

Updates on the Ownership Monad



2016 Oregon State University
Mike McGirr

Preview

Linear Types, Linear Logic, and Unique Types

Ownership

Borrowing

Lifetimes

Then perhaps why this matters.

Linear Types

Linear Types are types whose instances must be held in Linear Variable's. Linear Variable's must be accessed exactly once in their scope; linear objects' reference count is always exactly 1.

Linear types correspond to **linear logic** and ensures that objects are used exactly once, allowing the system to safely deallocate an object after its use.

Uniqueness types are a variant of linear types.

Uniqueness guarantees that a value has no other references to it, while Linear types guarantees that no more references can be made to a value.

Linear Logic

Jean-Yves Girard's **linear logic** is resource aware, in the sense that premises represent resources that cannot be duplicated or discarded, rather than as truths, which can be reused or ignored.

logician Jean-Yves Girard



Linear logic does this by removing several rules, known as the “structural rules,” from propositional logic.

In particular, linear logic removes:

- **Contraction**, which says that if you can deduce a conclusion A from some premises B, B, C, D , etc., then you can deduce A from B, C, D , etc.
- **Weakening**, which says that if you can deduce a conclusion A from some premises C, D , etc., then you can deduce A from B, C, D , etc.

Unique Types

The Concurrent Clean programming language makes use of uniqueness types (a variant of linear types) to help support concurrency, input/output and in-place update of arrays.



A unique type guarantees that a resource is used with at most a single reference to it.

Rust, Idris, and Clean are some programming languages that support linear or affine types (a version of linear types). In Clean and Idris they are sometimes used for doing I/O operations in lieu of monads.

Why Linear type systems

Linear types can be used to keep track of memory ownership, potentially to support region-based memory allocation, which promises memory safety without the overhead of tracing garbage collection.

In a pure language, they may be used to enable in-place updates of memory.

For example, a functional array update operation can know that the array passed in will not be accessed again, thus the “new” array that it returns may occupy the same memory as the old array, with an in-place update happening under the hood.

Why Linear type systems

In Idris there is some experimental support from writing effectively linear/substructural typing rules for some program state, all within the language, and using it to enforce resource usage or state transitions.

<http://eb.host.cs.st-andrews.ac.uk/drafts/eff-tutorial.pdf>

In its current form it's not all that usable yet and the error messages are positively Lovecraftian.

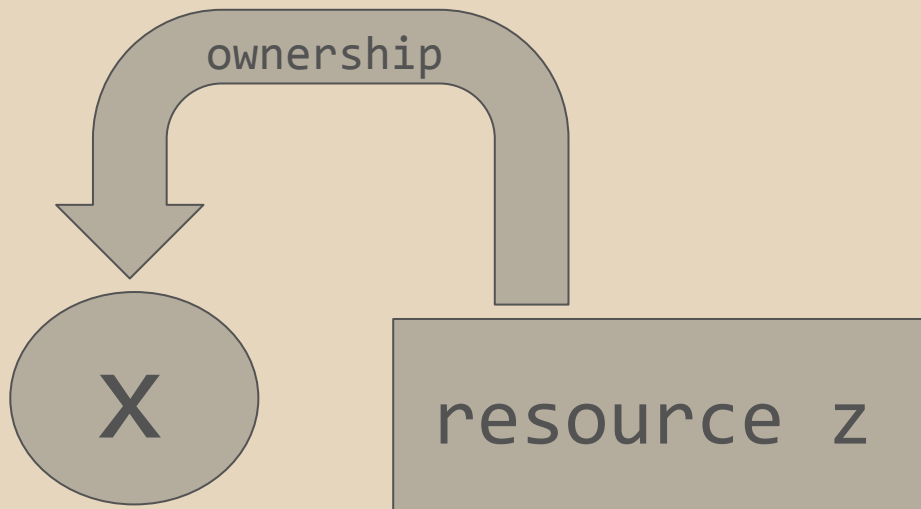


Intro to Rust Ownership

All resources, memory, must be owned by somebody.

```
let x = z;
```

“Create x and
x owns z”

