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 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.

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. In order to allow separate threads to communicate the basic mechanisms provided by Concurrent Haskell introduce mutable state. Adding mutable state and sharing it between threads explicitly introduces side-effects into otherwise pure and referentially transparent computations. 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.

For example a race condition can be created when a mutable reference is shared across a channel between two threads:

```
import Data.IORef
import Control.Concurrent
import Data.Char (toLower, toUpper)
```

```
main :: IO ()
main = do
  ref <- newIORef "test"
  ch <- newChan
  writeChan ch ref

  _ <- forkIO $ do
    ref' <- readChan ch
    modifyIORef ref' (map toLower)

  val <- readIORef ref
  putStrLn val

  modifyIORef ref (map toUpper)
  val <- readIORef ref
  putStrLn val
```

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.

Immutable data structures simplify concurrency and are preferable to mutable structures. However categories of problems still exist where a mutable structure is required.

Linear types are often suggested as a possible solution to limit the issues that result from mutable state [9] 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. [8]

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.

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 starvation. 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 MVar'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: 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 as well as Uniqueness Types from Idris.[7] [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 make use of the idea that by tracking resource use and applying rules to how resources are used - certain properties can be enforced.

2.1 Implementation

The type of the ownership monad:

```
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 provides 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 are created 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. One of the primary motivations for using the ownership monad by itself is as a safer way to introduce mutability, when

it is needed, into a Haskell program. This will be discussed later on when talking about operations for working with owned references and using the ownership monad.

2.2 The Ownership System

Resources are bound to a variable once they are created inside the ownership monad. These variables are the mechanism to access - or refer to - the underlying resource. In the library 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 increases the type safety of the code by requiring that a type be embedded with each `ORef` even though the type variable will not be used on the right side of the definition. This prevents references of different types from being mixed by creating a type error at compile time. The polymorphic `a` allows any type to be stored in an `ORef`. This datatype serves to provide a handle on each resource for the `Own` monad to use as it enforces ownership typing rules and tracks each resource.

When the operations inside that monad are complete the references will no longer exist and the resources will be marked as free. The information inside the resource which is bound to the owned reference 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 prevent violations. There are operations to create owned references, copy them, move the resource from one reference to another, drop a reference from the scope, and write a value to a reference. There are also operations for borrowing the resource inside a reference. This allows for a function to temporarily use the resource without taking ownership over it.

2.2.1 Reference Creation

The `newORef` function creates a new owned reference within the ownership monad and places the value provided to the function inside the reference.

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

The current state of the monad is needed in order to produce the new state which will include the new owned reference. The new owned reference is a mapping of an ID and the entry which contains the value and 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 thread ID of the thread that created the owned reference must be stored in order to prevent child threads from using the owned references that were potentially inherited through the name-spacing scope of their parent threads. This is set when the owned reference is initially created and requires an `IO` operation 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 its read and write flags initially set to true. 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.

2.2.2 Copying a Reference

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

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

The `copyORef` operation checks for an ownership system violation when it is run. A copy can be made if the underlying owned reference is readable and is in the same thread as the copy operation. The thread ID check needs to be lifted from the `IO` monad.

The owned reference needs to be readable for a copy operation to occur. When an owned reference is readable that signals that there may be other operations using the resource but doing so in a way that is immutable.

When an owned reference is readable and writable this indicates that no function is currently using the resource in a borrow in a way that may mutate the value stored in the owned reference.

The `checkEntryReadFlag` function will return true if the owned reference is readable and in the same thread. If the copy operation is able to occur the new owned reference is inserted into the store for that monad. The new owned reference is set as writable even if the reference it was copied from was not.

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

[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]

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.

2.2.3 Moving Ownership

A resource owned by a reference can also be transferred to a new reference or to an existing reference. After this 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 :: ORef a -> Own (ORef a)
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 exisiting (old) oref."
      True -> do
        new <- copyORef oldORef
        dropORef oldORef
        return new
```

The implementation of `moveORef` verifies that the old ORef is readable, and writable. The old ORef also must 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 readable 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 function is not currently borrowing 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 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. In this implementation

the old ORef and its entry remain in the map but it would be safe at this point to garbage collect or free the resource space inside the entry corresponding to the old ORef.

