Preface

In this post I will explain a concept for a new Rust language feature I'm calling "generic variables" (previously "associated values"[1]). Generic variables fuse static typing and runtime variables to allow code authors to specify through the type system that a type needs access to existing runtime variables without those variables needing to explicitly passed to functions/methods, or explicitly stored on the type itself. Generic variables aims to reduce duplication of pointers/reference between many instances of a type when it's obvious at compile time that all such instances (within a certain scope) share a common referent. The idea shares some similarities to dependant types, as well as some implementation similarities to "monomorphic data, polymorphic code" [1], [2].

I tried previously to explain this concept on Zulip and received mixed feedback. Many respondents seemed to misunderstand the ideas presented due to it being poorly explained. Accordingly, I didn't receive the volume of helpful feedback I was hoping for. This time I will be explaining the ideas presented using many examples and in much more detail. In general, this is a very, very long post, but it's for a reason. I want to give everyone that reads the whole thing I good of what it is I am suggesting and see if people out their have uses for this feature, and gauge if it might be worth prototyping. So let me know if you think this is a good idea or not, or if you have suggestions to make it better, or some glaring problem I have missed.

All new syntax are purely for demonstration purposes. They could be changed quite easily.

The Problem

A common pattern in Rust is allocating memory from the global allocator and subdividing it to be allocated for a more specialised purpose, using a more specialised allocation scheme. After the execution of side effects, this memory management objects will often return a handle or index that can later be used on the same object that the handle orientated from to manipulate the stored data. This can have several advantages, such increased allocation and deallocation speed, more flexible interfaces for reading and writing stored data, and the ability to track the validity of an access at runtime. These systems of memory management are used when flexibility and control over memory access and layout is of utmost importance. This makes these systems a poor fit for being a single, globally accessible variable, as it precludes having multiple instances at runtime. So if you want to be able to have more then one of these allocators exist at the same time, there are two ways to go about it. One is to store a reference to the relevant allocator object on any type that uses it. This can work but may bloat memory usage significantly if the same allocator is being used by many objects at the same time. The other option is to pass a reference to the allocator to any function or method that might need it. This avoids the memory bloat issue but requires you pass in the context as an argument to possibly many places, which can create noise in the code. It also means generic APIs defined in other places won't be able to use types that store data in your custom allocator.

Motivating example

I think the best examples of this problem are std collections like `Vec` and `BTreeMap`. Basically, `Vec` as it exists currently, is a tuple of a pointer to the allocation, a `usize` representing it length, `usize` representing its capacity, and a `allocator` representing the allocator the `Vec` is allocated from. Below is an illustration of the structure of `Vec`.

```
pub struct Vec<T, A: Allocator + Clone> {
    ptr: *mut T,
    len: usize,
    capacity: usize,
    allocator: A
}
```

Now what if we want to make an allocator that works at a local scope, like in a function. We could create and implement a custom `Allocator` for this purpose. In the following example a custom allocator `CustomAllocator` is defined. It implements the `Allocator` trait, allowing it to be used to allocate memory for std collections. Then a function `use_function_local_alloc` is defined, which creates an instance of `CustomAllocator` and assigns it to the local variable `alloc`. Finally, a 2 dimensional `Vec` is created storing `i32`, using the custom allocator is created. The outer vector stores the x dimension, and the inner vectors store the y dimension.

```
use std::vec::Vec;

struct CustomAllocator {
    /* implementation details */
}

impl std::alloc:Allocator for CustomAllocator {
    /* implementation details */
}

fn use_function_local_alloc() {
    let alloc = CustomAllocator::new();

    let vec_2d: Vec<Vec<i32, &CustomAllocator>, &CustomAllocator> =
Vec::with_capacity_in(10, &alloc);

    for x in 0..10 {
        let mut y_vec = Vec::with_capacity_in(10, &alloc);

        for y in 0..10 {
            y_vec.push(y);
        }

        vec2d.push(y_vec);
    }
}
```

