Refu Documentation

Release 0.5

Lefteris Karapetsas

Nov 30, 2017

Contents

Refu is a hybrid language with a strong type system based on algebraic data types. It is designed as a general use programming language with focus on system programming.

It is statically typed, designed for programming to the interface and has a powerful module system among other features.

CHAPTER 1

Useful Links

- Source Code
- Gitter Chat

CHAPTER 2

Language Documentation

In the following pages of the documentation we will see an introductory tutorial on Refu, presenting all the features of the language in order and how to use them effectively.

Furthermore if you want to dive into it is the official specification of the language check out the *Language Reference*.

CHAPTER 3

Introduction to Refulang

3.1 Getting Started

TODO

CHAPTER 4

Language Reference

4.1 Refu Language Reference Manual

4.1.1 Built-in data types

The following data types are built-in. Some of them correspond to the data types defined by the C99 standard in `<stdint.h>` but they follow the same naming scheme as in rust.

- **Unsigned numbers**
  - `uint`: compiler will decide the size depending on context
  - `u8`: 8 bit unsigned integer, corresponding to `uint8_t`
  - `u16`: 16 bit unsigned integer, corresponding to `uint16_t`
  - `u32`: 32 bit unsigned integer, corresponding to `uint32_t`
  - `u64`: 64 bit unsigned integer, corresponding to `uint64_t`
- **Signed numbers**
  - `int`: compiler will decide the size depending on context
  - `i8`: 8 bit signed integer, corresponding to `int8_t`
  - `i16`: 16 bit signed integer, corresponding to `int16_t`
  - `i32`: 32 bit signed integer, corresponding to `int32_t`
  - `i64`: 64 bit signed integer, corresponding to `int64_t`
- **Real numbers**
  - `f32`: corresponds to binary32, single precision floating point, as defined by IEE 754-2008
  - `f64`: corresponds to binary64, double precision floating point, as defined by IEE 754-2008
- **Strings**

  - `string`: UTF-8 encoded unicode string.
  - `string8`: Ascii encoded string
- **Other**
  - `bool`: A boolean true or false value
  - `nil`: the unit type, also known as NULL

Note: `string8` type is currently not implemented.

4.1.2 Alebraic Data Types

More complex data types can be defined by the user as Algrebraic data types. This is achieved with the `type` keyword.

```
type person {
    name:string, age:int |
    name:string, age:int, surname:string
}

type list {
     nil | (load:int, tail:list)
}

type foo {
    a:int,
    b:(string|float)
}

type foo {
    a:int,
    b:(string | (i:int, f:f32))
}
```

Above we have the definition of a person and a list. A person has a name and an age and optionally a surname. And a list is either empty (denoted by the `nil` keyword) or it has a load of an `int` and a tail which is another list.

Recursive Data Types

TODO: Describe recursive ADTs

Anonymous Data Types

An algebraic data type can be considered as the equivalent of a tagged union type in C. Refu also supports anonymous ADTs. That means, you can encounter the ADT syntax without it having been defined. For example, a function's argument can be an anonymous ADT.

```
fn print_me(a:(string | b:int, c:int))
{
    //do some initialization stuff
    ...
    //and now do the pattern matching
    match a {
```

```
        string   => print(a)
        int, int => print(a.b, a.c)
    }
}
```

```
fn print_me(a:string | (b:int, c:int)) -> int
_    => {
    print("%s", a)
    print("one argument")
}
_, _ => {
    print("%d %d", b, c)
    print("two arguments")
}
```

Above we have one function with an anomymous ADT. If such a function exists then it must have a match expression somewhere inside its body in order to distinguish what kind of input it is having before this input is used. The most explicit way to achieve this is to write the match expression explicitly as in example 1. To do that we match the keyword `fn` inside the function's body against the various cases.

In another case if the function body consists only of different branches depending on the input we can omit the function's body block completely and go with the way that example 2 is defined, which resembles a lot the way functions are defined in haskell. It is just syntactic sugar for achieving the same thing as in example 1.

