘‘greeting’’`. If, for example, an expression tried to use a moved resource through an old reference binding, the expression will evaluate to a `Left` value such as for the following code:

```
1 startOwn $ do
2   x <- newORef "greeting"
3   y <- moveORef x
4   xContents <- readORef x
5   yContents <- readORef y
6   return (xContents, yContents)
```

This expression violates the ownership system rules and will evaluate to a `Left` because the first `readORef` operation used is attempting to read the resource in `x` that has been moved to `y`.

3.1.6 Writing to a Resource

The write operation exists in order for a user to be able to write over the current value in an owned reference with a new value. This provides a way to safely mutate the contents of the reference while at the same time discarding it previous contents.

For example in the following expression the contents of `ref` will be destructively updated with the new contents. This expression does not violate any ownership rules and will evaluate to `Right ‘‘Some new contents’’`.

```
1 startOwn $ do
2   ref <- newORef "Original contents"
3   writeORef ref "Some new contents"
4   readORef ref
```

Write operations will follow the ownership rules. Consequently the operation will not allow a resource to be updated if the reference no longer exists. If the reference in the

previous example was dropped prior to the write operation the expression will evaluate to a `Left` value. The ownership monad will prevent the write operation from occuring because a dropped entry will no longer be writable.

```
1 startOwn $ do
2   ref <- newORef "Original contents"
3   dropORef ref
4   writeORef ref "Some new contents"
5   readORef ref
```

The value within the resource can be updated and changed through the reference that owns the resource. This is to say that the underlying resource can be changed by referring to its owner as long as it in scope and writable. This operation can be performed safely because the usage of the underlying resource is tracked by the ownership monad.

```
writeORef :: Typeable a => ORef a -> a -> Own ()
writeORef oref a = do
    ok <- checkORef oref
    case ok of
      False -> lift $
        left "Error during write operation. Checking if the entry\
             \ could be written to or if it was in the same thread\
             \ returned False."
      True -> setValue oref a
```

The reference that is being changed cannot have any functions using the `ORef` and it must be in the same thread as the write operation. When the operation is performed the existing resource inside the `ORef` is over written by the new value. This naturally limits the user to overwriting the referrence with a new value without any regards to what the previous value was inside the reference. In order to alter the contents of a reference with a function based on what the contents of the reference presently is, the user must use a borrow operation.

3.1.7 Borrowing a Resource

Borrowing a resource is a library operation which allows a function to be able to use the contents of an owned reference and have protect access to the current value of the reference. The function that is granted access to the resource will be required to remain in the context of the ownership monad.

As a result, this operation can be used to lend out the contents of a reference temporarily to be able to inspect its current contents in a print statement. While the resource is lent out other operations will be prevented from using the reference.

```
1 startOwn $ do
2   x <- newORef "Hello from inside the reference"
3   let f :: String -> Own ()
4       f a = liftIO $
5               putStrLn $ "The contents of the ORef is: " ++ a
6   _ <- borrowORef x f
7   return ()
```

The previous expression will evaluate to `Right ()` and the contents of the reference will be printed to standard output. This would also be the outcome in a situation in which a long running function borrowed the resource.

```
1 startOwn $ do
2   x <- newORef "Hello from inside the reference"
3   let f :: String -> Own ()
4       f a = liftIO $ do
5         putStrLn $ "The contents of the ORef is: " ++ a
6         threadDelay 1000000   -- delay for 1 second
7   _ <- borrowORef x f
8   contents <- readORef x
9   return contents
```

Because of the monadic sequencing of operations in a single thread, the outcome of a long acting function borrowing a function will be the same as that of a short acting function. The previous expression will print the contents of the referrence to standard output, delay for some time, and will then evalutate to a `Right` value. For single threaded

operations the ownership rules will not be violated if a long running function borrows the resource inside a reference.