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.

2.2.4 Dropping an Owned Reference

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 :: ORef a -> Own ()
dropORef oref = do
    ok <- checkORef oref
    case ok of
      False -> lift $
                 left "Error during drop operation -\
                      \ make sure old ORef is writable and doesn't have\
                      \ borrowers."
      True -> do
        setWriteFlag oref False
        setReadFlag oref False
```

Any further operations that try to use the dropped reference will be prevented from occurring. In order for an owned reference to be dropped it must not have any borrowers. The reference must also be in the same thread as the drop operation and the owned reference must be readable and writable. If the ownership system rules are not being violated then the ORef will be marked as not readable or writable. The resource space inside the entry of a dropped reference can be freed since it will no longer be used.

2.2.5 Writing to a Resource

A resource can be changed by its owner as long as it does not have any borrowers. The value within the resource can be updated and changed through the reference that owns the resource. 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 borrowers 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.

2.2.6 Borrowing a Resource

Borrowing a resource is the operation that allows a function to be able to use the contents of an owned reference. A resource can be used within the confines of the ownership monad by its owner and a function. This function will be required to return to the context of the original ownership monad.

This operation is similar to passing a value to a function as a mutable borrow in the Rust language. To give some background on what this means: depending on the type signature a function in Rust 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 - in a way - 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 which will track the borrows (with the borrow checker).

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.

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.