Instantiating objects

In order to construct an instance of a data type you have to use one of its constructors. A constructor of an object is simply defined as any of its sum type operands.

```
a:person = person("steven", 23)
b:person = person("celina", 22, "wojtowicz")
```

For ease of use, arguments can also be given to a constructor as keyword arguments. If one keyword argument is passed to a constructor then all arguments should be keyword arguments. Finally when passing keyword arguments the order of the arguments does not matter as opposed to when calling a constructor normally.

```
a:person = person(name="steven", age=23)
b:person = person(name="celina", surname="wojtowicz", age=23)
```

Note: Keyword arguments are currently not implemented

Instantiating Recursive data types

Note: Recursive data types are currently not implemented

Data types can also be recursive. This is how we can define collections in Refu. But how do you construct a collection?

```
a:list = nil
b:list = list(1, 2, 3, 4, 5)
c:list = list(1, list( 2, list(3, list(4, list(5, nil)))))
```

In the above examples list `b` and list `c` are equal. The canonical way to define a list would be exactly like list `c` is defined, having /1/ as its first element and using `nil` after `5` to denote the list's end.

As we can see above to construct a recursive data type we still use a constructor but we can take advantage of the fact that the type is recursive in order to construct it.

In the case of `b` 's construction Refu knows that a list's constructor can only accept an int and a next list pointer. Using that knowledge it can expand the `list(1, 2, 3, 4, 5)` to `list(1, list(2, list(3, list(4, list(5, nil)))))`.

Same thing can work for more complex recursive data types such as a binary tree. Look below for an example.

```
type binary_tree {
    nil | load:int, left:binary_tree, right:binary_tree
}

a:binary_tree = nil
b:binary_tree = binary_tree(8, (4, (1, 7)), (12, (10, 19)))
c:binary_tree = binary_tree(
    8,
    binary_tree(4,
                    binary_tree(1, nil, nil), binary_tree(7, nil, nil)),
    binary_tree(12,
                    binary_tree(10, nil, nil), binary_tree(19, nil, nil)))
```

4.1.3 Array Types

Array types are like simple C arrays that are aware of their own size so as to make sure there is no out of bounds access. An array is simply a contiguous block of memory containing values of the same type.

```
array_of_ints:int[20]
array_of_strings:string[20]
a:int = array_of_ints[5]
b = array_of_ints[5] // type deduction
c:int = array_of_ints[22] //compile error
```

Dynamic size arrays can also be instantiated with the built-in `make_arr(type, elements_number)` function. An array's size in elements can be queried by `array.size`.:

```
fn foo(b:u8[]) {
    b[3] = 16;
}

buffer:u8[] = make_arr(u8, 10)
foo(buffer)
printf("%d", buffer.size); // should print 10
printf("%d", buffer[3]); // should print 16
```

Note: Dynamic arrays and their instantiation is currently not implemented

4.1.4 Types and Conversion

All elementary types can be converted from and to another type. Type conversion can either be explicit or implicit.

Implicit Conversion

Implicit conversion happens when you simply assign a value of one type to a value of another type for which conversion is legal. It can also happen during almost all other parts of the code like in a function call, in a constructor e.t.c.

The implicit conversion rules for elementary types can be seen in the following table where:

- OK -> conversion is allowed
- NO -> conversion is not allowed
- WC -> conversion produces a warning for a value of type and error for a constant of type since then we are sure that data is going to be lost. Only works for assignments now.

from/to	i8	u8	i16	u16	i32	u32	i64	u64	f32	f64	string	bool	nil
i8	OK	WC	WC	OK	WC	OK	WC	OK	OK	OK	NO	OK	NO
u8	OK	OK	OK	OK	OK	OK	OK	OK	OK	OK	NO	OK	NO
i16	WC	WC	OK	WC	OK	WC	OK	WC	OK	OK	NO	OK	NO
u16	WC	WC	OK	OK	OK	OK	OK	OK	OK	OK	NO	OK	NO
i32	WC	WC	WC	WC	OK	WC	OK	WC	OK	OK	NO	OK	NO
u32	WC	WC	WC	WC	OK	OK	OK	OK	OK	OK	NO	OK	NO
i64	WC	WC	WC	WC	WC	WC	OK	WC	OK	OK	NO	OK	NO
u64	WC	WC	WC	WC	WC	WC	OK	OK	OK	OK	NO	OK	NO
f32	NO	NO	NO	NO	NO	NO	NO	NO	OK	OK	NO	NO	NO
f64	NO	NO	NO	NO	NO	NO	NO	NO	OK	OK	NO	NO	NO
string	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO
bool	OK	OK	OK	OK	OK	OK	OK	OK	NO	NO	NO	OK	NO
nil	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO	NO