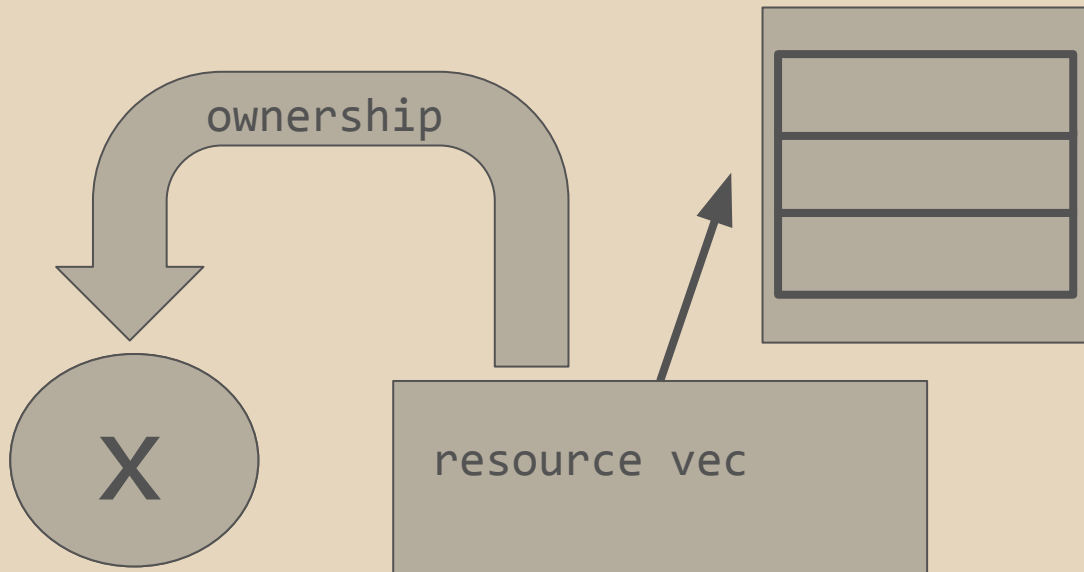


Intro to Rust Ownership

All resources, memory, must be owned by somebody.

```
let x = vec![1,2,3];
```

“Create x and
x owns this vec”

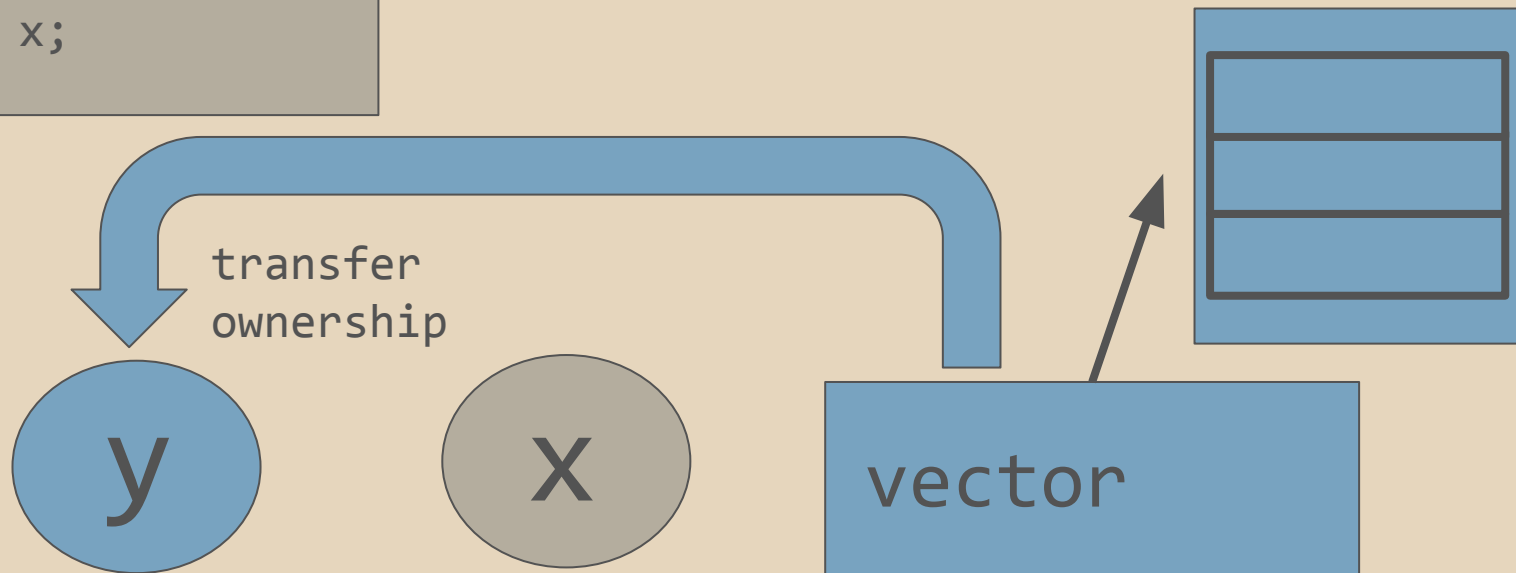


Intro to Rust Ownership

- We may transfer ownership to someone else. The moved values are then owned by someone else.
- We can't refer to the previous owner, since we and the compiler know we wouldn't be referring to anything.
- We can therefore guarantee the safeness of parallel operations since we are always keeping track of this.

