Some benchmarks for transformations of the CEK machine

kwxm

December 2018

Introduction

This document contains some timing results for various versions of the CEK machine. If you look at the description of the CEK machine you can see that the `return` phase is essentially calling a sort of concretised continuation, so I wondered if things would be any faster with real continuations (short answer: no).

Experiments

I used the following abstract machines.

- Three versions of the CEK machine from commit `96fb387` (pretty much the earliest version), but with the bounds check for sized integers in `Constant/Make.hs` updated to use the `bit` function rather than calculating 2^n for large n (this was slowing things down quite substantially and has been fixed in more recent versions).
  - The original CEK machine
  - The CEK machine with a refunctionalisation transformation applied so that it uses explicit continuations rather than the frames and `return` operation in the contextual version of the CEK machine.
  - The refunctionalised version with an "un-CPS" transformation applied. This essentially turns the machine into a simple recursive evaluator providing a direct implementation of a standard structural operational semantics.
- The current CEK machine at commit `c9a8ae24`. This has been significantly modified from the earlier version, using monads and including the new infrastructure for "dynamic" built-in functions.

Olivier Danvy and a number of collaborators have done a lot of work on transformations of abstract machines (see "A Functional Correspondence between Evaluators and Abstract Machines", for example), and the transformations here are instances of the kind of thing they've studied.

Inputs

The programs were run with the inputs shown in Figure 1. These are the same (hand-written) programs and inputs as were used for evaluation of the lazy machine in a previous document, and the same statistics were used (collected using `/usr/bin/time -f "%U %S %M"` on Linux). The programs are all recursive programs using the Z combinator:

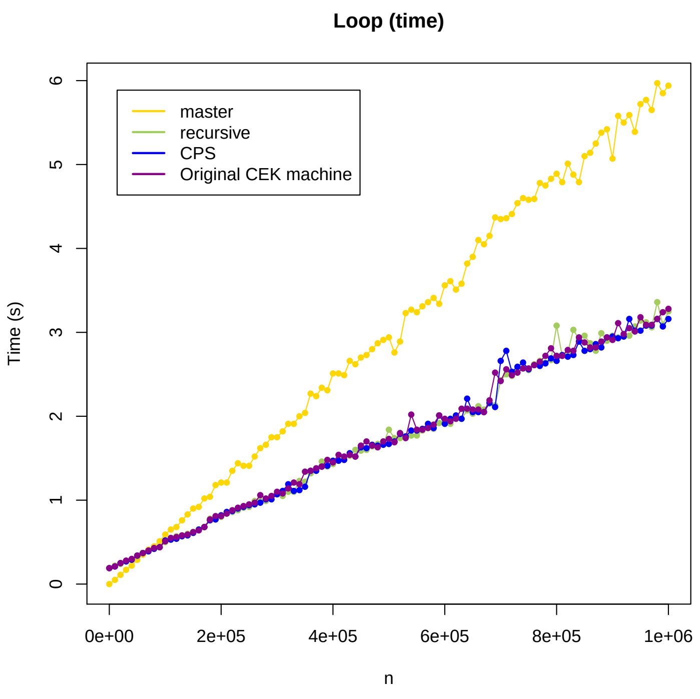
- `Loop`: loop n times.
- `Tri`: calculate $n + (n-1) + \ldots + 2 + 1$
- `Fac`: calculate $n(n-1) \cdots 2 \cdot 1$ (requires very large integers)
- `Fib`: Naive recursive Fibonacci

The programs were run once only for each input; ideally we'd run them several times each and take the average, but this would be a lengthy process and the results below don't suggest that we'd gain much from a more detailed test.

