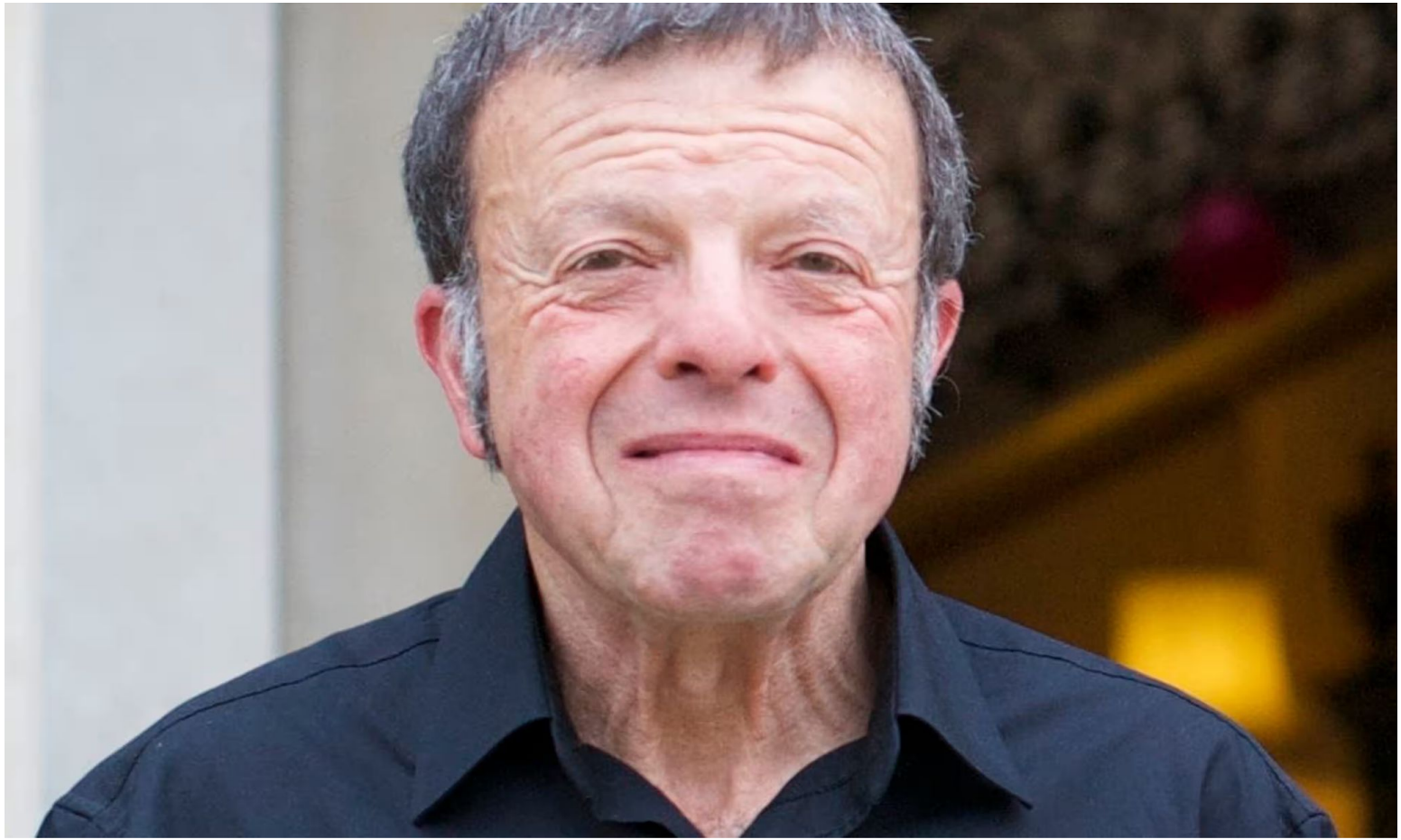
Combinators, revived
David Turner, in memoriam

Lennart Augustsson

WG2.8, Utrecht, 2024



David A.Turner, 1946-2023

A Little Personal History

- In the mid 1970 David Turner implemented the language SASL using the (abstract) SECD machine
- In 1978-79 he reimplemented it using SK **combinators**
- In 1980-81 Thomas Johnsson and I read the aforementioned paper
- It had a huge impact on us; it was the spark of the G-machine
- G-machine variations slowly became a (the?) standard way to implement lazy evaluation

BUT, SK combinators is still a simple and viable technique.