The borrow operation is similar to passing a value to a function as a mutable borrow in the Rust language. As we saw with the example this library takes a slightly different approach: instead of letting variables borrow a resource, a borrow operation instead lends the resource to a function which temporarily borrows the resource in order to use it and potentially mutate it in place.

```
borrowORef :: Typeable a => ORef a -> (a -> Own b) -> Own b
borrowORef oref k = do
    ok <- checkORef oref
    case ok of
      False -> lift $
        left "Error during borrow operation.\
             \ The checks for if the entry was in the same thread\
             \ as the borrow operation and if the entry could be\
             \ written to returned false."
      True -> do
        setORefLocked oref
        v <- getValue oref
        b <- k v
        setORefWritable oref
        return b
```

A borrow operation can occur if the owned reference is writable and in the same thread as the operation. While the resource is being borrowed it is prevented from being written to or read by other operations. The `setORefLocked` function adjusts the flag on the entry inside a owned reference to locked.

The function that is passed in a borrow can regard the value it sees as the current state of the resource in the `ORef`. This differs from reading a reference and making a copy of its value which will be discussed next.

The value inside a reference uses the internal library function `getValue`. This will raise an error condition if an attempt is made to get an empty `Nothing` value.

```
getValue :: Typeable a => ORef a -> Own a
getValue oref = do
  e <- getEntry oref
  v <- liftIO (value e)
  case v of
    Just a -> return a
    Nothing -> lift $ left "Cannot retrieve the value of an empty ORef"
```

The internal `value` function is responsible for reading a value from the `IOREf` it is stored in and reifing it to a concrete type. This function handles the failure conditions that could come about from casting the value inside an entry to a concrete type.

```
value :: Typeable a => Entry -> IO (Maybe a)
value (Entry _ _ (Just ioref)) = do
  v <- readIORef ioref
  case cast v of
    Just a  -> return (Just a)
    Nothing -> error "internal cast error"
value (Entry _ _ Nothing) = return Nothing
```

The function that borrows the resource is of type `(a -> Own b)`. The resource being consumed in the function remains inside of the context of the ownership monad while the function executes.

Much like the Rust language will not allow for a mutable resource to be lent to multiple borrowers - neither will the borrow operation on an `ORef`. This will ensure that the resource is not mutated or written to while is it lent out to the borrowing function. Because each borrow operation occurs within the context of the ownership monad the resource usage can be tracked and this property can be ensured. The reference that owns the resource will track the resource while the function executes.

The borrow operation also ensures that the original `ORef` does not go away before the function borrowing it is complete. This will make sure that the function is not referring to an `ORef` that no longer exists. The `ORef` that is being borrowed by the function is not able to be moved (or otherwise dropped) before the function that the resource has been

lent to is complete. This is enforced by the move and drop operations which individually check that a `ORef` does not have any functions borrowing the resource.

This library does not allow multiple functions to simultaneously perform *borrow* operations on an existing `ORef`. This restrictions means that the borrowing function is assumed to mutate the value in the reference but it is not required to do so.

For convenience a second variant of the borrow operation is provided which updates the contents of the borrowed reference with the result of the function. The type signature ensures that the function borrowing the reference will have the same resulting type as the reference:

```
borrowORef' :: Typeable a => ORef a -> (a -> Own a) -> Own ()
```

3.1.8 Reading from an ORef

A read operation is slightly different than a borrow operation. Where a borrow operation gives a function exclusive and monitored access to a resource, a read operation will take a snapshot of the current contents. The snapshot is returned but no guarantees can be made about the current contents of the `ORef` after the read operation has completed.

```
1 box <- newORef True
2 schrodingers_cat_alive <- readORef box
```

Reading from an `ORef` will reveal what the contents of the `ORef` was at the time the read was performed but the `ORef` will continue to exist after the read operation. Because of this the value inside the `ORef` may later change.