From the table we can understand that the general idea is that:

- All int types can be converted to each other except for:
  - signed to unsigned, which produces a warning and fails for constants
  - large to smaller, which produces a warning and fails for constants
- All int types can be converted to booleans
- All int types can be converted to floats
- Floats can be converted to each other
- Bool can be converted to integer types

There is one **major** exception to the above rules and that is pattern matching. In patern matching no implicit conversions are allowed, except for smaller sized integers of the same signednes towards bigger sized integers.

Below are some examples for assignments:

```
a:u8 = 128
b:u32 = a      // implicit conversion allowed
c:u8 = b       // implicit conversion will produce a warning. Larger to smaller.
d:u8 = 65535   // implicit conversion will fail. Constant is of larger type
f:i8 = 64
e:u8 = f       // implicit conversion will produce a warning. Signed to unsigned.
g:u8 = -64     // implicit conversion will fail. Signed constant to unsigned variable.
h:i64 = true   // implicit conversion allowed. h == 1
f:bool = 2245  // implicit conversion allowed. f == true
j:bool = 0     // implicit conversion allowed. j == false
```

As far as binary operators are concerned the result of an operation to elementary type is valid if one of the operands can be converted to each other. If that is the case the type of the operation is the type of the larger type.

```
a:u64 = 653432431
b:u16 = 2324
c:f32 = 3.14
d:string = "abc"
a + b // valid u16->u64, type will be u64
b + c // valid u16->f32, type will be f32
c + d // invalid
```

A type can be implicitly converted to a sum type by succesfull conversion to either of its sum operands. For example:

```
fn foo (a:u64 | b:string) { }

fn main() {
    foo(45)
    foo("eleos")
}
```

Only one implicit conversion is allowed per type comparison. Continuing from the above example we can't have:

```
fn foo (a:u64 | b:string) { }

fn main() {
    foo(true)
}
```

That's because this would require two different implicit conversions.

Explicit Conversion

Explicit conversions allow for quite a bit more freedom for converting between types. An explicit conversion is achieved with a function call to the name of the type. Much like a constructor of a user defined type, which itself could be thought of as a sort of a type conversion.

All integer types can be converted to each other with explicit conversions, except for constants that would obviously cause loss of data.

```
a:u64 = 542312
b:i16 = i16(a)     // valid explicit conversion, would give warning as implicit
c:u16 = u16(b)     // valid explicit conversion, would give warning as implicit
d:u8  = u8(-13)    // invalid explicit conversion, obvious loss of data
e:u8  = u8(65535) // invalid explicit conversion, obvious loss of data
```

Floats can be explicitly converted to ints.

```
a:f32 = 3.2313;
b:f64 = 123.231233;
c:u32 = u32(a);  // valid explicit conversion, would give warning as implicit
d:u16 = u16(b);  // valid explicit conversion, would give warning as implicit
```

An interesting case is explicit conversion to string. Explicit conversion to string is allowed but only for integer/floating constant and booleans.

```
a:u32 = 2321;
b:string = string(2313)    // valid conversion
```

```
c:string = string(a)       // invalid conversion, not constant
d:string = string(23.231) // valid conversion
e:bool = (3 == 4)
e:string = string(e)       // valid conversion
```

4.1.5 If Expressions

In Refu an `if` can act either as an expression or like a statement depending on the context. That means, that you can assign an if expression as values to variables. The general syntax is as follows:

```
if i > 10 {
    increase_a_value()
    compress_a_file()
} elif i < 0 {
    do_something_else()
} else {
    do_last_thing()
}
```