David's seminal paper

A New Implementation Technique for Applicative Languages
SOFTWARE-PRACTICE AND EXPERIENCE, VOL. 9, 31-49 (1979)

Describes a new implementation of SASL.

```
sieve (from 2)
where
from n = n: from (n+1)
sieve (p : x) = p: sieve (filter x)
                where
                filter (n : x) =
                      n rem p = 0 → filter x;
                      n: filter x
```

Figure 6. The list of all the prime numbers

```
sieve (from 2)
where
  from n = n : from (n + 1)
  sieve (p : x) = p : sieve (filter x)
                  where
                    filter (n : x) =
                      if n `rem` p == 0 then filter x
                      else n : filter x
```

Haskell version

Then and now

In 1981 I implemented combinator reduction à la Turner.
And in 2023 I did it again (hoping to show it to David).

Year	CPU	Word size	Clock	IPC	Cores	Memory	Language	Compiler
1981	TMS9900	16	3 MHz	1/8	1	64kB	C	homebrew
2023	AMD Ryzen	64	2.7-4.2 GHz	8?	64	256GB	C	gcc
2023	Apple M1	64	3.2 GHz	8	4+4	16GB	C	clang

MicroHs

I needed combinators to run.

So I wrote yet another Haskell compiler: MicroHs.

- Haskell2010 + a number of extensions.
- Translates Haskell to combinators + some primops (arithmetic etc.).
- Uses Scott encoding for data types.
- Minimal dependencies.

Cool features

- Runtime can serialize (almost) anything
- "Scan-free" mark-scan garbage collection

MicroHs

- Haskell2010 + a number of extensions.
  - BangPatterns ConstraintKinds DoAndIfThenElse DuplicateRecordFields EmptyDataDecls ExistentialQuantification ExtendedDefaultRules FlexibleContexts FlexibleInstance ForeignFunctionInterface FunctionalDependencies IncoherentInstances KindSignatures MonoLocalBinds MultiParamTypeClasses NamedFieldPuns NegativeLiterals NoMonomorphismRestriction NoStarIsType OverlappingInstances OverloadedRecordDot OverloadedRecordUpdate OverloadedStrings PolyKinds RankNTypes RecordWildCards QualifiedDo ScopedTypeVariables StandaloneKindSignatures TupleSections TypeLits TypeSynonymInstances UndecidableInstances UndecidableSuperClasses ViewPatterns

Demo

Bracket abstraction

We need a way to translate lambda expressions to equivalent combinator expressions, *bracket abstraction*.

We define it so that

$\lambda x.\ e = [x]e$

$[x]x = \mathbf{I}$
$[x]y = \mathbf{K}\ y$
$[x](e_1\ e_2) = \mathbf{S}\ ([x]e_1)\ ([x]e_2)$
$[x](\lambda y.\ e) = [x]([y]e)$

And behold, there are no longer any (bound) variables!

More combinators

Similar to **S**, but only sends the last argument one way:

B $f\ g\ x = f\ (g\ x)$
C $f\ g\ x = (f\ x)\ g$

Some more variations that are pretty common

S' $k\ f\ g\ x = k\ (f\ x)\ (g\ x)$
B' $k\ f\ g\ x = k\ f\ (g\ x)$
C' $k\ f\ g\ x = k\ (f\ x)\ g$

Simplification rules

Some compile time rewrites

$$\mathbf{S}\ (\mathbf{K}\ e_1)\ (\mathbf{K}\ e_2) = \mathbf{K}\ (e_1\ e_2)$$
$$\mathbf{S}\ (\mathbf{K}\ e)\ \mathbf{I} = e$$
$$\mathbf{S}\ (\mathbf{K}\ e_1)\ e_2 = \mathbf{B}\ e_1\ e_2$$
$$\mathbf{S}\ e_1\ (\mathbf{K}\ e_2) = \mathbf{C}\ e_1\ e_2$$
$$\mathbf{S}\ (\mathbf{B}\ e_1\ e_2)\ e_3 = \mathbf{S'}\ e_1\ e_2\ e_3$$
$$\mathbf{C}\ (\mathbf{B}\ e_1\ e_2)\ e_3 = \mathbf{C'}\ e_1\ e_2\ e_3$$
$$\mathbf{B}\ (e_1\ e_2)\ e_3 = \mathbf{B'}\ e_1\ e_2\ e_3$$

