Kempe Compiler & Language Manual

Vanessa McHale

Contents

Introduction

Kempe is a stack-based language, and `kc` is a toy compiler for x86_64 and aarch64.

Installing kc

First, install cabal and GHC. Then:

```
cabal install kempe
```

This provides `kc`, the Kempe compiler.

`kc` requires NASM when targeting x86_64.

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
   =: [ ]
```

Literals

Integer literals have type `-- Int`.

Positive literals followed by a `u` have type `-- Word`, e.g. `1u`.

Negative integer literals are indicated by an underscore, `_`, i.e. `_1` has type `-- Int`.

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`
- `drop : a --`

For arithmetic:

- `+ : Int Int -- Int`
- `* : Int Int -- Int`
- `- : Int Int -- Int`
- `/ : Int Int -- Int`
- `% : Int Int -- Int`
- `>> : Int Int -- Int`
- `<< : Int Int -- Int`
- `xori : Int Int -- Int`
- `+~ : Word Word -- Word`
- `*~ : Word Word -- Word`
- `/~ : Word Word -- Word`
- `%~ : Word Word -- Word`
- `>>~ : Word Word -- Word`
- `<<~ : Word Word -- Word`
- `xoru : Word Word -- Word`
- `popcount : Word -- Int`

- `= : Int Int -- Bool`
- `> : Int Int -- Bool`
- `< : Int Int -- Bool`
- `!= : Int Int -- Bool`
- `<= : Int Int -- Bool`
- `>= : Int Int -- Bool`
- `& : Bool Bool -- Bool`
- `|| : Bool Bool -- Bool`
- `xor : Bool Bool -- Bool`
- `~ : Int -- Int`

`%` is like Haskell's `rem` and `/` is like Haskell's `quot`. `>>`, `<<`, `>>~`, and `<<~` are like Haskell's `rotate`; i.e. they are logical shifts (not arithmetic shifts).

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

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

If Blocks

If-blocks are atoms which contain two blocks of atoms on each arm. If the next item on the stack is `True`, the first will be executed, otherwise the second.

```
loop : Int Int -- Int
    =: [ swap dup 0 =
         if( drop
           , dup 1 - dip(*) swap loop )
       ]

fac_tailrec : Int -- Int
            =: [ 1 loop ]
```

Sum Types

Kempe supports sum types, for instance:

```
type Maybe a { Just a | Nothing }
```

Note that empty sum types such as

```
type Void {}
```

are not really supported.

Pattern Matching

Sum types are taken apart with pattern matching, viz.

```
isJust : (Maybe a) -- Bool
      =: [
    { case
        | Just -> drop True
        | Nothing -> False
    }
]
```

Note that pattern matches in Kempe must be exhaustive.

Imports

Kempe has rudimentary imports. As an example:

```
import "prelude/fn.kmp"

type Pair a b { Pair a b }

...

snd : ((Pair a) b) -- b
    =: [ unPair nip ]
```

where **prelude/fn.kmp** contains

```
...

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

...
```

The import system is sort of defective.

FFI

Kempe can call into C functions. Suppose we have

```
int rand(void);
```

Then we can declare this as:

```
rand : -- Int
     =: $cfun"rand"
```

And `rand` will be available as a Kempe function.

Recursion

`kc` optimizes tail recursion.

Non-Features

Kempe is missing a good many features, such as:

- Floats
- Dynamically sized data types
- Strings
- Recursive data types
- Pointers
- Operator overloading

Programming in Kempe

Invoking the Compiler

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

Kempe functions can be exported with a C ABI:

```
fac : Int -- Int
    =: [ dup 0 =
        if( drop 1
          , dup 1 - fac * )
       ]

%foreign cabi fac
```

This would be called with a C wrapper like so:

```c
#include <stdio.h>

extern int fac(int);

int main(int argc, char *argv[]) {
    printf("%d", fac(3));
}
```

The C ABI should work on Unix; it does not target Windows.

There is also an alternate ABI, armabi, which takes a stack (to be used as the Kempe data stack) as the first argument. One would use it like so:

```
%foreign armabi fac

#include <stdio.h>
#include <stdlib.h>

extern int fact(void*, int);

int main(int argc, char *argv[]) {
    void* kptr = malloc(32 * 1024);
    printf("%d", fac(kptr, 3));
}
```

Unlike the frontend and type checker, the backend is dodgy.

Generating C Headers

kc has the cdecl subcommand, which generates headers from exported Kempe functions.

For the above example, one would get

```c
extern int fac(int);
```

for cabi and

```c
extern int fac(void*, int);
```

for armabi.

Cross-Compilation

`kc` is a cross-compiler; the target architecture can be set by passing one of `x64` or `aarch64` to `--arch`. By default `kc` targets the architecture of the host machine.

You will need the appropriate assembler installed.

Internals

Kempe maintains its own stack and stores the pointer in `rbp` (x86) or `x19` (aarch64).

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

C Calls

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

Note that the Kempe data pointer is static, so calling different Kempe functions in different threads will fail unpredictably.

Kempe ABI

Sum types have a guaranteed representation so that they can be used from other languages.

Consider:

```
type Param a b c
    { C a b b
    | D a b c
    }
```

Kempe types always have the same size; a value constructed with `C` will occupy the same number of bytes on the stack as a value constructed with `D`.

So, for instance

```
mkD : Int8 Int Int8 -- (((Param Int8) Int) Int8)
    =: [ D ]
```

will pad the value with 7 bytes, as a `(((Param Int8) Int) Int8)` constructed with `C` would be 7 bytes bigger.

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 30u >>~ xoru 0xbf58476d1ce4e5b9u *~
          dup 27u >>~ xoru 0x94d049bb133111ebu *~
          dup 31u >>~ xoru
       ]

%foreign kabi next
```

Compare this C implementation:

```c
#include <stdint.h>

// modified to have ""multiple return"" with destination-passing style
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 )
       ]
```

Mutual Recursion

kc supports mutual recursion:

```
odd : Int -- Bool
    =: [ dup 0 =
            if( drop False
               , - 1 even )
       ]

even : Int -- Bool
     =: [ dup 0 =
             if( drop True
                , - 1 odd )
        ]
```