The above `if` acts as a statement since it is not in the right side of any kind of assignment. But observe below another example usage where `if` is used as an expression. Depending on the value of `if`, we assign a specific value to `a`.

```
a:int
a = if i > 10 {
    20
} elif i < 0 {
    40
} else {
    100
}
```

Unlike some other languages the curly braces can't be omitted in any branch of the if. If the condition of an if branch is complex enough then it should be enclosed in parentheses like below.:

```
if ((i > 10 && i <20) || (x > 30 && x < 40)) {
      do_something()
}
```

4.1.6 For Expressions

The simplest way to iterate something in refu is by using a `for` expression. The syntax is simple. For a simple iteration over a range of integers you can use the following.

```
for i in 0:10 {
      //this will iterate 10 times, with i ranging from 0 to 9
      do_something()
}
```

The simple iteration syntax is `for` *identifier* `in` *iterator*. Iterator can either be a *range* in the form of `start:step:end` or a collection. For ranges the step is optional and is shown in the next example. By default `step` is `1`. Step can also be negative.

```
for i in 0:2:10 {
     //this will print 0, 2, 4, 6, 8
```

```
     print(i)
}

for i in 10:-2:0 {
     //this will print 10, 8, 6, 4, 2
     print(i)
}
```

For expressions are also heavily customizeable on a per type basis. By deriving the standard library's iterator typeclass you can define how the expression behave for a specific type. For example:

```
type list {
    nil | payload:int, tail:list
}

instance std::iterator<list> {
    fn(self:list)->list
    {
        match(self) {
          (nil) => return nil
          (val, tail) => return (val, tail)
        }
    }
}

my_list:list = (1, 2, 3, 4, 5)
for i in my_list {
     //this should print all the values of the list.
     print(i)
}
```

Note: The iterator typeclass and the surrounding behaviour of defining your own iterator behaviour is not yet implemented.

By defining the `list_iter` instance of the iterator typeclass we just defined the way that lists can be iterated. Afterwards whenever a `for` expression is used on a list, the defined implementation is used. The iterator typeclass looks like this:

```
class iterator <type T> {
     fn(self:T) -> (nil | (Any, T))
}
```

So, all implementations need to do is define the value at each iteration, the next object of the iteration and the condition under which the iteration terminates. The function must return either `nil` to denote the end of the iteration, or a value of type `T` and the next object for iteration.

Finally, `for` expressions can also be assigned. For example an array can be assigned like this:

```
arr:int[3] = [5, 6, 7]
another_arr:int[] = for i in arr { i + 3 }
```

`another_arr` will contain `[8, 9, 10]`. Ofcourse these expressions are checked at compile time for validity of type assignment. If the for block had something that is not an int, or if it had more statements then it would be a compile error. On the left hand of the assignment any identifier whose type would agree with `(nil | int, T)` would be acceptable.

4.1.7 Pattern Matching

Algebraic data types go hand in hand with the ability to use pattern matching on those types in order to deconstruct them. This is offered by the `match` expression keyword in refu.

Pattern matching is the elimination construct for algebraic data types. That means that a pattern matching expression, expresses how one should consume a partciular ADT. For example look below.

```
type list {
    nil | load:int, tail:list
}

a:list
match a {
     nil    => print("empty list")
     i, _   => print("Head of the list is %d", i)
}
```

Match expressions can also be recursive. A `match()` inside a match expression renders the whole match recursive. For example look at the matching below which calculates the length of a list.

```
fn find_length(a:<list>) -> int
{
    return match a {
        nil      => 0
        _, tail  => 1 + match(tail)
    }
}
```

For completeness sake it should be noted that the above example can be written in a simpler way, having the function block omitted:

```
fn find_length(a:list) -> int
    nil      => 0
    _, tail  => 1 + find_length(tail)
```

In a `match`, all possible value combinations must be exhausted. _ means any value, `nil` means no value and anything else is interpreted as an identifier to recognize that particular positional argument. Another way to match something would be depending on the type. For example.

```
type list <T> {
    nil | (load:T, tail:list)
}

a:list<int> = list<int>(1, 2, 3)
list_type:string = match a {
   nil      => "empty list"
   int, _   => "list of ints"
   _        => "other kind of list"
}
```

From the above, one can notice the following. A match expression is just that, an expression and can as easily be assigned to something. Finally it is a compile error to not exhaust all possible matches, so the _ at the end matches all other cases.

```
type foo {
    a:i16 | b:u16 | c:string | d:bool
}
```

```
a:foo
match a {
    string | bool    => "not a number"
    a:(i16 | u16)    => 5 + a
}
```

Another way to define patterns is by using the type operators. As can be seen above one can combine possible type reductions using the same operators we use when a type is defined.