Simplification Rules

It makes a huge difference

λ*x*. λ*y*. *a y x*

turns into

S (**S** (**K S**) (**S** (**S** (**K S**) (**S** (**K K**) (**K** a))) (**K I**))) (**S** (**K K**) **I**)

simplified

B (**S** a) **K**

For a different take on this, see Oleg Kiselyov, *λ to SKI, Semantically*

Ad hoc set of combinators

		David	Haskell	Smullyan*
S	`x y z = x z (y z)`	Y	`(<*>)`	Starling
K	`x y = x`	Y	`const, False, []`	Kestrel
I	`x = x`	Y	`id`	Idiot
B	`x y z = x (y z)`	Y	`(.)`	Bluebird
C	`x y z = x z y`	Y	`flip`	Cardinal
S'	`x y z w = x (y w) (z w)`	Y		
B'	`x y z w = x y (z w)`	Y		Dove
C'	`x y z w = x (y w) z`	Y		
U	`x y = y x`	Y	`uncurry`	Thrush
P	`x y z = z x y`	Y	`(,)`	Vireo
A	`x y = y`		`True`	Kite
Z	`x y z = x y`			
R	`x y z = y z x`			Robin
O	`x y z w = w x y`		`(:)`	
Y	`x = x (Y x)`		`fix`	Why Bird

* Raymond Smullyan, To Mock a Mockingbird, 1985

Self-optimization

David said:

> The second principal advantage is that the combinatory 'code' turns out to have some remarkable self-optimizing properties including that constant calculations are automatically moved outside loops and that the overhead cost of calling a user-defined function falls to zero after the first occasion of its use.

f x = (2+3) + *x*

f = **intAdd** (**intAdd** 2 3)

The first time *f* is used the (**intAdd** 2 3) reduction will happen and the new definition will be
f = **intAdd** 5

Implementing overloading (self-optimization)

Source

```
class Num a where
  (+) :: a -> a -> a
  fromInteger :: Integer -> a
  ...


instance Num Int where
  (+) = primIntAdd
  fromInteger = primIToInt
  ...

inc :: forall a . Num a => a -> a
inc x =
  x + 1


incInt :: Int -> Int
incInt = inc
```

Transformed

```
data Num a = Num {
  (+) :: a -> a -> a,
  fromInteger :: Integer -> a
  ... }

instNumInt :: Num Int
instNumInt = Num {
  (+) = primIntAdd,
  fromInteger = primIToInt
  ... }

inc :: forall a . Num a -> a -> a
inc numDict x =
  ((+) numDict) x
          (fromInteger numDict 1)

incInt :: Int -> Int
incInt = inc instNumInt
```

Implementing overloading (self-optimization)

Transformed

```
data Num a = Num {
  (+) :: a -> a -> a,
  fromInteger :: Integer -> a
  ... }

instNumInt :: Num Int
instNumInt = Num {
  (+) = primIntAdd,
  fromInteger = primIToInt
  ... }

inc :: forall a . Num a -> a -> a
inc numDict x =
  ((+) numDict) x
          (fromInteger numDict 1)

incInt :: Int -> Int
incInt = inc instNumInt
```

Reduction (actually happens with combinators):

```
incInt ⟶

inc instNumInt ⟶
\x-> ((+) intNumInt) x
       (fromInteger instNumInt 1)

⟶

\x-> primIntAdd x (primIToInt 1) ⟶

\x-> primIntAdd x #1 ⟶

\x-> flip primIntAdd #1 x ⟶

flip primIntAdd #1 ⟶
C primIntAdd #1
```

Demo

```
inc :: Num a => a -> a
inc x = x + 1

incInt :: Int -> Int
incInt = inc
```

Easy execution

After conversion to combinators we have this data type (assuming only Int literals):

```
data Exp = App Exp Exp | Comb String | LitInt Int
```

How can we convert an expression tree (an `Exp`) to the value it represent?
Ignoring types, it's very easy!

```
translate :: Exp -> Any
translate (LitInt i) = unsafeCoerce i
translate (Comb s)   = lookupComb s
translate (App f a)  = (unsafeCoerce (translate f)) (translate a)

lookupComb :: String -> Any
lookupComb "K" = unsafeCoerce const
lookupComb "I" = unsafeCoerce id
lookupComb "S" = unsafeCoerce $ \ f g x -> (f x) (g x)
lookupComb "+" = unsafeCoerce $ ((+) :: Int->Int->Int)
```

Combinator evaluation

The runtime system has two evaluators:

- `eval()`
  - evaluate a pure term, IO terms are considered in normal form
- `execio()` (which is really the same as `unsafePerformIO`)
  - execute an IO term, pure terms will not be encountered
  - implements `return` and `(>>=)` directly

Eval

The eval procedure in words:

1. Start at the top node
2. Go down the left spine of applications
3. At a primitive
    a. If combinator, rewrite according to rule
    b. If an arithmetic primitive
        i. recursively eval the arguments
        ii. do the arithmetic
4. At the updated node
    a. If an application, go to 2
    b. If a constant, done

To find nodes higher up for step 3, keep a stack of pointers to the spine.

