Ownership Monad

Michael McGirr
Oregon State University
Graduate School of Electrical Engineering and Computer Science

Abstract

It is well understood that a strong aspect of functional programming languages is *referential transparency*: effectively, an expression will always have the same value when it is expressed.

This principle would make it seem that pure inplace updates of arrays, among other operations, would not be possible. Uniqueness typing preserves referential typing by ensuring that the same expression never occurs more than once.

This paper presents a library for Haskell that implements an ownership-style set of rules for resource-aware programming. We show that while the restrictions that this implementation of Uniqueness Types 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.

Further, our approach shows that when Uniqueness Types are used generally on all variables in a program, rather than selectively, the benefits can be applied to the program wholistically. It also 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.

The ST monad in Haskell produces a similar outcome of making it possible to control memory operations. However where the ST Monad is quite explicit we will show that an ownership system in an EDSL can hide this abstraction. The usage of Uniqueness types is then able to be done transparently without further code annotation on the part of the programmer and the enforcement of the rules can be done at compile time.

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

Previous work done in Haskell for Atom has shown that a Haskell EDSL is entirely suitable for creating hard realtime software that requires constant memory use.

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. This allows the programmer to write most of their code in the usual way and only run into the restrictions that Uniqueness Typing imposes when they choose to. But doing so also incurs the cost that these non-unique types inherently carry by limiting the potential for exercising precise control over memory usage.

Using the type system to encode and explain what is happening with the underlying resources in a general way that can be independently verified and checked greatly increases our confidence and ability to know when a resource is finished being used.