

Archetype: Domain specific language for the representation of Abstract Algebra

Introduction

We seek to make a language which can be used to represent and manipulate algebraic structures, which we call Archetype. The language is designed to be used by mathematicians, and so the syntax is designed to be similar to mathematical notation while being concise and easy to learn.

Syntax

Statements

- The language is case sensitive.
- All statements end with a semicolon.
- Examples:

```
let a: u32 ; // declaration
a = 1; // assignment
a = func(2); // function call
```

Types of statements:

- Declaration: Must begin with the `let` keyword. The type of the variable must be specified after the variable name, with colon as the separator.

```
let a: u32;
```

- Assignment: Assigning a value (expression) to a variable requires no keyword.

```
a = 1;
```

- Initialisation: Does both declaration and assignment, requires a `let` keyword.

```
let a: u32 = 1;
let b: Vector<str> = ["a", "b", "c"];
let c: u32 = a + g(2, 3);
```

- Function call:

```
print(2);
```

- Return: Can only be used within a function.

```
return 2;
```

An assignment or initialisation statement can also include function calls, as in `let a: u32 = func(2);`.

Comments

- Single line comments begin with `//`.
- Multi-line comments begin with `/*` and end with `*/`.

- Comments cannot be nested.

Operators

- Relational (<, <=, >, >=, ==, !=): For the integer types, `BigRational`, and `float` only.
- Logical (&&, ||, !): For booleans only.
- Arithmetic (+, *, -, /): Can be overridden through Archetypes, and so must follow the rules.
 - For integers and `float`, the operators are defined as usual.
 - % is the modulo operator, only for integer types.
 - (+=, *=, -=, /=, %=): When relevant
- The dot (.) operator: For accessing struct fields.
- The slice operator (..): For creating slices of buffers or strings (similar to : in Python).
- The @ operator: This operator is used to compute the inner product of two vectors.

All the operators have the same meaning as in C, with enhanced functionality for non-C types (matrices, for example).

Conditionals

- The keywords `if` and `else` are used as in standard languages. Not all types may be compared using relational operators. However, they may appear as part of the predicate.
- The body of statements is enclosed in curly braces.
- The syntax:

```
let a: u32, b: u32, c: u32;
let max: u32;
if (a > b && a > c) {
    max = a;
}
else if (b > c) {
    max = b;
}
else {
    max = c;
}
```

Loops

- There are two `for` loops:
 - Similar to C, with 3 parts.


```
for (declaration; predicate; operation) {
    ...
}
```
 - Similar to Python, but only to iterate over the members of a `Buf`:


```
let list: Buf<u32> = [1, 2, 3];
for member in list {
    ...
}
```
- The `while` keyword can be used with a predicate as usual.


```
while (predicate) {
    ...
}
```

Functions

- Function prototypes begin with the **fn** keyword, followed by the function name, the arguments within parentheses, and then the return type.

```
fn foo(a: u32, b: Buf<float>): u32 {  
    return a;  
}
```

- Functions calls are identical to C: `foo(bar, baz)`. Functions can be returned from using the **return** keyword, which is again identical to C.

Builtins

‘Functions’ like `print` are offered directly by the language, much like in Python.

```
print("Hello world\n");
```

Forges

Functions provided by the language to convert between types (similar to Python’s `int('123')` and `str(123)`)

- Forges such as `float()` and `u32()` are used to cast between types.

```
let a: u32 = 1;  
let b: float = float(a);
```

- Forges like `Matrix(Buf<Buf<T>>)` are used for more complex type conversion.

```
let b: Matrix<u32> = matrix([[1, 2, 3], [4, 5, 6]]); // 2x3 matrix
```

- Forges accept multiple kinds of arguments.

```
let a: u8 = 1;  
let b: str = "123";  
let c1: u32 = u32(a);  
let c2: u32 = u32(b);
```

Type system

We have devised a rich and flexible type system to aid in expressing complex algebraic concepts. They work with the diverse Forges to allow the programmer to express their ideas in a concise and elegant manner.

Structs

Definition:

```
struct Foo {  
    field1: str,  
    field2: u32,  
}
```

Note that the trailing comma is optional.

To access a field, use the `.` operator.

```
let u: Foo; // declaration  
u.field1 = 1; // assignment
```

The same operator can be used to access fields, even from references to structs.

```
let u: Foo;  
let v: &name = &u;  
v.field1 = 1;
```

Enums

```
enum Bar {  
    Variant1,  
    Variant2,  
}
```

Note that the trailing comma is optional.

Use the `::` operator to depict enum variants.

```
let u: Bar = Bar::Variant1;
```

Archetypes

Archetypes are a powerful tool to allow the programmer to **claim** that their type satisfies the requirements for some algebraic structure. They are similar to traits in Rust. Unlike Rust, Archetype has exactly 4 Archetypes (**Group**, **Ring**, **Field**, **Space**).

claiming Archetypes

Each Archetype has a set of methods that must be implemented. These methods are discussed in the below sections for each Archetype.

