Funk Programming Language Proposal

Naser AlDuaij, Senyao Du, Noura Farra, Yuan Kang, Andrea Lottarini
{nya2102, sd2693, naf2116, yjk2106, al3125} @columbia.edu

26 September, 2012

Motivation

Modern programming languages support multiple programming paradigms. They mix imperative, object oriented and functional programming styles into a single programming language. Notable examples include Python, Ruby, Go, Scala and C# 4.0. Some of these languages also provide an easier way to achieve parallel programming using high level constructs like tasks and channels, compared to manually managed threads and memory like in C.

In particular, compared to imperative code, functional code is more straightforward to parallelize, for each execution unit has clearer boundaries and more defined scope, suggesting a strong connection between the two.

For this project, we want to create a modern programming language that has an imperative structure, with parallel execution support and higher order functions. We will focus on creating a translator from this language to C code.

Language Description

The programming language is inspired by Go, which has very clean syntax and logical constructs. Funk is a generic programming language designed to support general mathematical and logical (boolean) operations. Our language has native support for higher order functions and parallel programming. Furthermore, in order to implement higher order functions, we need to implement closures.

Our language has specific constructs to define parallel computation. Similar to Go, it is possible to simply define blocks of code that run on separate threads. Compared to the threading libraries for C, our language allows expressing parallelism in an easier way.

Syntax Examples

Except for the additional features, the syntax of Funk is a subset of the syntax of Go.

```
func main() {
    /*
     * An imperative sequence of statements, including
     * declarations, assignments, conditional statements
     * and function calls.
     * Multi-line comments are allowed.
     */
    // So are single-line comments
    // a is 0, so is b
    var a, b int := 0, 0;
    // a_eq_b is TRUE
    var a_eq_b bool := a == b;
    for i := 0; i < 32; i++ {
        a += i
    }
    if a_eq_b && a == 0 {
        println("a is 0, so is b");
    } else {
        println("at least one of a b is not zero")
    }
}
```

Listing 1: Go-like syntax

Functions

The **func** keyword indicates that the block that follows is a routine. They are first class citizens in Funk so they can be assigned to variables and passed/returned as arguments. Since functions can be passed around, closures become necessary to bind free variables of a function to a specific variable.

```
// func ID (PARAMS) RETURN_TYPE
func outside_adder() func(int) int {
    // Closure
    var sum int := 10
    // Anonymous functions supported, without the ID
    return func(x int) int {
        sum += x
        return sum
    }
}
func main() {
    var adder func := outside_adder() //Using closure propery
    var another_adder func

    println(adder(5)) // 15

    //First class variable
    another_adder := adder
    println(another_adder(60)) // 75

    var yet_another_adder func := outside_adder() // A different closure

    println(yet_another_adder(60)) // 70
    println(adder(60)) // 135
}
```

Listing 2: Example of high order functions and closures

Parallel programming in Funk

Blocks that execute asynchronously are prefixed by the **async** keyword. We can wrap a generic piece of code inside an async block. Race conditions along with other common mistakes of parallel programming are avoided by performing a closure of the code inside the async block. In other words, the async block has no side effects. Only a single assignment to a variable in the outside scope is allowed (using the keyword *return*).

```
// Asynchronous Fibonacci sequence

func fib(n int) int {
   var a, b int

   // We don't wait for fib(n - 1)...
   a = async {
      return fib(n - 1)
   }
   // ...because we can do fib(n - 2) in parallel
   b = async {
      return fib(n - 2)
   }

   // but the program will wait here, since a and b are used
   return a + b
}
```

Listing 3: Parallel Fibonacci function in Funk.

Appendix

Example of code in Funk, demonstrating the conversion of Go-like [1] syntax into C.

```
// fib returns a function that returns
// successive Fibonacci numbers.
func fib() func() int {
   var a, b int := 0, 1
   return func() int {
      a, b := b, a+b
      return a
   }
}

func main() {
   f := fib()
   // Function calls are evaluated left-to-right.
   println(f()) //1
   println(f()) //1
   println(f()) //2
   println(f()) //3
   println(f()) //5
}
```

Listing 4: Fibonacci implemented with closures

Compiled C Code for the Fibonacci closure

```
#include <stdio.h>
```

[1]http://golang.org/

```c
#include <stdlib.h>

struct env {
   int a, b;
};

struct closure {
   int (*call)(struct env *);
   struct env * env;
};

int block(struct env * env) {
   int sum = env->a + env->b;
   env->a = env->b;
   env->b = sum;
   return env->a;
}

struct closure fib() {
   struct env * env = (struct env *) malloc(sizeof(struct env));
   env->a = 0;
   env->b = 1;

   struct closure closure;
   closure.env = env;
   closure.call = block;
   return closure;
}

int main() {
   struct closure c = fib();
   int i = 0;
   for (i = 0; i < 5; i++) {
      printf("%d ", c.call(c.env));
   }
   printf("\n");
   return 0;
}
```

Listing 5: Code of Listing 4 translated in C.