But this isn't exactly ideal, since each `Vec` on the x axis (storing the y values for that x) must store a reference to `alloc`, even though it's duplicated across all such y vectors. We as programmers can see this, and see the obvious inefficiency this entails, but the compiler and the type system have no notion of a local runtime variables shared across instances of a type, so we must settle for duplicating the reference. If we were reimplementing `Vec` ourselves, we could provide a reference to the custom allocator as an argument, but then

we couldn't benefit from all the pre-existing implementations on `Vec`, nor the interoperability with other crates that comes from `Vec`. Conversely, we can't change the implementation of `Vec` to support this without breaking much pre-existing code. This same problem can appear with any std collection and a custom allocator that exists in a local scope.

More generally, the trichotomy of choosing between a global context, a local context with parameter passing, or a local context with reference counting/reference duplication can appear pretty much anywhere you have an application or framework that has to do complex state tracking of dynamically allocated data. Good examples would include UI frameworks and game engines. But I didn't use these as examples as I don't have enough experience with them to say for certain if they would benefit from the features I am proposing. But I suspect at least some of them might.

Variables as types

I believe the solution to this problem is being able to encode variables into the type system, to use variables as types. If we could use variables as types, we could change our vector in the previous example to something like `Vec<Vec<i32, alloc>, alloc>`, using the name of the local variable as a type in and of itself. Then, the compiler could lower the code using `vec_2d` to have an additional implicit argument on its methods, one which provides the pointer to `alloc`, without having to store the pointer to `alloc` on every `Vec`.

Before we can use variables as types, we must be able to name said variables in a way that the type system could make sense of. Specifically, a way to create local variables whose names can't be shadowed. This is because in Rust items can be declared in any order, which is allowed because their names can't shadow each other and because they all exist simultaneously at compile time. Local variables on the other hand can shadow each other, and the order in which they are declared does matter. So if we use the name of a local variable in a type signature, and there are multiple variables with that name, which one are we referring to? Whatever system you use to disambiguate this situation, you will only be able to ever refer to one of the variables. To fix this problem we'll need to add a way to declared local variables whose name can't be shadowed, not only among other local variables, but also items in scope. How this could be done is described next.

The var keyword and var statement

The "var" statement, and its associated keyword "var" are my suggested way to tackle this problem. Syntactically, var statements are written in source code as let statements are, only with the keyword "let" replaced with the keyword "var". Theres the "var" keyword, then the identifier of the variable, then optionally a colon and a type, then an assignment expression. The only differences is that var statements can't have their names shadowed by other items or local variables in scope, and vars can't be mutable. The following example declares a single var `X`, of type `usize`, and assigns it to a value of 1.

```
var X: usize = 1;
```

A note on naming

The "var" keyword was chosen to reflect the variable declaration in imperative programming languages where variables names can not shadow each other, while also acknowledging that the value the var holds can vary at runtime. This reflects the semantics of the var statement, making its usage intuitive. However, the actual

keyword used is arbitrary and is only used to illiterate the ideas presented. "var" also makes for a nice keyword as it can be intuitively reused to declare generic variables, as described later.

Variables as types in Rust

Now we could imagine treating each individual var as its own type, in addition to being a variable with a value. If a type signature uses the identifier of the var as part of its type, you must provide that variable specifically. You could think of this as similar to singletons in OOP languages. In an OOP, singletons are a design pattern in which certain classes can only have one instance existing at runtime. Because singletons are classes, if you type a variable or an argument to be of a class which is a singleton, you can be sure that it is the same instance as anywhere else in the code where that class is used. In this way the name of the singleton refers both to a classes (type) and an instance (variable). What I am suggesting is to take that same idea and apply it to local variables declared using the var statement.