The advantage of using a read instead of a borrow is that the read operation will remove a copy of the resource from the confines of the `ORef`. This copied value is freed from the restrictions imposed on values contained inside references by the ownership system. As previous examples for the other ownership operations have shown, read operations are useful for extracting the final value from a reference at the completion of a series of ownerhip monad operations.

```
readORef :: Typeable a => ORef a -> Own a
readORef oref = borrowORef oref return
```

3.2 Internal Implementation

The operations discussed so far which make up the public facing API are available to the user but the implementation of these functions is hidden. Apart from ownership monad type `Own`, the data structures and functions used in the implementation of the internal module would also be invisible to the user of the library. The first major piece of the internals is the ownership monad. The type of the ownership monad internally is:

```
type Own a = StateT (ID,Store) (EitherT String IO) a
```

State is represented using the `StateT` monad transformer. The ownership monad needs to track the state of ownership system operations that occur within its context but it does not need to be aware of the operations occurring in other ownership monads. This latter aspect will become important when discussing Owned Channels between separate threads later on.

The state in the ownership monad is comprised of the next `ID` to use and the current `Store`. The `Store` maps each unique reference `ID` to an `Entry`.

```
data Entry =
  forall v. Typeable v => Entry Flag ThreadId (Maybe (IORef v))
```

The first notable parts of the `Entry` datatype are the explicitly quantified `forall v` along with the `Typeable` typeclass constraint on the type variable `v`. The combination of these allows for the references in the state to be heterogeneous. Otherwise it would be necessary to limit the `Store` to owned references of only one type per monadic state. Using the `Typeable` typeclass does require that values in each entry are `cast` in a type-cast operation. However this is handled internally by the library when values are retrieved from an entry. The `cast` function reifies the generic type `v` into a real type. Doing so is necessary because it allows the value inside an entry to be used as a concrete type rather than a polymorphic value.

Using the existentially quantified `forall v` also produces the ancillary benefit that the `Entry` datatype does not need a type variable. As a result, the `Store` type does not need to be a parameterized abstract data type and can instead be a simple mapping between an integer ID and an entry.

```
type Store = IntMap Entry
```

Each owned reference `Entry` maintains a `Flag` to indicate the level of access currently allowed by the ownership monad on the entry's value.

```
data Flag = Locked
          | Readable
          | Writable
```

The ID of the thread which owns the entry is also stored as part of the `Entry` datatype.

The value `v` in an `Entry` is maintained within the `(Maybe (IORef v))` field of the entry datatype. The main purpose of using the `Maybe` datatype is it allows an empty entry to be represented. Empty entries result when entries are dropped from one ownership monad's context. This allows the Glasgow Haskell Compiler to know that the runtime memory storing the prior value in an entry can be freed.

Internally the value stored in a non-empty entry is placed in an `IORef`. An `IORef` in Haskell is a mutable reference in the IO monad - an IO reference. The ownership monad uses entries to store the values assigned to owned references. Even though the values in entries are mutable because of the internal `IORef` implementation, they are encased in the ownership system types which control access to the value.

3.2.1 Running the Ownership Monad

An ownership monad expression can be evaluated using the `startOwn` function. This will run the expression in an initially empty ownership context, one that does not have any existing owned references.

```
startOwn :: Own a -> IO (Either String a)
startOwn x = runEitherT (evalStateT x (0, empty))
```

The `evalOwn` function can be used to evaluate an ownership computation with the initial context passed as an argument.

```
evalOwn :: Own a -> (ID,Store) -> IO (Either String a)
```

```
evalOwn actions startState =
  runEitherT (evalStateT actions startState)
```

The `continueOwn` function can be used to evaluate a nested ownership computation from within an existing ownership monad. This will use the existing monad as the context to run the nested ownership computation.

```
continueOwn :: Own a -> Own (Either String a)
continueOwn x = do
  s <- get
  liftIO $ runEitherT (evalStateT x s)
```

Functions which operate in the ownership monad, or small ownership expressions, might exist which a user would like to run in a separate thread. The `forkOwn` function provides a way for an ownership expression to be evaluated in a separate thread.

```
forkOwn :: Own a -> Own ()
forkOwn innerOps = do
  _ <- liftIO $ forkIO $ do
    childResult <- startOwn innerOps
    case childResult of
      Left violation ->
         putStrLn $ "A child thread failed with the following: " ++ violation
      Right _ -> return ()
  return ()
```

The ownership expresion of type `Own a` is passed as an argument. The `forkOwn` function evaluates the ownership operations using a new ownership monad state. The child thread will execute the expression it is given and will print debugging information to standard output in the case of an ownership violation.