Intro to Rust Ownership

```
let x = vec![1,2,3];  
let y = x;
```



Intro to Rust Borrowing

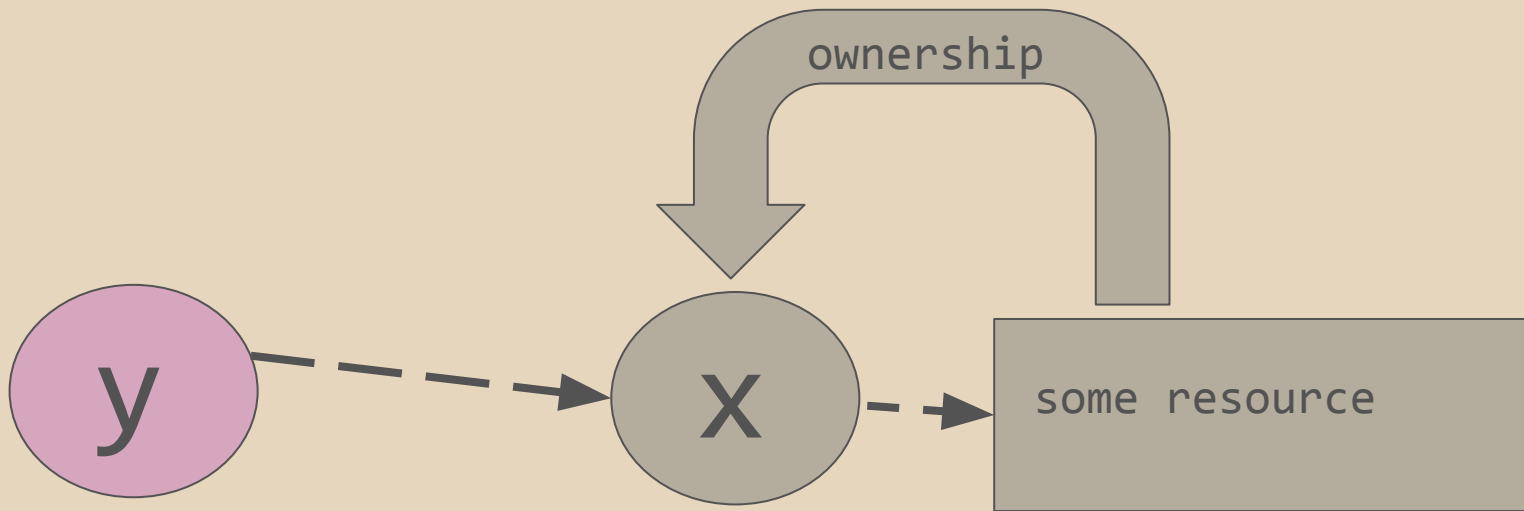
What if we don't want to own some resource permanently, we just want it temporarily?

We can “borrow” the resource from the variable that owns it.

We can lend out access to the resources that we own and Rust's compiler will check that these leases do not outlive the resource being borrowed.

Intro to Rust Borrowing

Borrowing is basically referencing.