```
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 - this occured while\
                      \ checking if the entry was in in the same thread\
                      \ or if it could be read returned false."
      True -> do
        setWriteFlag oref False
        setReadFlag oref False
        v <- getValue oref
        b <- k v
        setReadFlag oref True
        setWriteFlag oref True
        return b
```

A borrow operation can occur if the owned reference is readable, writable, and in the same thread as the operation. While the resource is being borrowed it is prevented from being written to or read. 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` is not able to 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 borrowers before they operate on the `ORef`.

This library does not allow multiple functions to simultaneously perform *borrow* operations on an existing `ORef`. Doing so would prevent the contents of the `ORef` from being a mutable value like an `IORef`. This restrictions means that the borrowing function is assumed to mutate the value in the reference but it is not required to do so. There is no equivalent to a variable being borrowed by more than one immutable borrower in Rust. This could be done by making multiple copies of an `ORef` and then giving these to individual borrowers.

To allow a mutable borrow it is necessary to know if other functions are borrowing the resource. A mutable borrow can only occur if the reference is readable and writable. Each `ORef` already tracks if it is able to be read from or written to.

To allow for a borrower to have mutable access to the reference it needs to be the sole borrower. If a resource can be both read from and written to then we know it doesn't have any borrowers.

2.3 ORef's

TODO ORef's example section

Chapter 3: Owned Channels

3.1 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.

Owned channels operate using the idea that instead of sending just the resource across a channel - send the 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 automatically gains ownership over the resource it consumes from the channel.

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

Traditionally the variable that was written to a channel would still exist in the scope of the code in the thread that originally wrote the resource to a 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. 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.

3.1.1 Using Owned References to Construct Owned Channels

As discussed earlier, owned references provide a fundamental set of rules generalizing resource ownership and how resources may be used within that context. Within the ownership monad it is possible to use the resources in owned references safely knowing that any violations of the ownership rules will be caught and prevented. For that reason owned references provide an useful building block for constructing larger abstractions

that are concerned with tracking resource ownership.

The key to allowing multiple (potentially mutable) concurrent operations to occur on a shared resource is to make sure that they will not occur simultaneously. As the chapter introduction alluded - one solution to this problem is to create a system for shared access to resources that tracks both who owns the resource in addition to what the resource is. Using owned references as a building block it is quite easy to build such a system. Additionally it is possible to do so on top of the existing Channel interface and the concurrency abstractions which Channels provide in Haskell.

3.2 Owned Channel Operations

A major idea that owned channels take advantage of is that ownership of resources - once granted - can be tracked and used in isolation. For that reason each thread can exist inside its own ownership monad bubble and remain isolated from the state of resources which exist in other threads. The act of giving up ownership - on write operations - is enough information to facilitate the transfer of resource ownership between threads. Beyond keeping track of this inter-thread ownership information - which is facilitated by the owned channel operations - each thread will be able to govern its own resources. This prevents any accidental shared ownership of a resource from occurring and does so without having to resort to using an additional form of communication between the threads or a resource scheduler. This saves adding any additional overhead.

3.2.1 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.

This operation can conceptually be thought of as 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.

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 borrowers and it must exist within the context of that ownership monad. 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 - inside the owned channel - the function writing the resource to the Channel needs to borrow the resource. This might seem odd since it would appear that the borrower function does not intend to ever return the resource. This is accurate, the resource that was written to the channel will not be returned; under the rules for a borrow this kind of use is actually allowed. If it were not for the fact that the borrower function acts on resources shared between threads there would not be any issue. Given the nature of concurrent operation between threads - doing this step alone with the function to write to a channel - would be considered dangerous. It is also crucial to remember that this is still occurring inside the larger owned channel write operation and will be opaque to the user of the library.

The second `ORef` operation is to drop the `ORef` that was just borrowed. This immediately follows the completion of the borrow operation. The borrow operation used the underlying resource in order to send it to the channel and upon completion - as far as the ownership monad for that thread is concerned - the `ORef` no longer has any borrowers. [1] Because the `ORef` no longer has any borrowers 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 - and prevents any further operations from using that resource in that thread.

3.2.2 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 take the contents of an `ORef` and give away ownership of the resource inside it - by writing the resource to a channel - we are taking ownership of a resource from the channel and encapsulating it within a new `ORef`.

The first `ORef` operation inside a read from an owned channel is to read from the traditional channel inside the owned channel. This retrieves the resource from the channel and will prevent other threads from reading it. This latter aspect leverages a previously

[1] Since the borrow just performed a dangerous multi-threaded IO operation it is not entirely true to say that it does not have any borrowers - the subsequent drop operation however makes this irrelevant.

existing aspect of Haskell Channels.

The second operation is to place the freshly acquired resource inside a new `ORef`. This `ORef` will then be returned into the ownership monad context by the read operation on the owned channel. The thread can now have access to the `ORef` and the value within it using the provided operations in the ownership monad.

3.2.3 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.

It should be noted that there are write and read functions provided in the `OChan` library that can operate on normal `IO` bound channels. It is recommended to not use these and instead use write and read functions that only operate on owned channel to safeguard against accidentally trying to use the traditional channel write and read functions with `liftIO`.

3.3 Detecting Race Conditions

In the original example in the introduction an undectable race condition was created by passing an IORef across a channel between two threads. Using `OChan` and `ORef` operations it is now possible to pass an `IOREf` between two channels and detect a situation that would result in a race condition before the race condition happens.

This is an example of an operation that will succeed. Here we create an `IORef` and place it in an `ORef`. The parent thread then gives up ownership of the `ORef` when it writes it to the owned channel. The child thread can read the `ORef` from the `OChan` and claim ownership over its contents. The child thread can mutate the resource in a borrow operation and then write it back to the channel.

```
mutableOChanTest :: Own ()
mutableOChanTest = do
  ch <- newOChan

  ioref <- liftIO $ do
    r <- newIORef "hello"
```

```
  return r

ref <- newORef ioref

writeOChan ch ref

_ <- liftIO $ forkIO $ do
  ex <- startOwn $ do
    oref <- readOChan ch
    borrowORef
      oref
      (\x -> do
          liftIO $ modifyIORef x ((++) " from the child thread")
      )
    writeOChan ch oref
    return ()
  return ()

oref' <- readOChan ch
borrowORef
  oref'
  (\x -> do
      liftIO $ do
        contents <- readIORef x
        putStrLn contents
  )
return ()
```

Chapter 4: Related Works

4.1 Uniqueness Types

Idris, which treats Uniqueness Types as a subkind of regular Types, shows the other way of approaching this and the benefits and trade-offs of doing so.

Chapter 5: Conclusion

5.1 TODO

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] 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.

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