4.1.8 Memory Model

Note: As of the moment of writting the memory model is still unimplemented and there are ideas floating around which need to be solidified. Some of these ideas are presented here.

TODO

4.1.9 Functions

Functions are declared in Refu just like in the Rust language. The keyword `fn` followed by the name of the function, the arguments and finally by an arrow pointing to the return value.

Return value

As mentioned, whatever follows `->` is the function's return value. If there is no return value then the arrow is omitted. Some examples follow:

```
fn add_two_ints(a:int, b:int) -> int
{
     return a + b
}

fn print_something()
{
    print("something")
}
```

Inside the function's body a `return` statement denotes the expression that determines the return value. A function may return a value but still need no return statement if it's compact enough and has all its functionality under a `match`, `if` or `for` expression. For example:

```
fn int_inside_range(x:int, from:int, to:int) -> bool
{
    if (x >= from && x <= to) { true } else { false}
}
```

In the absense of a return value the function's last expression statement value is interpreted as the return value. For example the following function's return value is determined by `a + 1`.

```
fn do_something(a:int) -> int
{
    a = a * 2
```

```
    if (a > 10) {
       a - 5
    } else {
       a - 1
    }
    a + 1
}
```

Moreover a function can also completely omit a body block if it has a `match` expression on its arguments like below:

```
fn find_length(a:~list) -> int
    (nil) => 0
    (_, tail) => 1 + find_length(tail)
```

Argument Evaluation Strategy

The argument evaluation strategy is pass by value for all elementary types, except for `string`. That means that they are copied inside the function. Objects of all user defined types are passed by reference, which means that simply a pointer to the object is passed along in the function.

4.1.10 Type Parameters

Refu supports type parameters, which syntactically look like generics of other programming languages. Their use will be seen heavily in the use of typeclasses below but first let's take a look at the syntax.

```
type list <type T> {
     nil | payload:T , tail:list
}
..
..

a:list<int> = (5, 6, 7, 8)
```

This would define a generic ADT list, and later the user declares a list of ints and populates it. Same thing can be done with an ADT binary tree.

```
type binary_tree <type T> {
    nil | payload:T , left_branch:uptr<binary_tree>, right_branch:uptr<binary_tree>
}
...
...
/*
              1.0
              / \
          0.1    2.0
          /  \    / \
       0.01 0.2  1.5 3.3
*/

a:binary_tree<double> = ( 1.0, (0.1, (0.01), (0.2)),  (2.0, (1.5, 3.3)))
a:binary_tree<double> = (1.0, cons(0.1, cons(0.01, Nil), cons(0.2, Nil) ),  cons(2.0,␣
↪cons(1.5, Nil), cons(3.3, Nil)))
```

Type parameters can be of either a concrete type as designated by `type` or by a type of a type also known as a `kind`. We will read more about kinds in the corresponding section.

Note: Type Parameters are only implemented during the parsing stage of the compiler and are not yet ready.

4.1.11 Kinds

Note: Kinds are not implemented whatsoever and their usage in the language is not yet well thought out.

Kinds are essentially the types of types. In Kind syntax a concrete type is represented as $\star$. The syntax of kinds is similar to that of types. A $\star$ represents any concrete type.

To understand kinds picture them as the types of a type constructor (generic type). For example the `Maybe` type.

```
type maybe<type T> {
    T | nil
}
```

has a kind of $\star->\star$ which means that it takes one concrete type and derives another concrete type.

4.1.12 Typeclasses

Note: Typeclasses are only implemented in the parsing stage of the language at the moment. But they are high in the priority list so they should be ready soon.

Refu relies heavily on the use of typeclasses. They are an important way to guarantee behaviour about objects of a given type. There are quite a few builtin typeclasses in the standard library. The concept of a typeclass is similar to that of an interface in some other languages.