To claim that a type satisfies an Archetype, one uses the **claim** keyword. This is similar to Rust's `impl Trait` syntax.

For example, to claim that a type **Foo** satisfies the **Group** Archetype, one uses

```
struct Foo {  
    z1: bool,  
    z2: bool,  
}  
  
// Claim that Foo is a Group (Z2 x Z2)  
claim Foo is Group {  
    fn add(a: Foo, b: Foo): Foo {  
        let foo: Foo;  
        foo.z1 = a.z1 != b.z1;  
        foo.z2 = a.z2 != b.z2;  
    }  
  
    fn zero(): Foo {  
        let foo: Foo;  
        foo.z1 = false;  
        foo.z2 = false;  
    }  
  
    fn neg(a: Foo): Foo {  
        let foo: Foo;  
        foo.z1 = a.z1;  
        foo.z2 = a.z2;  
    }  
};
```

While in the above example **Foo** is a **struct**, **claim** can also accept **enums**. Archetypes cannot be implemented for system types, but some default implementations are provided.

Group

A group is defined as a set S and an operation $f(a, b) = a + b$ which satisfies the following bounds:

- Closure: $\forall a, b \in S, a + b \in S$.
- Associativity: $a + (b + c) = (a + b) + c$
- identity:
 - $\exists 0 \in S$ such that $a + 0 = 0 + a = a$ for all $a \in S$
- inverse:
 - $\forall a \in S$ there exists $(-a) \in S$ such that $a + (-a) = 0$

A **Group** may be **claimed** in our language (see below) by specifying an operation which satisfies these bounds, as well as an identity element and the inverse operation.

Some examples of **Groups** are:

Group	Provided Type	Description
Z_n	<code>Cyclic<n: u32></code>	Cyclic group
S_n	<code>Symmetric<n></code>	Symmetric group
A_n	<code>Alternating<n></code>	Alternating group
D_{2n}	<code>Dihedral<n></code>	Dihedral group
$GL_n[F]$	<code>InvMat<n, F: claims Field></code>	Invertible $n \times n$ matrices over a field F

Required methods to claim (Note that **Self** is the type that is **claiming** the **Group** Archetype)

```
fn add(a: Self, b: Self): Self;  
fn zero(): Self;  
fn neg(a: Self): Self;
```

The **add** method overrides the $+$ operator, and **neg** overrides the unary $-$ operator.

Ring

A ring is an abelian group with operation $+$ with another operation, $*$. Using the same notation as before, the additional properties of a ring are:

- Closure over $*$: $\forall a, b \in S, a * b \in S$
- Associativity over $*$: $a * (b * c) = (a * b) * c$
- Distributivity of $*$ over $+$:
 - $a * (b + c) = a * b + a * c$
 - $(b + c) * a = b * a + c * a$
- Identity over $*$: $\exists e \in S | a * e = a \forall a \in S$

Some examples of **Rings** are:

Ring	Provided Type	Description
Z_p	<code>Cyclic<p: u32></code>	Integers mod p , p is prime
Z	<code>BigInt</code>	Integers
$M_n[F]$	<code>Matrix<n, F></code>	$n \times n$ matrices over a field F
$F[x]$	<code>Polynomial<F></code>	Polynomials over a field F

Note that the System Types `u8`, `u16`, `u32`, `u64` also **claim** the **Ring** Archetype.

Required methods to claim To **claim** the **Ring** Archetype, a type must first **claim** the **Group** Archetype. Then, the following methods must be implemented:

```
fn one(): Self;
fn mul(a: Self, b: Self): Self;
fn inv(a: Self): Self;
```

The `mul` method overrides the `*` operator, and `/` is overridden using the `inv` method.

INCOMPLETE FROM HERE

Field

A field is a commutative ring with the additional property that every non-zero element has an inverse in the second operation. Using prior notation, $\forall a \in S, a \neq i \Rightarrow \exists a^{-1} \in S | a \cdot a^{-1} = e$.

Members:

- reals
- complex numbers
- BigRational
- Non-Singular matrix (multiple Archetypes)
- Polynomials over a field (multiple multiple Archetypes)

Some examples of Rings are:

Ring	Provided Type	Description
Z_p	<code>Cyclic<p: u32></code>	Integers mod p , p is prime

Reals The `real` type represents an infinite precision floating point number, i.e. a real number.

Complex numbers The `complex` type represents a complex number, and unlike some implementations, is not generic. The syntax is as follows:

```
let a: Complex = complex(1, 2); // 1 + 2i
```

where both arguments are assumed to be reals.

Polynomials The `Polynomial` type is generic over any type that claims `Field` or `Ring`. The syntax is as follows:

```
let a: Polynomial<u32> = polynomial([1, 2, 3]); // 1 + 2x + 3x^2
```

Space

The only member is the `Vec` - for vector. It is generic over types that claim `Field` and `Ring`. In literature, a ‘vector space’ over a ring is known as a module, but we implement that functionality within `Vec` itself.