Program	Minimum input	Step	Maximum input	Integer size (bytes)
`Loop`	0	20,000	1,000,000	4
`Tri`	0	50,000	2,000,000	8
`Fac`	0	5,000	100,000	190,000
`Fib`	1	1	31	4

Figure 1: Programs and inputs

Results



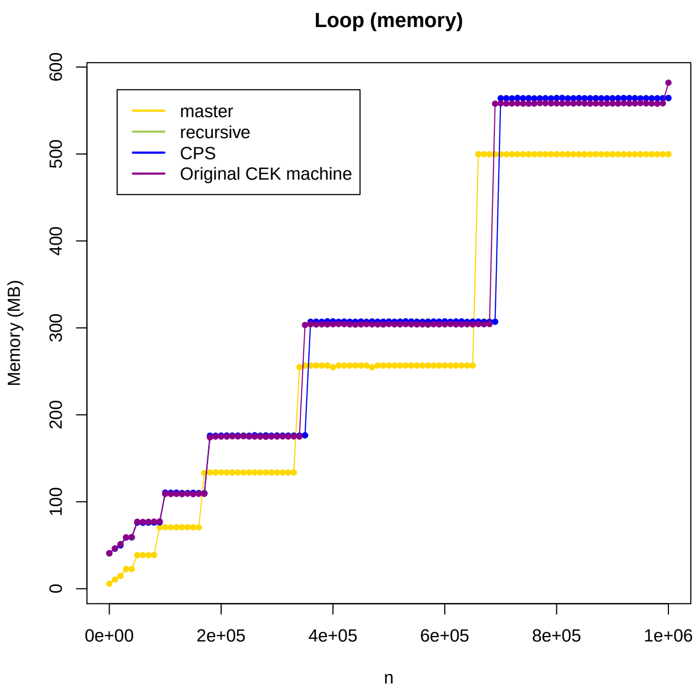
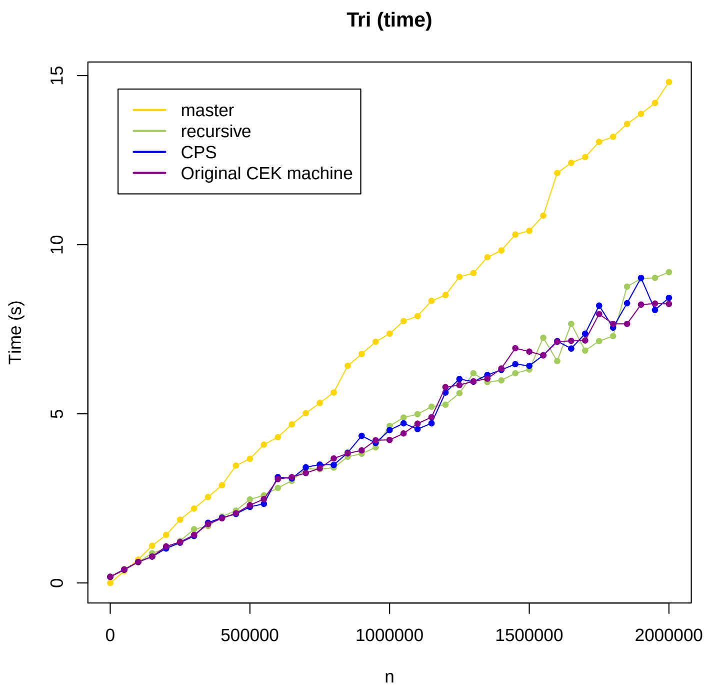


