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

1.1 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 guaranties that adhering to these rules facilitates offers improvements in specific areas. Furthermore applying this system provides an example for how monadic based systems with fewer restrictions than Linear Types can offer many of the same abilities to reason about resource use without becoming as unwieldly as some linear type systems.

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 are immutable. Because of this there is no inherent state. Haskell's purity therefore allows for referential transparency where values and variables can be thought of as being interchangeable in the sense that under any context evaluating an expression will always lead to the same result. Referential transparency is a very desirable property which allows for a greater ability to reason about the behavior and correctness of a program.

Unfortunately 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 can exist with shared-state concurrency.

Linear types are often suggested as a possible solution for 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 embdedded domain specific languages within a monadic context. [8]

Linear types however can become cumbersome to work with. Other less restrictive forms logic have been used in type systems for similar resource tracking. Affine type systems weaken the restrictions imposed by Linear types system. 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.

Concurrent Haskell programs can still fall prey to the same fundamental problems that other impure languages can, namely deadlock and starvation. This project report will demonstrate what some of these problems look like using basic concurrency tools available in Haskell such as shared state with MVar's and message passing with channels. In Concurrent Haskell the most basic way to allow separate threads to communicate is with a shared mutable variable.

It will then demonstrate and explain the benefits of tracking resource usage with a set of rules similar to affine types. The contribution that tracking resources while they are being used and sent between threads will be shown.

1.2 Contributions

While this library does not do so - by tracking resources with the ownership system it becomes in theory possible to reason about the memory usage over the lifetime of a program using the type system of the language. This method makes it possible to do a form of automatic deterministic destruction instead of the typical garbage collection approaches. This paper will show where in an example program this could occur.

1.3 Background

Approaching resource usage with this style of implementation is not a new concept. Restricting all entities to following the rules specified under a affine type system discipline is applied under the Ownership System in the Rust programming language.

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. By allowing non-unique types to exist and be used along side Unique Types, Idris offers a degree of flexibility with it's approach to Uniqueness Typing that is not present with ours.

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. *Move Semantics* describe the outcome from using the operations provided for the *Ownership System*.

The *Ownership System* and *Move Semantics* this library implements are inspired by the Ownership system in Rust as well as Uniqueness types from Idris.[7] [3] The *Ownership System* described by this paper approximates some of the features from the Rust language but differences between the two result from the different language paradigms and the different use cases.

Uniqueness types in Idris, ownership in Rust, and the *Ownership Monad* 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 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 are called ORef's - or *Owned References*.

2.1.1 Reference Creation

An *Owned Reference* is created within the Ownership Monad and bound to a 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 of the resource within the *Owned Reference* can only be accessed by the provided operations for operating on references within the *Ownership Monad*. These operations will verify whether the ownership rules are being followed.

The newly created reference *owns* the resource it was given when it was created. Resources that are put into references can only have one owner at any given time. This

reference bound to the newly initialized resource becomes the sole owner of that resource.

2.1.2 Copying a Reference

The underlying resource owned by a reference may be copied by other references within the scope of that ownership monad. When this occurs the new references is created and is then given ownership over their 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.

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.

2.1.3 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.

Move operations provide a way for a references to interact with other references and provide a building block for larger more complex abstractions that will be discussed later on.

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

[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]

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.

2.1.4 Immutably Borrowing a Resource

Borrowing a resource is the operation that allows a function to have access 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 that will be required to return the resource to the context of the ownership monad.

This operation is similar to passing a value to a function as an immutable 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. While the resource is being borrowed it is prevented from being written to. The function remains inside of the context of the Ownership Monad while it 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 a `ORef`. The `ORef` 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* which individually check that a `ORef` does not have any borrowers before they operate on the `ORef`.

This library also allows multiple functions to simultaneously perform *borrow* operations on an existing `ORef`. This is equivalent to a variable being borrowed by more than one immutable borrower in Rust. To do this a borrower count is maintained by each `ORef` and the *writable* flag is not reset until this count is zero.

In this library the borrow operation can only read the value and will not allow the resource to be mutated by the function. In order to be able to borrow and mutate the reference a different operation is required.

2.1.5 Mutably Borrowing a Resource

This library also allows a function borrowing a resource to be able to write to (or mutate) the resource in the original `ORef`. In Rust this would be equivalent to having a single mutable borrower. The *mutable borrow* operation in this library permits the function borrowing that resource to read and set the value in the original `ORef`.

To allow a mutable borrow it is necessary to know if other functions are borrowing the resource. Each `ORef` tracks if it has borrowers by keeping a ledger indicating if it is able to be read from or written to and how many living borrowers it has. When a resource has one or more immutable borrowers it is no longer able to be written to - but it can still be read from. The reason why `ORef`'s have two flags - one for read and one for write - is that if a resource can be both read from and written to then we know it doesn't have any borrowers; if it cannot be read from or written to we know it was

either dropped or it currently already has a mutable borrower. In order to allow one borrower to be able to mutate the resource it is required that it is the only borrower of the resource at that time. A mutable borrow is prevented from happening is the `ORef` is not writable.

2.1.6 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.

2.1.7 Dropping an Owned Reference

The *drop* operation will remove an *Owned Reference* from the *Ownership Monad* it previously existing in. This will destroy the resource from the point of view of that Owership context. Any further operations will be prevented from occuring using the dropped reference. In order to be dropped a resource must not have any borrowers, it must exist in that context, and it must be readable and writable. If not the drop operation will not be allowed to occur.

2.2 Move Semantics

2.2.1 Behavior

TODO

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 bewteen 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 - an 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 an immutable 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

[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.

channel and will prevent other threads from reading it. This latter aspect leverages a previously 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 *Owership 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 Preventing Deadlock

TODO

3.3.1 Explicit Resource Ownership Prevents Deadlock

TODO

3.4 Beyond Threads

TODO

How Ownership could be used for resources between processes using D-Bus and also servers in distributed computing.

Chapter 4: Conclusion

4.1 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.