- Similar to vectors in C++.
- They provide basic array functionalities such as indexing, appending, etc., but also algebraic vector operations such as adding two arrays together, and scalar multiplication.
- The underlying type need not have commutative multiplication. For example, a `Vec` of `Matrixes` (which claim `Ring`) is a valid type, and the multiplication operation is defined as matrix multiplication.

```
let a: Vec<u64> = [1, 2, 3]; // initialisation
let b: Vec<u64> = a * 2; // Scalar Multiplication
let c = a + b; // Vector Addition
let c: u64 = a[0]; // Indexing
```

However, the following code is invalid.

```
let a: Vec<u64> = [1, 2, 3];
let b = 0.5 * a; // Scalar multiplication not closed for reals and integers
```

Corrected, the code becomes

```
let a: Vec<real> = [1, 2, 3];
let b = 0.5 * a; // Works
```

In general the type of the scalar is checked for compatibility with the type of the vector before multiplication.

Inner products

This is automatically implemented.

```
...
let a: Vec<u64> = [1, 2, 3];
let b: Vec<u64> = [4, 5, 6];
let c: u64 = a @ b; // Inner product
...
```

If the programmer wishes to claim the ``Space`` Archetype, they must implement the inner product operation `t`.

Cartesian Products

The cartesian product of two Archetypes is also an Archetype. This fact is used to implement tuples, with the syntax for the cartesian product of two Archetypes being `(Archetype, Archetype)`.

```
let a: (u32, u32) = (1, 2);
let b: u32 = a.0;
```

System type

- These are the data types/objects offered by the system, and while they may be represented using algebraic constructs (aka Archetypes), those structures are relatively more complex and esoteric. Naturally, the programmer may use these types to build more complex structures.
- Wrapping a `System` type within a `struct` allows the programmer to claim an Archetype for these types, and thus use them in algebraic operations.

Pointers

Pointers are used to refer to objects in memory, in a very similar fashion to C and C++. The syntax is as follows:

```
let a: u32 = 1;
let b: &u32 = &a; // b is a pointer to a
let c: u32 = *b; // c is the value pointed to by b
```

Boolean

Booleans are implemented as `System` types even though they technically satisfy the definition of a group. This is because they are used in the control flow of the program, and thus are not used in algebraic operations. The associated keywords are `true` and `false`.

```
let a: bool;
a = true;
a = !false;
```

Logical (`&&`, `||`, `!`) operators work on booleans as expected.

Buffers

Buffers are used to store data in memory. They are similar to arrays in C, and do not allow scalar multiplication or element-wise addition. The syntax is as follows:

```
let a: Buf<u32> = [1, 2, 3];
let b: u32 = a[0];
```

Buffers can have views (aka slices) using the `..` operator. The syntax is as follows:

```
let a: Buf<u32> = [1, 2, 3, 4, 5];
let b: Buf<u32> = a[0..2];
print(b) // [1, 2]
```

Strings

A `str` is equivalent to a buffer over `u8` (bytes), and enclosed with double quotes. The syntax is as follows:

```
let a: str = "Hello, World!";
let b: u8 = a[0];
```

They can be interconverted using

Strings can have views (aka slices) using the `..` operator. The syntax is as follows:

```
let a: str = "Hello, World!";
let b: str = a[0..5];
print(b); // Hello
```

There are no tuples for `System` types. For grouping, use `structs` and claim an Archetype.

Morphisms TODO

- Homomorphisms/Isomorphisms are functions which map between algebraic structures. They are defined using the `morph` keyword, followed by the function name, the arguments within parentheses, and then the return type.

```
ring morph foo(a: A) -> B {
  let b: B = /* some operation on a */;
  return b;
}
group morph foo(a: A) == B {
  let b: B = /* some operation on a */;
  return b;
}
```

Tokens

Reserved words

- `let`
- `if`
- `else`
- `for`
- `while`
- `fn`
- `claim`
- `is`
- `struct`
- `enum`

- `true`
- `false`
- `return`

Builtins

- `print`

Data types and their Forges

Type	Forge
Real	<code>real()</code>
u8	<code>u8()</code>
u16	<code>u16()</code>
u32	<code>u32()</code>
u64	<code>u64()</code>
BigInt	<code>int()</code>
Matrix	<code>matrix()</code>
BigRational	<code>rational()</code>
Vec	<code>vec()</code>
Buf	<code>buf()</code>
Str	<code>str()</code>
Complex	<code>complex()</code>
Polynomial	<code>polynomial()</code>
Permutation	<code>permute()</code>

Operators

- `@`
- `..`
- `::`
- `*`
- `+`
- `-`
- `/`
- `==`
- `!=`
- `>`
- `<`
- `&&`
- `||`
- `!`
- `;`
- `,`
- `:`
- `=`
- `.`

Special characters

- `(,)`
- `[,]`
- `{, }`

Comment characters

- `/*, */`
- `//`