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.

```
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 as in

  ```
  let a: u32;
  ```

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

```
a = 1;
```

- Builtin call: Statements like `print(2);`.
- Initialisation: A statement like `let a: u32 = 1;`, which does both declaration and assignment, requires a `let` keyword.
- Return: `return a;`, which can only be used within a function.

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: `(>, <, ==)` These operators are imlemented for the integer types, the rational type BigRational, and reals only.
- Logical: `(&&, ||, !)` These operators are implemented for booleans(i.e. predicates) only.
- Arithmetic: `(+, *, -, /)` These operators are implemented through archetypes, and thus can be used as the Archetype specifies. The exception is for integer types, where division is integer division.
- The dot `(.)` operator: This operator is used to access fields of structs, and to access other Archetype/System operations as 'methods'.
- The view operator `(..)`: This operator is used to create views (aka slices) of buffers and strings.
- 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:

```
if (pred) {
    body
}
else if (pred) {
    body
}
else {
    body
}
```

Loops

- The `for` keyword is used to iterate over a range of values. The syntax is similar to C.

```
for (declaration; predicate; operation) {
    .
    .
    .
}
```

- Using the `for` and `in` keywords, we can iterate over the members of a `vec` (see `Space`).

```
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 <name>  (arg: type, ...) : <return type> {
    <body>
}
```

Functions calls are identical to C: `name(args)`. 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(2);
```

prints 2. - Builtins such as `real()` and `u32()` are used to convert between types. They are used as follows:

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

- Builtins like `permute` and `matrix` are used for more complex type conversion. They are used as follows:

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

Type system

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

Structs

Definition:

```
struct name {
    members
}
```

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

```
let u: name; // declaration
u.field = 1; // assignment
```

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

```
let u: name; // declaration
let v: &name = &u; // pointer to u
v.field = 1; // assignment
```

Enums

```
enum name {
    .
    .
    .
}
```

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

```
let u: name; // declaration
u = name::variant; // assignment
```

Archetypes

Each type, except for the System types, is assigned one or more of the {four}(five if we adding Collection) Archetypes, which are as follows

Group

A group may be defined as an amalgamation of a set and an operation. The operation must satisfy certain bounds. With a set S and an operation $f(a, b) = a.b$:

- Closure: $\forall a, b \in S, a.b \in S$.
- Associativity: $ a.(b.c) = (a.b).c$
- Existence of
  - identity $\exists i \in S | a.i = a \forall a \in S$
  - inverse $\forall a \in S \exists a^{-1} \in S | a.a^{-1} = i$

Thus, a Group may be `claim`'d in our language (see below) by specifying an operation which satisfies these bounds, as well as an identity element and the inverse operation.

Abelian Group A group is abelian if it's operator is commutative, i.e., $a.b = b.a \forall a, b \in S$.

Permutation groups The groups S_n represent the permutations of n objects. They are generic over any type that claims an Archetype, as all the other Archetypes are subtypes of the Group Archetype. The syntax is as follows:

```
let a: Permutation<u32> = permute([1, 2, 3]);
let b: Permutation<real> = permute([1.1, 2.2, 3.3]);
```

Polygon symmetries The groups D_{2n} represent the symmetries of an n-gon with reflections.

Ring

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

- Closure: $\forall a, b \in S, a * b \in S$
- Associativity: $a * (b * c) = (a * b) * c$
- Distributivity: $a * (b.c) = (a * b).(a * c)$
- Existence of identity: $\exists e \in S | a * e = a \forall a \in S$

Members:

- Unsigned integers `u8`, `u16`, `u32`, and `u64` : Our language does not treat integers and reals as primitive data types.
- `BigInt`: Unbounded integer type, similar to integers in Python.
- `Matrix`
  - A matrix is treated as a generic over any type, but the methods it provides wil depend on the Archetype of that type. For example, a matrix of integers will not have the inverse operation, because the inverse of a matrix of integers may contain reals.

- `Polynomial`
  - Similar to matrices, they can be generic over any type.

Commutative rings

- When the $*$ operation is commutative, the ring is said to be commutative.

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, $\forall a \in S, a \neq i \Rightarrow \exists a^{-1} \in S | a.a^{-1} = e$.

Members:

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

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

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 `Matrix`es (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 themselves.

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 eve 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];
```

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 `struct`s and claim an Archetype.

The claim keyword

To create a new instance of an Archetype, the programmer may use the claim keyword. The syntax is as follows:

```
claim (name is Archetype) {
    implement operations
};
```

And one can check whether a type has claimed an Archetype using

```
if (name is Archetype) {
    .
    .
    .
}
```

where name is the name of a type that has already been declared (struct). The new type may directly implement the operations, or define mappings (morphisms) to some other type which implements the operations, like so:

```
morph (self to other) {
    Function accepting self and returning other
};
```

Morphisms TODO: Define morphisms.

Note, this is not inheritance, as the programmer cannot write `claim cat is animal {...}`.

Tokens

Reserved words

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

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

Builtins

- `permute`
- `matrix`
- `print`

Data types

- `real`

- `u8`
- `u16`
- `u32`
- `u64`
- `BigInt`
- `BigRational`
- `Vec`
- `Buf`
- `str`

Operators

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

Special characters

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

Comment characters

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