Kempe Compiler & Language Manual

Vanessa McHale

Contents

Introduction

Kempe is a stack-based language, and `kc` is a toy compiler for x86_64.

Installing kc

First, install cabal and GHC. Then:

```
cabal install kempe
```

This provides `kc`, the Kempe compiler.

Editor Integration

A vim plugin is available.

To install with vim-plug:

```
Plug 'vmchale/kempe' , { 'rtp' : 'vim' }
```

Kempe Language

Types

Kempe has a stack-based type system. So if you see a type signature:

```
next : Word -- Word Word
```

that means that the stack must have a `Word` on it for `next` to be invoked, and that it will have two `Word`s on the stack after it is invoked.

Polymorphism

Kempe allows polymorphic functions. So we can define:

```
id : a -- a
   =: [ ]
```

Builtins

The Kempe compiler has a few builtin functions that you can use for arithmetic and for shuffling data around. Many of them are familiar to stack-based programmers:

- `dup : a -- a a`
- `swap : a b -- b a`

There is one higher-order construct, `dip`, which we illustrate by example:

```
nip : a b -- b
    =: [ dip(drop) ]
```

Recursion

`kc` optimizes tail recursion.

Programming in Kempe

Invoking the Compiler

`kc` cannot be used to produce executables. Rather, the Kempe compiler will produce `.o` files which contain functions.

Internals

Kempe maintains its own stack and stores the pointer in `rbp`.

Kempe procedures do not require any registers to be preserved across function calls.

C Calls

When exporting to C, `kc` generates code that initializes the Kempe data pointer (`rbp`). Thus, one should avoid calling into Kempe code too often!

Kempe ABI

Examples

Splitmix Pseudorandom Number Generator

The generator in question comes from a recent paper.

Implementation turns out to be quite nice thanks to Kempe's multiple return values:

```
; given a seed, return a random value and the new seed
next : Word -- Word Word
     =: [ 0x9e3779b97f4a7c15u +~ dup
          dup 30i8 >>~ xoru 0xbf58476d1ce4e5b9u *~
          dup 27i8 >>~ xoru 0x94d049bb133111ebu *~
          dup 31i8 >>~ xoru
        ]

%foreign kabi next
```

Compare the C implementation:

```c
#include <stdint.h>

// modified to have ""multiple return"" since C doesn't really have that
uint64_t next(uint64_t x, uint64_t* y) {
    uint64_t z = (x += 0x9e3779b97f4a7c15);
    z = (z ^ (z >> 30)) * 0xbf58476d1ce4e5b9;
    z = (z ^ (z >> 27)) * 0x94d049bb133111eb;
    *y = x;
    return z ^ (z >> 31);
}
```

GCD

```
gcd : Int Int -- Int
    =: [ dup 0 =
         if( drop
           , dup dip(%) swap gcd )
       ]
```