3.3 Mitigating Deadlock

Typically mutable resources are protected by granting exclusive access of the resource to one thread at a time. Other threads will have to block until the thread, which possesses access, releases the resource.

This form of concurrency control is often referred to as a mutex, a portmanteau of the words mutual and exclusion. While a mutex is enough to prevent a data race, where two resources have unfettered access to the same shared resource, without careful consideration a mutex can easily create a deadlock situation.

Deadlock situations are common when shared resources, protected through mutual exclusion, are nested. MVar's (mutable variables) in Haskell can be used to demonstrate such a situation:[2]

```
1  nestedResources :: MVar Int -> MVar Int -> IO ()
2  nestedResources outerResource innerResource = do
3    modifyMVar_ outerResource $ \outer -> do
4        yield
5        modifyMVar_ innerResource $
6                        \inner -> return (inner + 1)
7        return (outer + 1)
8    return ()
9
10 deadlockMVar :: IO ()
11 deadlockMVar = do
12   resourceA <- newMVar 0
13   resourceB <- newMVar 0
14   forkIO $ nestedResources resourceA resourceB
15   forkIO $ nestedResources resourceB resourceA
16   return ()
```

The `yield` function is used here to force a context switch to the other available runnable thread. Lock inversion deadlock in general, and in this example specifically, is created because in order for the function to release the outer resource (if it possesses it) it must gain access to the inner resource. In a multi-threaded program, one thread may gain access to one resource, while at the same time another thread gains access to the other resource. In such a situation both threads now need what the other thread has in order to release their resource they already hold.

[2] This example of lock order inversion deadlock adapted from the example of deadlock from Chapter 24 of Real World Haskell [9]

This kind of bug is difficult to find and debug because it cannot be dependably reproduced. The reason for this is that the situation described in this example which creates the deadlock will not always occur. This contributes to deadlock bugs being difficult to isolate and fix when they do occur.

This example can be adapted to work with owned references. When this is done, instead of spasmodically creating a deadlock condition, it will always create an ownership violation and fail with a description of the violation. Instead of having to remember what order to acquire locks and where the resources are being accessed in their program, the library user can fix the ownership violation they introduced into their multi-threaded program:

```
1  nestedORef :: ORef Int -> ORef Int -> Own ()
2  nestedORef outerRef innerRef = do
3    borrowORef' outerRef $ \outer -> do
4      liftIO $ yield
5      borrowORef' innerRef $
6                    \inner -> return (inner + 1)
7      return (outer + 1)
8    return ()
9
10 deadlockORef :: Own ()
11 deadlockORef = do
12   orefA <- newORef 0
13   orefB <- newORef 0
14   forkOwn $ nestedORef orefA orefB
15   forkOwn $ nestedORef orefB orefB
16   return ()
```

This example produces an ownership violation because owned references are only owned by the thread which created them. The contents of the owned references, while internally mutable, cannot be shared across a thread as a mutable value in the same way that MVar's or IORef's can be. This restriction is intentional; the mutability only exists within the scope of the monad which owns the `ORef`.

The issue presented in the earlier locking access example with MVar's is rooted in the

ability to share access to mutable resources in un-intended ways. In comparison, while owned references are mutable (as we have seen with their internal `IORef` implementation) this is wrapped in the ownership system rules.

3.3.1 ORef's in Multi-threaded Programs

As the previous example demonstrated owned references are only valid within the scope of the thread that created them. But this does not preclude them from being used in multi-threaded programs. In order to do that, a way to give up ownership of an `ORef` when it's been shared between threads is needed. The ownership system needs to prevent a situation in which a thread attempts to use a resource that the thread has given up ownership over by sharing.