In theory this lets us do some interesting stuff. Take for instance this simple example. A function called `3d_array` takes 3 `usize` arguments, `x`, `y` and `z`, next it declares 3 vars, `X`, `Y`, and `Z`, assigned to their respective arguments. Then it creates a variable `array3d` of type `[[[i32; Z]; Y]; X]`. Breaking that down, we have 3 nested arrays, each with a different length defined by a different variable, `X`, `Y`, and `Z`, representing the side lengths of a 3 dimensional array. In each cell, an `i32` is stored. Notice that `X`, `Y`, and `Z` are all runtime variables, not constants or generic constants. We also have a local functions within `3d_array` called `takes_array3d` defined above everything else, which has a single argument `a` of type `[[[i32; Z]; Y]; X]`. We later call `takes_array3d`, passing `array3d` into it. While `takes_array3d` is declared before any of `X`, `Y`, or `Z` are declared, this is allowed because they are in both the same lexical scope and the same namespace, and thus `X`, `Y`, and `Z` must be unconditionally initialised at some point. If any of `X`, `Y`, or `Z` where declared in a different block, or contained in a `if`, their names would not be in scope.

```
fn 3d_array(x: usize, y: usize, z: usize) {
    fn takes_array3d(a: [[[i32; Z]; Y]; X]) {}

    var X: usize = x;
    var Y: usize = y;
    var Z: usize = z;

    let array3d: [[[i32; Z]; Y]; X] = [[[0; Z]; Y]; X];

    takes_array3d(array3d);
}
```

Making that code compile is not my goal. Specifically, arrays that have a size variable at runtime is not something I am interested in. But they make a nice example and considering how we might implement such a feature makes a natural segue into my real proposal.

So then, if we wanted to support of arrays of a variable length at runtime, how would we need to extend the type system (beyond what I have already shown)? Well the main question is how do handle such arrays if they are not merely assigned to a local variable, but stored as part of a larger composite type? Well, we can already do this if the size of the array is constant, or if it variable at runtime. We can't use a constant because we want the size to vary at runtime. But Rust's understanding of "variable at runtime" is that all instances of the type are uniquely sized, which isn't quite what we mean. So instead we need a different kind of constant, a constant

that can be variable at runtime, but is immutable one declared. We already have this in the for of the `var` statement of course. That will let us create a type where the `var` is in scope and use it to define the length of our array there. We already did this in the previous example on the function signature of `takes_array3d`. This works but no one writes code this way, and for good reason; it's difficult to keep things organised when everything has to be declared as a local item inside a single function. You might notice something else about this approach, that we are taking in arguments to our function, then assigning them to `var`s so we can use them in types. What if instead the `var`s where in scope right from the beginning of the function and were provided as arguments directly, that way we don't have to declare them ourselves and we can use them in the type signatures of our arguments and the return signature.

Now that I have led you through the thought process that led me here, I will present a generalisation of the "variables as types" concept, and show how it could be implemented.

Generics Variables

What we need is the `var`s for a given function to be both in the type signature, and the exact variable being used to be itself variable. What we need is *generic variables*. This kills two birds with one stone, first, it allows `var`s to be passed as arguments to a function, allowing the caller to decide when and where to declare the variables themselves then pass them in as arguments. This fixes the code organisation problem. Second, we can generalise this idea to types as well, where generic variables can act as a type parameter, allow us to create custom types which use variables as part of their definition. Next lets take a look at how that might work.

Syntax and declaration

Generic variables, being a generic, would be declared in the generic clauses of existing Rust syntax. To declare a generic variable, the previously proposed "var" keyword would be used, followed by an identifier, a colon, then a lifetime annotation (which may be optional depending on context), then a type. The preceding may be repeated any number of times, allowing for an arbitrary number of generic variables. In my examples, generic variables will always be the last generics[1]. The type of generic variable may itself be generic, or of a type with a generic variable. The identifier and type are self-explanatory. The lifetime refers to the lifetime of the variable itself, which is based on what scope it was declared in. On types declarations I will always include the lifetime annotation, but I will omit it in most other places.