Figure 2: Loop



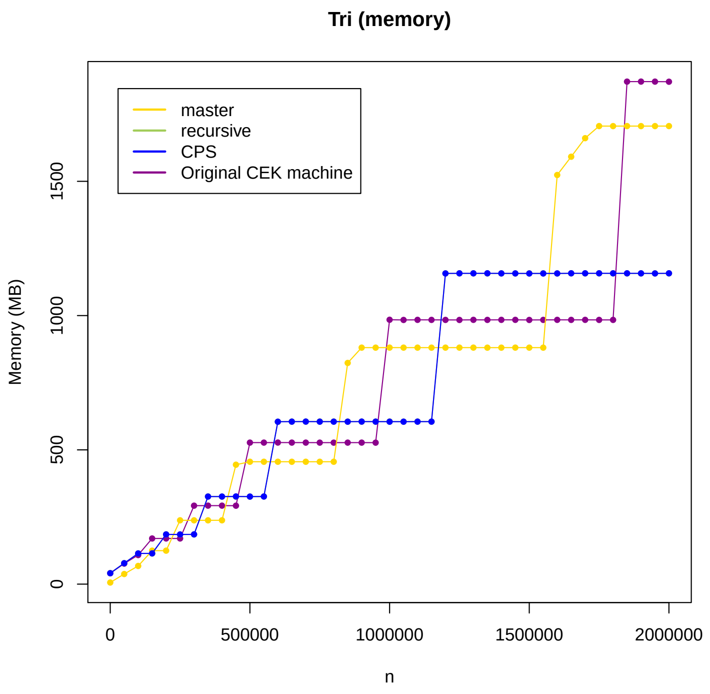
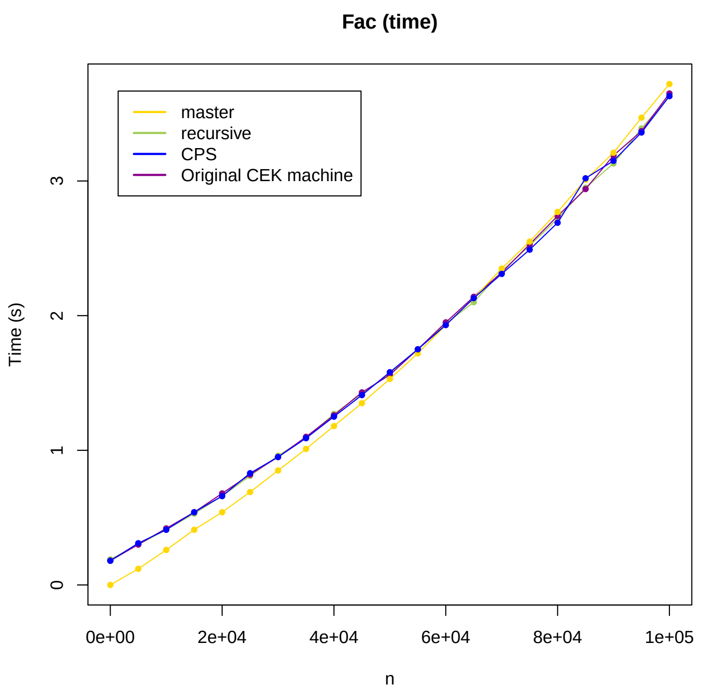