Chapter 4: Owned Channels

4.1 Introduction

Owned Channels expand on the concept of using channels between threads to write to and read from a shared location. As with traditional channels, this location can be used by concurrent threads in order to share resources and to communicate.

Instead of sending the resource across a channel, owned channels send ownership of the resource as well. The key idea behind owned channels is for the thread sending a resource across a channel to relinquish ownership over the resource. A thread reading a resource from the channel will automatically gain ownership over the resource it consumes from the channel.

Once a thread has sent a resource over the channel, later operations in the thread will be prevented from using that resource. If the thread needed to use the resource again, it would have to read the resource from a channel and reclaim ownership over the resource.

4.1.1 Motivation

With traditional concurrent channels, the thread that originally wrote a mutable resource to a channel would retain access to the resource. This is because the variable bound to the resource is still valid in the scope of the code which wrote to the channel.

As a result it would be perfectly legal to write code that later referred to the resource through an existing variable binding in the original thread. Channels from the collection of concurrency abstractions available in the base libraries of Haskell do not track resource ownership as it shared between threads. In order to enforce which thread owns which resource, there would need to be some way to track not just what a resource is but also what variable (and what thread) owns a resource.

4.1.2 Using Owned References to Construct Owned Channels

As discussed earlier, the functions to operate on owned references enforce how resources may be used within the context of the ownership monad. It is possible to use the resources in owned references safely, knowing that any violations of the ownership rules will be detected. For that reason, owned references provide a useful building block for constructing larger abstractions that are concerned with tracking resource ownership.

One such area of concern is how to enable expressions in separate threads to access the same resource. Multiple (potentially mutable) concurrent expressions can evaluate a shared resource as long as they do not do so simultaneously. It is well understood that problems arise when threads attempt to access a shared resource without a control mechanism. Likewise, a deadlock situation can accidentally be created when access to shared resources is hoarded by a single thread. As the chapter introduction alluded, one solution to this problem is to create a system for sharing access to resources between channels which tracks both what the resources are as well as who owns the resources.

Using owned references as a building block it is quite easy to create such a system. Additionally it is possible to do so on top of the existing channel interface and the concurrency abstractions which the base libraries in Haskell provide.

4.2 Owned Channel Operations

The major idea that owned channels take advantage of is that ownership of resources can be tracked and used in isolation. Each ownership monad only needs to concern itself with the references that are in the scope of that monad.

For that reason multiple threads can exist and each can have their own ownership monad without sacrificing the ability to reason about resource use. The resources available to each of these threads are covered by that thread's ownership monad. The act of giving up ownership (on write operations) is enough information to facilitate the transfer of resource ownership between threads.

Owned channels are the abstraction built on top of the ownership monad which handle the inter-thread ownership information. The operations are intended to be similar to the channel operations from `Control.Concurrent.Chan` in the base libraries of Haskell. There are operations for creating an owned channel, writing to the owned channel and

reading from an owned channel:

```
type OChan a = Own (Chan a)

newOChan   :: OChan a
writeOChan :: Typeable a => OChan a -> ORef a -> Own ()
readOChan  :: Typeable a => OChan a -> Own (ORef a)
```

Owned channel operations detect ownership violations and prevent the accidental shared ownership of a resource. This allows each thread to be able to acquire or give up ownership of resources in a safe manner. The read and write operations will only allow the user to read or write references to the channel. Read and write operations are also restricted to the context of the ownership monad.

4.2.1 Writing to an Owned Channel

Writing to an owned channel takes the contents of an `ORef` in one thread and writes it to the channel - the shared state between the threads. From the view-point of the thread that wrote to the channel, this operation consumes the `ORef` and the thread loses the ability to further use the `ORef` in later operations.

```
writeOChan :: Typeable a => OChan a -> ORef a -> Own ()
writeOChan ch oref = ch >>= (\x -> writeOChan x oref)
```

The `writeOChan` function takes an owned channel and writes the resource to it. Because the owned reference only exists within the ownership monad it is not necessary for resource safety to use an `OChan` and a regular `Chan` type can be substituted in it's place if used in combination with the `writeOChan'` function. Using an `OChan` with `writeOChan` saves the user from having to lift the regular channel out of the IO monad within their ownership monad expression and is provided for convenience.