Simple Example

Here is one example which defines the operation of the adding operator. This allows an object to define how it shall be added. One can notice the keyword `self` which defines the object the function will be called for and also the generic syntax of `<type T>` since we can't know the type of the object we are adding.

```
class addition <type T> {
   fn add(self:T, other:T) -> T
}

type vector {
    x:int, y:int, z:int
}

//A type would declare that it derives the typeclass
instance addition<vector> {
   fn add(self:T, other:vector) -> vector
   {
       ret:vector
       ret.x = self.x + other.x
       ret.y = self.y + other.y
       ret.z = self.z + other.z
       return ret
```

```
    }
}
```

So what the above code declares is that there is some type called `vector`. That type is an instance of the addition typeclass with the given implementation. The addition typeclass like some other special typeclasses allow for special operations. In particular it allows for overloading operator +. So adding two vectors would in essence call the instance of the typeclass.

Note that `self` can be omitted from the arguments of a typeclass and its type instance if no special evaluation/ownership strategy needs to be used. It will always be implied.

Advanced example - Iteratable Collections

Another example of a typeclass would be a class of types that can be iterated. All collection types should be iterateable so the following typeclass definition makes sense.

```
class iteratable <collection c> {
      fn iterate(self:collection, cb:function)
}
```

And below we can see a nice example for how an instance of this typeclass would be implemented.

```
type array <type T> {
    T[]
}

instance iteratable <array, type T> {
    fn iterate(self:T, cb:function)
    {
        for i in range(0:self.length) {
            cb(self[i])
        }
    }
}
```

Typeclass Inheritance

The use of typeclasses is extended by the possibility of inheritance between typeclasses.

```
class equality <type T> {
    fn equals(self:T, other:T) -> bool
    fn nequals(self:T, other:T) -> bool
}

class comparison <type T> extends equality{
    fn greater_than(self:T, other:T) -> bool
    fn less_than(self:T, other:T) -> bool
}

class super_comparison <type T> extends comparison{
    fn gteq(self:T, other:T) -> bool
    fn lteq(self:T, other:T) -> bool
}

//multiple inheritance
```

```
class reader <type T> {
   fn read(a:T)
}

class writer <type T> {
   fn write(a:T)
}

class io <type T> extends (reader, writer) {
    // can be empty or can have additional functions to implement
}
```

The typeclass equality above allows for types that instantiate it to use its 2 equality functions, while the comparison typeclass on the other hand allows for greater and less than comparison in addition to the equality functions. Additionally multiple levels of inheritance can be valid as we can see from the `super_comparison` typeclass and also multiple inheritance as the `io` typeclass shows.

Multiple Typeclass Instancess for a Specific Type

A typeclass can have different instantiations for a single type and they could be swapped even in runtime. As an example let us take a typeclass called 'Ordering' which denotes how the members of a type should be ordered. Then we have two instances of this typeclass, both implemented by a type, say a list. One implements an ascending order ordering and the other a descending order ordering. There should be a way to choose in runtime which of the two implementations the ordering would use.

So let's look at the following example, which will not compile.

```
instance ordering ord_ascend<vector> {
    fn(self)
    {
        ...
    }
}

instance ordering ord_descend<vector>{
    fn(self)
    {
        ...
    }
}
```

Here we can see an additional feature. Instances of a typeclass must have an extra identifier if we implement more than one instance of a typeclass for a type. But why will this not compile? Well simply because 2 different instances are declared for a type without specifying one as the default implementation for all objects of type vector.

```
instance ordering ord_ascend<vector> default {
    fn(self)
    {
        ...
    }
}

instance ordering ord_descend<vector> {
    fn(self)
    {
        ...
```

```
    }
}
```

With the above code we can declare the `ord_ascend` instance as default and as such all vector types unless otherwise specified will have this implementation for the ordering typeclass

And finally below we can see how to change the choice of typeclass instance in runtime.

```
a:list // ord_ascend
b:list<ordering: ord_ascend> // ord_ascend
c:list<ordering: ord_descend> //ord_descend
```