In the example below serval types with generic variables are declared, presenting different ways to use the feature. `Single` is a struct with a lifetime `'a` and a generic variable `X`, with a lifetime of `'a`. This could be read as "a memory location called `X` with a lifetime of `'a`, storing a value of type `usize`". Next we have `Generic`, a struct with a lifetime of `'a`, a generic type of `T`, and a generic variable `X` with a lifetime of `'a` and a type of `T`, showing how generic variables can have generic data types. Finally, we have a function signature for a function `create_single` which takes a generic variable `X` and returns a `Single` using `X` as its generic variable.

```
// a struct with a generic variable 'X' of the usize type
struct Single<'a, var X: 'a usize>;

// a generic variable having a generic type
struct Generic<'a, T, var X: 'a T>;
```

```
// generic variables on a function signature
fn create_single<var X: usize>() -> Single<X>;
```

Instantiating generic variables

Generic variables can be instantiated in much the same way as constant generics or type generics; through the use of turbo fish syntax. At each position in the turbo fish clause where the type system determines a generic variable should be (according to type signatures), there should be one identifier for a var that exists in the current scope of the correct type. Assuming the variable can not be inferred through type inference. In the following example we create a function called `instantiate_single` showing how to instantiate a single generic variable on a type. The generic variable `X` on `Single` is instantiated using `VAR` and assigned to the let variable `foo`. Thus the type of `foo` is `Single<VAR>`. This makes `VAR` the *instantiating variable* of `X`. Conversely, can also say `X` has been instantiated by `VAR`.

```
fn instantiate_single() {
    var VAR: usize = /* something */;

    let foo: Single<VAR> = Single::<VAR>;
}
```

Usage

To be useful, generic variables must be accessible to functions that use types with generic variables. When declared in the generic clause of an impl block, the identifiers of those generic variables will be in scope for all methods, functions, constant initialisers, and blocks, inside the containing impl block. The same is true for free functions, where generic variables will reside in the generic clause of the function, and be in scope for its signature and body. When used in this way, generic variables are a kind of implicit argument passing with additional type checking.

```
struct Foo<'a, var X: 'a usize>;

// although the generic variable is equivalent to passing the value as an argument
// in this case, it may still be used this way
fn mul_x2<var X: usize>() -> usize {
    X * 2
}

impl<var X: '_ usize> for Foo<X> {
    fn mul_x2_method(&self) -> usize {
        X * 2
    }
}

impl<var X: '_ usize> Trait for Foo<X> {
    // no reference to `self` is needed, because X is associated with the type,
not the value
    fn mul_x2_() -> usize {
```

```
            X * 2
        }
    }
```

When lowered to machine code, each generic variable will correspond to a pointer passed as an extra argument on functions using that use that generic variable, whether explicitly, or implicitly as with functions with generic types when those types have generic variables. For example, the method `mul_x2_method` would lower to something like this:

```
fn mul_x2_method(selph: &Foo, X: &usize) -> usize {
    X * 2
}
```

Dynamic dispatch

Generic variables are defined on types, types that may implement traits that may be dynamically dispatched. With how dynamic dispatch currently works, there is not prevision for a type having associated runtime information. It is simply assumed to be sufficient to know the static type.

There are two ways to go about solving this problem. Prevent dynamic dispatch from being used on types with generic variables, or make generic variables work with dynamic dispatch. Preventing dynamic dispatch from being used on type with generic variables would require a significant breakage of existing code and or hacks to implement. In general, this seems like a bad strategy. This was a major mistake I made last time I posted about this and many people pointed out to me that I couldn't make such major breaking changes to support a new feature. Instead types with generic variables would have to support dynamic dispatch.

This can be done by widening the dyn pointer to 3 pointers. The third one would point the instantiating variables of the value. This was a controversial idea when I suggested it last time, and I suspect it still will be, as it imposes a runtime cost on all code using dynamic dispatch not just code using generic variables. If people like the ideas I have presented so far and are interested in discussing the implementation details, I will create a separate post about the cost of widening dyn pointers, and it can be discussed there.

Footnotes

1. The exact placement of the generic variable part of a generic clause could be anywhere and it would be backwards compatible, although before or after constant generics seems like the most logical place.