This write operation is composed from the combination of two `ORef` operations, a borrow followed by a drop operation, although these `ORef` operations are hidden from the user of the `OChan` library.

```
writeOChan' :: Typeable a => Chan a -> ORef a -> Own ()
```

```
writeOChan' ch oref = do
  borrowORef oref (\v -> liftIO $ writeChan ch v)
  dropORef oref
```

To write an `ORef` to a channel, the owned channel operation needs to use the value that the owned reference refers to. This means that before any further operations can occur the `ORef` must not have any functions currently using its contents. The owned reference must exist within the context of the ownership monad as well. If those conditions are satisfied then the first of the two `ORef` operations can occur.

In order to write the resource inside the `ORef` to a traditional channel, within the owned channel operation, the function writing the resource to the channel needs to borrow the resource. This might seem odd since it would appear that the function borrowing access to the resource intends to send the resource to another thread. This is accurate, the resource that was written to the channel will not be returned and under the rules for a borrow this kind of use is allowed. This is similar to a resource being duplicated using the copy operation, the only difference is that the function which is borrowing access performs an `IO` operation to send a copy of the resource to the channel.

This step stands out because the function borrowing the resource is sharing it with other threads and with different ownership monads. However, this step occurs inside the larger owned channel write operation and will be opaque to the user of the library.

When the resource has been written to the channel it will still exist in the original thread. This is resolved in the next `ORef` operation which is to drop the `ORef` that was just borrowed. This immediately follows the completion of the borrow operation which used the underlying resource in order to send it to the channel. Upon completion, as far as the ownership monad for that thread is concerned, the `ORef` has been consumed. [1]

Because the `ORef` no longer has any functions using it, it is now possible to drop it from that monad using the `ORef` drop operation. This will update the resource ledger for that monad - crucially without touching the resource itself. This prevents any further operations from using the resource in that thread.

[1] The value sent across the channel will not be in an IORef. The borrow operation allows a function act on the contents of the IORef in an ORef entry but not the IORef itself.

```
1 singleThreadedWrite :: Own ()
2 singleThreadedWrite = do
3   ref <- newORef ""
4   writeORef ref "Quark"
5   let ch = newOChan
6   writeOChan ch ref
7   writeORef ref "Odo"
8   -- ^^ writing to a ref that's no longer owned
```

This example will evaluate to a `Left` value which identifies a violation caused by a `writeORef` operation in this monad. After the `ORef`, named `ref`, is written to the owned channel on line 6 it no longer exists within this ownership environment. For that reason when the expression tries to use `ref` on line 7, an ownership violation is detected.

A write operation should totally consume the resource. Even though a read operation seems suitable for the first internal step in a write operation, the borrow operation on owned references is used instead. This is done in order to ensure that the actual value in the owned reference is being used and sent across the channel. The read operation would report what the value was at the time of reading it but other functions could then mutate it in quick succession after the read is complete.

4.2.2 Reading from an Owned Channel

Reading from an owned channel can be thought of as the reverse of writing to an owned channel. Rather than sending the contents of an `ORef` to another thread and giving up ownership of the resource, we are instead taking ownership of a resource from the channel and encapsulating it within a new `ORef`.

```
readOChan :: Typeable a => OChan a -> Own (ORef a)
readOChan oc = oc >>= readOChan
```

A normal channel can be used with the `readOChan'` function because the reference which is read from the channel will remain within the context of the ownership monad.

```
readOChan' :: Typeable a => Chan a -> Own (ORef a)
```

```
readOChan' ch = do
  v <- liftIO $ readChan ch
  newORef v
```

Within the read operation a resource is first read from the traditional channel. This retrieves and removes the resource from the channel - which prevents other threads from then reading that resource.

