Specifications for Yudi's Checked Programming Language

Yudi Yang

I. Introduction

Yudi's Checked Programming Language (YCPL) is a new programming language aims at enhancing statically checking program correctness including sanity type checking and avoidance of pointer related security vulnerabilities including dangling pointer and illegal access of pointers. It is mostly inspired by Rust, but with a major difference that the code generated would be entirely in C, and the original coding structure would be preserved at best. That means that the C compiler (gcc, llvm) would handle all the code optimization. Also, programmers can provide additional information to the YCPL compiler to do additional code optimization.

II. Types

a) Boolean: bool

b) Integers: char, uint8, uint16, uint32, uint64, int8, int16, int32, int64

c) Floating Points: fp16, fp32, fp64

d) Immutable String: str

e) Array: for any type T can be defined with T[len], where len is a constant that specifies the length of array. An array with variable length can be specified with T[], which is stored in heap. Variable length array must be initialized before used.

f) Pointer: does not exist in YCPL.

g) Class: type that contains class variables and functions. Note that all class references are equivalent to pointer in C. Details in later chapters.

h) Enum: type that defines different sets of values stored. Details in later chapters.

III. Keywords

YCPL reserves a ranges of keywords to use:

a) Basics: var const function class this enum

b) Types: bool char uint8 uint16 uint32 uint64 int8 int16 int32 int64 fp16 fp32 fp64 str

c) Control Flow: if else match while for break continue return

d) Constants: True False

IV. Grammar

A. Naming

B. File Scope

In the scope of file, several grammars are possible:

a) Global Variables: Global variables are available in the scope after it is defined.

Variable definitions are

```
var Var1 : Type1; // Var1 declared with
    type Type1
var Var2 : Type2[5]; // an array with
    type Type2 defined on the stack with
    5 elements.
var Var3 : Type3[]; // an array with type
    Type3 defined on the heap which
    requires initializer
var Var4 = val; // a variable with value
    equals to val and type is infered;
const Const1 = val; // a constant with
    value equals to val
```

b) Function: Definition.

```
function Fun1(Var1 : Type1, Var2 : Type2)
    : ReturnType {
  // Function Scope
}
```

A function with generic type

```
function Fun1 <T> (Var1 : T, Var2 :
    Type2) : ReturnType {
// Function Scope
}
```

c) Class: types.

```
class ClassName {
  // instance variables
  var Var1 : Type1;
  var Var2 : Type2;
  // instance functions
  function Fun1(Var1 : Type1, Var2 :
      Type2) : ReturnType {
    // coding example
    this->Var1 = Var1;
  }

  // special function that works as
      initializer: ClassName(). Required
      for every class.
  ClassName(Var1 : Type1, Var2 : Type2)
      : ClassName {
    return this;
  }
}
```

A class with generic type

```
class ClassName <T, T1> {
  var Var1 : T;

  function Fun1(Var1 : T) : T1 {
    // Function Scope
  }
}
```

d) Enum: types.

```
enum EnumName {
  Name1;
  Name2(Var1 : Type1);
  Name3(Var1 : Type1, Var2 : Type2);
}
```

C. Function Scope

In the scope of functions, it is the actual program runs. There are several grammars

a) Evaluations and Assignment: Basic statements.

```
var a : int32 = 0;      // variable
    definition and initialization
var b = ClassA(a);      // class must have
    initializer
var c = EnumName.Name2(val1); // enum
    must have initializer
a = 3 * Func1(b);       // assigning to a
    variable
ClassA.call(a);         // evaluating a
    function
```

b) Branching: if-else branch (else may be omitted)

```
if (statement) {
 // if-true
} else {
 // if-false
}
```

c) Pattern Matching: an Enum type.

```
Var enum1 : Enum1 = Val1;
match (enum1) {
  Enum1.Name1(): enum1 = Enum1.Name2();
  Enum1.Name2: {
    statement1;
    statement2;
  }
  Enum1.Name3(a, b): enum1 =
     Enum1.Name3(a+1, b+1);
}
```

Note that pattern matching must cover all cases of possible enumerate types.

d) Loop: For loops

```
for (var i = 1; i < 100; i += 1) {
  statement;
}
```

While Loops

```
while (condition) {
statement;
}
```