Figure 3: Triangular numbers



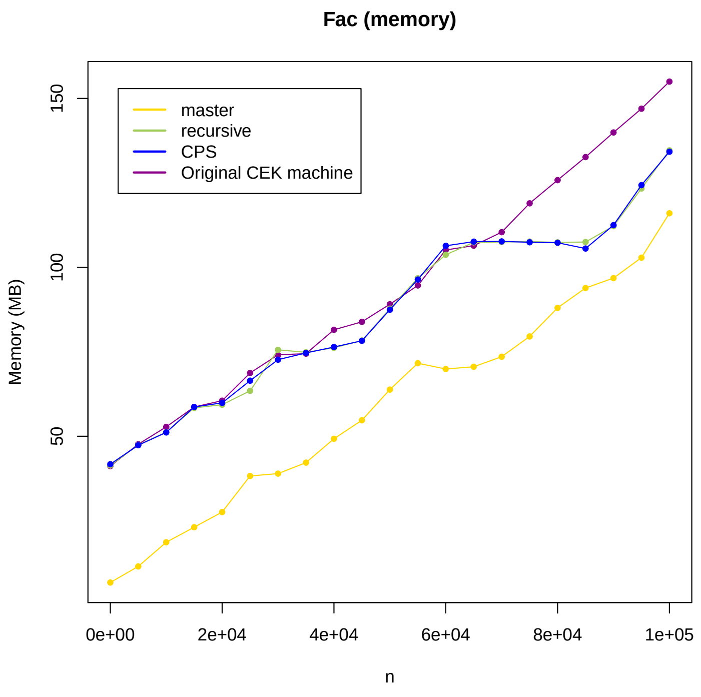
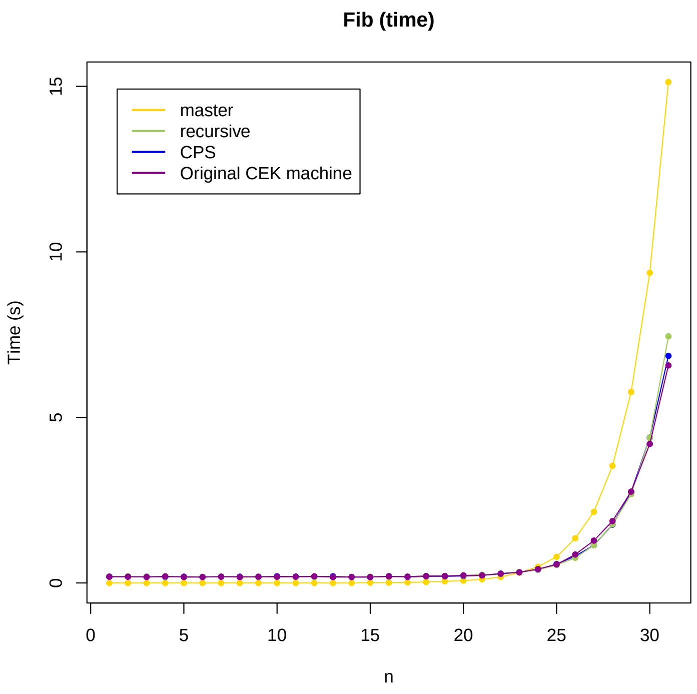


Figure 4: Factorial



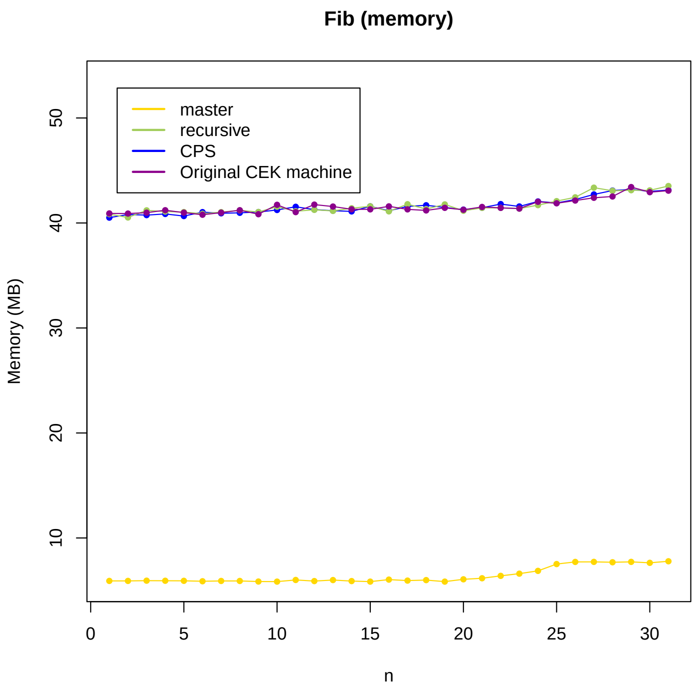


Figure 5: Fibonacci

Conclusions

The results are pretty inconclusive: the variations on the original CEK machine don't seem to make a lot of difference, possibly because GHC will be transforming things behind the scenes anyway.

It's notable that the current version of the CEK machine is quite a bit slower than the original one in some cases. This is presumably because it's quite a bit more complicated now, and also partly because there's at least one problem (to do with renaming variables in booleans) which we've identified but which I don't think has been fixed in the master branch yet.

It's also the case that the memory usage of the current version is significantly lower than the old version in some cases: I have no idea why this is. We should do some detailed profiling on complicated examples.