The second operation is to place the freshly acquired resource inside a new `ORef`. This `ORef` will then be returned into the final ownership monad context. The thread which performed the read operation will now have access to the `ORef` and will be able to manipulate the value inside it using the provided operations for owned references.

```
1  chanTest :: Own ()
2  chanTest = do
3    ch <- newOChan
4    ref <- newORef ""
5    writeORef ref "Quark"
6    writeOChan' ch ref
7    _childThrID <- forkOwn $ do
8      -- in the child thread
9      childRef <- readOChan' ch
10     val <- readORef childRef
11     liftIO $ putStrLn $ "Child thread received: " ++ val
12     liftIO $ do
13       yield
14       threadDelay 3000000  -- 3 seconds
15     writeOChan' ch childRef
16     return ()
17   -- back in the parent thread
18   newParentRef <- readOChan' ch
19   writeORef newParentRef "Odo"
20   return ()
```

The previous example shows a set of operations that will not create an ownership

violation. In this example an owned channel is created in the parent thread and an owned reference is written to the channel. This will remove that reference from the parent environment. The thread is then forked with `forkOwn` which will evaluate the ownership expression passed to it in a child thread. The child thread reads the owned reference from the channel and prints it. Even though the child thread then delays for three seconds (line 14) before writing the reference back to the owned channel, there will not be an ownership violation in the parent thread. The parent thread regains ownership over the reference by reading it from the owned channel an is able to operate on it once again.

4.2.3 Creating an Owned Channel

Creating an owned channel, comparatively, is a very simple operation. A traditional channel is created through an `IO` operation and placed within the ownership monad.

```
newOChan :: OChan a
newOChan = do
  ch <- liftIO newChan
  return ch
```

It was noted earlier that there are write and read functions provided in the `OChan` library that can operate on normal `IO` bound channels. Using a regular channel in the ownership monad, with the write and read operation for owned channels, provides the same ability to reason about resource use and detect violations.

Chapter 5: Conclusion

This project report has demonstrated a monadic library-based approach for tracking resources. Resource management done in this way allows a user to detect when a program tries to a use a resource in a referent with more than one variable binding.

Other languages have approached resource management with similar solutions: notably language level support for ownership typing in Rust as well as uniqueness types in Idris which make use of Idris' dependent type system. This report has demonstrated how a similar approach to tracking resource use can be implemented in Haskell as a library.

The library approach shows how a similar ability for reasoning about resource use can be attained without abstractions which require language level support for affine or linear types. While systems based on these approaches may provide more features for resource tracking, the resource tracking functionality defined in the ownership monad leverages existing language features and libraries in Haskell.

This report has also shown that the operations presented with the ownership monad can be used to construct larger abstractions which concern themselves with resource use.

Bibliography

[1] Linear types. `https://ghc.haskell.org/trac/ghc/wiki/LinearTypes`, 2017.

[2] Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, volume 10, pages 222–236. Springer, 2010.

[3] The Idris Community. Uniqueness types. `http://docs.idris-lang.org/en/latest/reference/uniqueness-types.html`, 2017.

[4] The Rust Project Developers. References and borrowing. `https://doc.rust-lang.org/book/second-edition/ch04-02-references-and-borrowing.html`, 2017.

[5] The Rust Project Developers. Trait std::clone::clone. `https://doc.rust-lang.org/std/clone/trait.Clone.html`, 2017.

[6] The Rust Project Developers. Traits: Defining shared behavior. `https://doc.rust-lang.org/book/second-edition/ch10-02-traits.html`, 2017.

[7] The Rust Project Developers. What is ownership? `https://doc.rust-lang.org/book/second-edition/ch04-01-what-is-ownership.html`, 2017.

[8] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL):66:1–66:34, December 2017.

[9] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[10] Jennifer Paykin and Steve Zdancewic. The linearity monad. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell*, Haskell 2017, pages 117–132, New York, NY, USA, 2017. ACM.

[11] Philip Wadler. Linear types can change the world! In *PROGRAMMING CONCEPTS AND METHODS*. North, 1990.