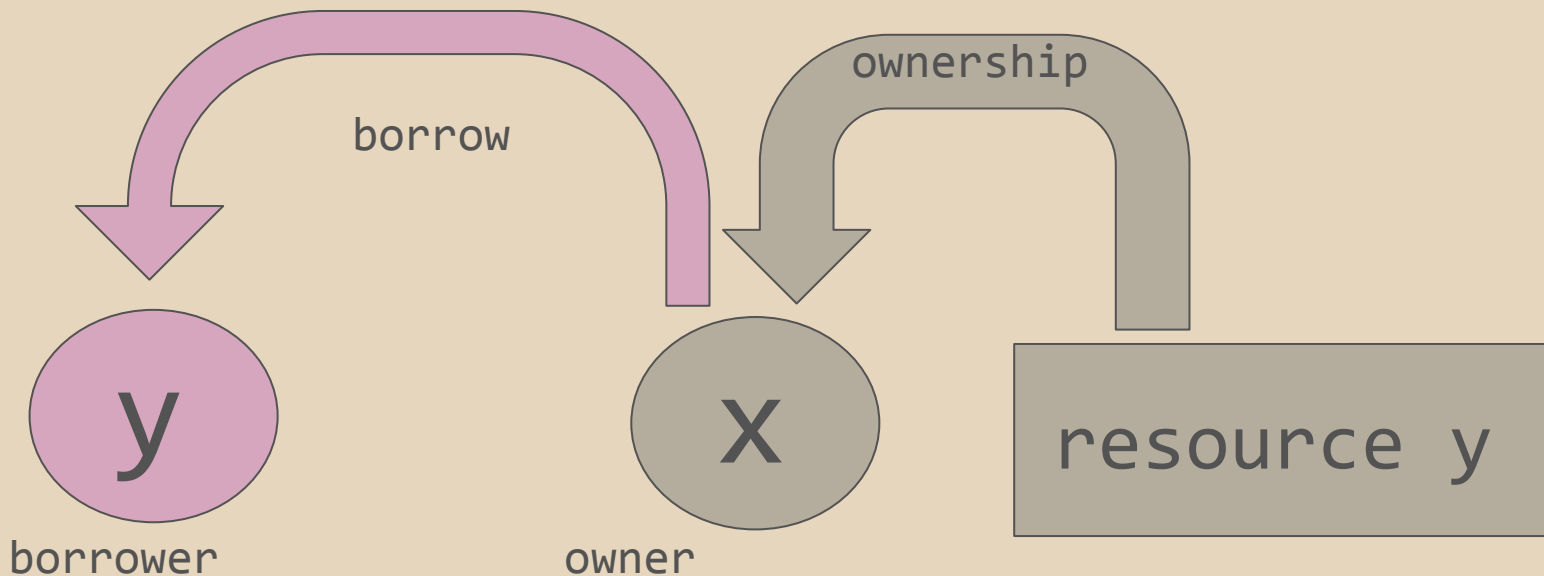
Intro to Rust Borrowing

Borrowing is basically referencing.

```
fn main () {  
    let x = vec![1,1,2,3,5,8];  
    let y = &x;  
  
    let mut acc : i32 = 0;  
    for i in y {  
        acc = acc + i;  
        println!("I found {} in a list now I have {}", i, acc);  
    }  
}
```

Intro to Rust Borrowing

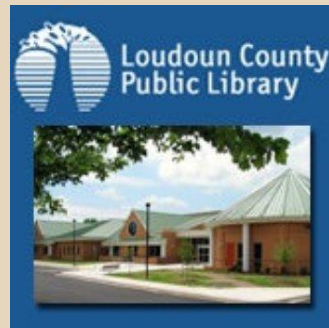
A borrow allows restricted access to the resource.



Intro to Rust Borrowing

Two kinds of borrowing:

1. Default is an immutable borrow.
2. The other is a mutable borrow.



While the resource is borrowed the original owner cannot mutate or destroy the lent out resource.

Only one mutable borrow can be made at a time.

Intro to Rust Lifetimes

In Rust, resources are bound to variables.

A *lifetime* in Rust is way to describe the scope (timespan) in which that variable is valid.

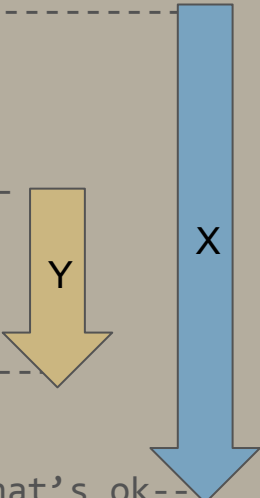
We keep track of this to be able to reason about how long the borrower's need the owner's lifetime to be.

Intro to Rust Lifetimes

At the end of a lifetime the resource is freed.

The variable can no longer be referenced (borrowed) mutably or immutably from any other variable.

```
fn main () {  
    let mut x = vec![1,2,3]; // -----  
  
    { // create a scope  
        let y = &x; // -----  
        // doing stuff with y  
        // x must be kept alive  
    } // y's lifetime ends-----  
    // ok x is no longer needed  
} // x's lifetime ends here and that's ok--
```



Motivation for research

- One of the fascinating ambitions of Rust is to bring linear type systems to the mainstream.
- The programming world, in general, is so unacquainted with linear types, yet the possibilities are staggering.
- Functional programming languages were, in some sense, always based around referential transparency: the property that a name and its value are interchangeable.
- Rust takes another approach: a name is an owner of value, and owners are linear resources.

Is a smaller language dying to break out of this larger one? Practical linear typing implemented on the top of Haskell (or even full-dependent systems) is exciting.

IORef and State Monads

We already have many ways to achieve mutable state in Haskell but let's look at ways applying ownership rules and practical linear typing on top can help.

In Haskell, all variables *are indeed immutable*, but there are ways to construct mutable references where we can change what the reference points to.

The IORef module lets you use stateful variables within the IO monad (Mutable references in the IO monad)

State monad: lets us treat mutable variables as we would in any other programming language, using functions to get or set variables.

Citations

<http://c2.com/cgi/wiki?LinearTypes>

<http://www.slideshare.net/saneyuki/rusts-ownership-and-move-semantics>

<https://nercury.github.io/rust/guide/2015/01/19/ownership.html>