Archetype: Domain specific language for the representation of Abstract Algebra structures - Compilers Group 2

Abhay Shankar K: cs21btech11001 Kartheek Tammana: cs21btech11028
Prasham Walvekar: cs21btech11047 Sumedh Kashikar: es21btech11033

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

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

Comments

- Single line comments begin with `//`.
- Multi-line comments begin with `/*` and end with `*/`.
- Comments cannot be nested.

Statement types

- 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;
let b: str, c: bool, d: float;
```

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

```
a = 1;
foo.bar = 2; // Accessing a field of a struct
```

- 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);
foo(bar, baz);
```

- Return: Can only be used within a function.

```
return 2;
```

Operators

- Binary operators:
  - Relational `(<, <=, >, >=, ==, !=)`: For the integer types, `Rational`, 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.
- Unary operators:
  - `!`: Logical negation
  - `-`: Arithmetic negation
  - `&`: Reference operator
  - `*`: Dereference operator
- 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;
}
```

- The `switch` case conditional is also provided, similar to C. The keyword `switch` is used followed by an expression in parantheses, whose value can be of integer, character, or enum variant type only (Similar to C).
- The case constants consist of the keyword `case` followed by the case constant (integer, character or enum variant type, and also the same type of the value returned by the switch case expression), followed by the arrow operator (=>). Each case block is enclosed in curly braces. The default case is optional.
- The syntax:

```
let a: u32 = 5001;
switch(a%2) {
    case 0 => { b = Parity::Even;}
    case 1 => { b = Parity::Odd; }
  }
```

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: [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.
- Functions calls are identical to C. Functions can be returned from using the `return` keyword, which is again identical to C.

  ```
  fn foo(a: u32, b: [float]): u32 {
    return a;
  }
  let b: u32 = foo(1, [1.0, 2.0, 3.0]);
  ```

  Builtins

  - Functions like `print` are offered directly by the language, much like in Python. They are called in the same way as user-defined functions.

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

Forges

- Functions provided by the language to convert between types (similar to Python's `int('123')` and `str(123)`)
- Forges are Archetypes' equivalent to constructors. They are defined using the `forge` keyword, similar to functions, and casting can be done using the `as` keyword.
- The syntax is:

  ```
  enum Parity {
    Even,
    Odd
  }

  forge (a: u8) as (b: Parity) {
    if (a % 2 == 0) {
      b = Parity::Even;
    } else {
      b = Parity::Odd;
    }
  }

  forge (a: (u8, u8)) as (c: Parity) {
    if (a.0 % 2 == 0 && a.1 % 2 == 0) {
      c = Parity::Even;
    } else {
      c = Parity::Odd;
  ```

```
  }
}

fn main() : i32 {
  let a: Parity = 1 as (Parity);
    // a is now Parity::Odd
  let x: (u8, u8) = (1,2);
  let b: Parity = x as (Parity);
    // b is now Parity::Odd
}
```

- Forges like `(a: [[u32]]) as (b: Matrix)` are used for more complex type conversion.

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

- Definition:

```
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 operations that must be implemented. These operations 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:

  ```
  struct Foo {
    z1: bool,
    z2: bool,
  }

  // Claim that Foo is a Group (Z2 x Z2)
  claim Foo is Group {
    (foo = a + b) => {
      foo.z1 = a.z1 != b.z1;
      foo.z2 = a.z2 != b.z2;
    }

    (foo = 0) => {
      foo.z1 = false;
      foo.z2 = false;
    }

    (foo = -a) => {
      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.
- An alternate way to claim an archetype is through an isomorphism. If there is an existing type that implements the Archetype, and there is an isomorphism between the two types, then the Archetype can be claimed using the `claim ... with ...` syntax:

  ```
  struct Foo { ... }
  struct Bar { ... }

  claim Bar is Group { ... }

  fn foo_to_bar(foo: Foo): Bar { ... }
  fn bar_to_foo(bar: Bar): Foo { ... }

  claim Foo is Group with (foo_to_bar, bar_to_foo);
  ```

- Both `foo_to_bar` and `bar_to_foo` must be isomorphisms, and should be inverses of each other.

Group

- A group is defined as a set S and an operation $+$ 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 `claim`ed 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 `Group`s are:

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

- Note that `claims` is not a keyword. It is simply used within this document to indicate that the type argument `F` must be a `Field`.

To `claim`

```
(c = a + b) => {
  ...
}

(c = 0) => {
  ...
}

(c = -a) => {
  ...
}
```

Ring

- A ring is a group with operation $+$ with another operation, $*$. Using the same notation as before, the additional properties of a ring are:
  - Abelian (commutative) group over $+$: $a + b = b + a$
  - 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 `Ring`s are:

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

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

To `claim`

- To `claim` the `Ring` Archetype, a type must first `claim` the `Group` Archetype. Then, the following operations must be implemented:

  ```
  (c = a * b) => {
    ...
  }

  (c = 1) => {
    ...
  }
  ```

Field

- A field is a ring with the following additional properties:
  - The operation $*$ is commutative: $a * b = b * a$
  - Multiplicative inverse: $\forall a \in S, a \neq 0 \Rightarrow \exists a^{-1} \in S | a * a^{-1} = 1$
- Some examples of `Field`s are:

Field	Provided Type	Description
Z_p	`Cyclic<p: u32>`	Integers mod p, p is prime
Q	`Rational`	Rational numbers
C	`Complex`	Complex numbers

- Note that the `Complex` type is over `Rational`s, and not Reals. Archetype does not provide a `Real` type, as it is not possible to represent a real number in a computer.

To `claim`

- To `claim` the `Field` Archetype, a type must first `claim` the `Ring` Archetype. Then, the following operations must be implemented:

  ```
  (c = 1 / a) => {
    ...
  }
  ```

Space

- Refer to the Wikipedia article for a formal definition.
- The only provided member is the `Vec<F: claims Field>`. It is generic over types that claim `Field`.
  - Similar to vectors in other languages
  - It provides basic array functionalities such as indexing, appending, etc., but also algebraic vector operations - adding two arrays together, and scalar multiplication.

  ```
  let a: Vec<u64> = [1, 2, 3] as (Vec<u64>);
  let b: Vec<u64> = a * 2;
  let c = a + b;
  let d: u64 = a[0];
  ```

  ```
  let a: Vec<Rational> = ([(1, 2) as (Rational), (1, 2) as (Rational), (1, 2) as (Rational)]) as (Vec<Ra
  let b = 0.5 as (Rational) * a; // Works
  ```

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

To claim

```
claim Foo is Space {
  Field = (insert field F here);

  // Here u and v are Foo, a is Field
  // 0 is the additive identity of Foo, not Field

  (w = u + v) => {}

  (w = -u) => {}

  (w = 0) => {}

  (w = a * u) => {}

}
```

Inner products

- This is automatically implemented for `Vec<F: claims Field>` as the `@` operator, using the dot product.

  ```
  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 Archetypes for tuples, with the syntax for the cartesian product being `(Archetype, Archetype)`.

  ```
  let a: (u32, u32) = (1, 2);
  let b: (u32, u32) = (3, 4);
  let c: (u32, u32) = a + b; // + is automatically implemented because (u32, u32) is a Cartesian Produc
  ```

System type

- These are the data types 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.

Integers

- The integer types are:
  - `u8`, `u16`, `u32`, `u64` - unsigned 8-bit integer, etc.
  - `i8`, `i16`, `i32`, `i64` - signed 8-bit integer, etc.

References

- These work very similarly to C++:

  ```
  let a: u32 = 1;
  let b: &u32 = &a; // b is a refernce to a
  let c: u32 = *b; // c is the value pointed to by b (i.e. a)
  ```

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: (Note that the array initialization syntax returns a Buf type, and not a `Vec`.)

  ```
  let a: [u32] = [1, 2, 3]; // [u32] denotes that it is a 1-D buffer of type u32
  let b: u32 = a[0];
  ```

- Buffers can be sliced using the `..` operator (similar to `:` in Python). The syntax is as follows:

  ```
  let a: [u32] = [1, 2, 3, 4, 5];
  let b: [u32] = a[0..2]; // b is [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 `structs` and claim an Archetype.

Sample Code

- Note that Archetype code uses the `.arc` extension.

  ```
  enum Bar {
      Zero,
      One,
      Two
  }

  let Z0: Cyclic<3>;
  let Z1: Cyclic<3>;
  let Z2: Cyclic<3>;

  forge (a: Cyclic<3>) as (b: Bar) {
      let a: u8 = a as (u8);
      if(a == 0) {
          b = Bar::Zero;
      }
      if(a == 1) {
          b = Bar::One;
  ```

```
    }
    else {
        b = Bar::Two;
    }
}

forge (a: Bar) as (b: Cyclic<3>) {
    if(a == Bar::Zero) {
        b = Z0;
    }
    else if(a == Bar::One) {
        b = Z1;
    }
    else {
        b = Z2;
    }
}

claim Bar is Group {
    (c = x + y) => {
        c = (x as (Cyclic<3>) + y as (Cyclic<3>)) as (Bar);
    }
    (c = 0) => {
        c = Bar::Zero;
    }

    (c = -x) => {
        c = (- (x as (Cyclic<3>))) as (Bar);
    }
}

struct Foo {
    a: u8,
    var: Bar
}

forge (a: u8) as (out: Foo) {
    out.a = a;
    out.var = Bar::Zero;
}

forge (a: u8, var: Bar) as (out: Foo) {
    out.a = a;
    out.var = var;
}

claim Foo is Group {
    (out = x + y) => {
        out.a = x.a + y.a;
        out.var = x.var + y.var;
    }

    (c = 0) => {
        c = (0, Bar::Zero) as (Foo);
    }
}
```

```
fn main() {
    print("Hello world.\n");
    let q: Foo = 0 as (Foo);
    let qq: Foo = 1 as (Foo);

    let qqq: Foo = q + qq;

    print(qqq as (str));
}
```