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

Page

LIST OF ALGORITHMS

Chapter 1: Introduction

1.1 Introduction

This paper presents a library for Haskell that implements an *ownership-style* set of rules 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 significant improvements in specific areas.

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 inherent changing state. Haskell's purity 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. This is a very desirable property that Haskell's purity grants and it allows for a greater ability to reason about the behavior of a program.

Unfortunately even in a pure language like Haskell, this property breaks down in the context of concurrency. Concurrency introduces a changing state as separate threads interleave actions. Under some circumstances this can make a program in Haskell look and act as though it were an imperative language.

Concurrent Haskell programs can still fall prey to the same fundamental problems that other impure languages can, namely deadlock and starvation. This paper will demonstrate what some of these problems look like using basic concurrency tools available in Haskell such as shared state with MVars 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. The contribution that tracking resources while they are being used and sent between threads will be shown.

1.2 Additional 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 tradeoffs 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. The *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. [5] [1] The *Ownership System* described by this paper and implemented by the accompanying library approximates some of the features from the Rust language Differences result between the two from the different language paradigms and the different use cases. Uniqueness types in Idris, ownership in Rust, and the *Ownership Monad* all use some form 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 *Onwership 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. [5] 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. [4] [3]

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 Reading a Resource

A resource can be used within the confines of the ownership monad by its owner and a function that will return the resource to the ownership monad. This is similar to passing a borrowed value in Rust to a function.

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 will is automatically returned when the function has finished execution.[5] A function in Rust that takes a borrowed value as an argument is - in a way - syntactic sugar over that function taking ownership of the value and returning it at part of the return value. Instead of having to do this explicitly a value can be borrowed by having the function take a reference to that the value and then the ownership is handed back by 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 if the borrowers never mutate the underlying resource - or we can lend it to a single borrower that will be able to mutate the resource.[2] 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) is not allowed.

This library takes a slightly different approach: a read operation lends the resource to a function which temporarily borrows the resource in order to use it. The reference that owns the resource can only lend the resource out to one function at a time and the function remains inside of the context of the Ownership monad. Much like Rust will not lend out a mutable resource to multiple borrowers - neither will the read operation on a ORef. The key to allowing multiple (potential mutable) operations to occur on a resource is to make sure that they will not occur simultaneously. Because each read 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 and other functions or owned references will be prevented from using the resource as long as long as it is borrowed.

2.1.5 Writing to a Resource

A resource can be changed by its owner as long as it does not have any borrower. The value within the resource can be updated and changed by 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.

2.2 Move Semantics

2.2.1 Behavior

With the Ownership system enforcing the rules dictating how a resource can be used

2.3 ORef's

ORef's section

Bibliography

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

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

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

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

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