eval(), some C code

```
for(;;) {
  tag = GETTAG(n);
  switch(tag) {
  case T_IND:  n = INDIR(n); break;
  case T_AP:   PUSH(n); n = FUN(n); break;
  case T_INT:  RET;
  case T_S:    GCCHECK(2); CHKARG3; GOAP(new_ap(x, z), new_ap(y, z));             /* S x y z = x z (y z) */
  case T_SS:   GCCHECK(3); CHKARG4; GOAP(new_ap(x, new_ap(y, w)), new_ap(z, w)); /* S' x y z w = x (y w) (z w) */
  case T_K:                CHKARG2; GOIND(x);                                    /* K x y = *x */
  case T_A:                CHKARG2; GOIND(y);                                    /* A x y = *y */
  case T_U:                CHKARG2; GOAP(y, x);                                  /* U x y = y x */
  case T_I:                CHKARG1; GOIND(x);                                    /* I x = *x */
  case T_Y:                CHKARG1; GOAP(x, n);                                  /* n@(Y x) = x n */
  case T_B:    GCCHECK(1); CHKARG3; GOAP(x, new_ap(y, z));                       /* B x y z = x (y z) */
  case T_BB:   GCCHECK(2); CHKARG4; GOAP(new_ap(x, y), new_ap(z, w));            /* B' x y z w = x y (z w) */
  case T_Z:                CHKARG3; GOAP(x, y);                                  /* Z x y z = x y */
  case T_C:    GCCHECK(1); CHKARG3; GOAP(new_ap(x, z), y);                       /* C x y z = x z y */
  case T_CC:   GCCHECK(2); CHKARG4; GOAP(new_ap(x, new_ap(y, w)), z);            /* C' x y z w = x (y w) z */
  case T_P:    GCCHECK(1); CHKARG3; GOAP(new_ap(z, x), y);                       /* P x y z = z x y */
  case T_R:    GCCHECK(1); CHKARG3; GOAP(new_ap(y, z), x);                       /* R x y z = y z x */
  case T_O:    GCCHECK(1); CHKARG4; GOAP(new_ap(w, x), y);                       /* O x y z w = w x y */
  case T_ADD:  ARITHBIN(+);
  case T_SUB:  ARITHBIN(-);
  ...
  }
}
```

execio(), some C code

```
for(;;) {
  eval(n);
  tag = GETTAG(n);
  switch(tag) {
  case T_IND:       n = INDIR(n); break;
  case T_AP:        PUSH(n); n = FUN(n); break;
  case T_IO_BIND:   CHECKIO(2);   x = execio(ARG(TOP(1))); f = ARG(TOP(2)); n = new_ap(f, x); POP(3); break;
  case T_IO_THEN:   CHECKIO(2); (void)execio(ARG(TOP(1))); n = ARG(TOP(2));                    POP(3); break;
  case T_IO_RETURN: CHECKIO(1); n = ARG(TOP(1)); RETIO(n);
  case T_IO_CCALL:  ...
  case T_IO_CATCH:  ... setjmp(jbuf); ...
  ...
  default: ERR("non-IO encountered");
  }
}
```

execio(), some C code

```
for(;;) {
  eval(n);
  tag = GETTAG(n);
  switch(tag) {
  case T_IND:       n = INDIR(n); break;
  case T_AP:        PUSH(n); n = FUN(n); break;
  case T_IO_BIND:   CHECKIO(2);   x = execio(ARG(TOP(1))); f = ARG(TOP(2)); n = new_ap(f, x); POP(3); break;
  case T_IO_THEN:   CHECKIO(2); (void)execio(ARG(TOP(1))); n = ARG(TOP(2));                    POP(3); break;
  case T_IO_RETURN: CHECKIO(1); n = ARG(TOP(1)); RETIO(n);
  case T_IO_CCALL:  ...
  case T_IO_CATCH:  ... setjmp(jbuf); ...
  ...
  default: ERR("non-IO encountered");
  }
}
```

This code has a problem. IO bind operations are often left biased, e.g.,

```
(((a >>= b) >>= c) >>= d >>= e)
```

This means that the recursive `execio()` calls can get deeply nested.

Solution: recognize this at runtime and reassociate.

```
(m >>= g) >>= h      →  m >>= (\ x -> g x >>= h)
```

With combinators:

```
BIND ((BIND m) g) h  →  (BIND m) (((C' BIND) g) h)
```

Runtime system

- Very few dependencies:
  - malloc, free, str*()
  - optional stdio
  - optional floating point and math library
  - optional <unistd.h>
- Mark-scan garbage collector
  - Marking in a bit map, no explicit scan phase.
  - Each allocation scans the bit map
    - Uses FFS (Find First Set) instruction for fast allocation
- Easily portable
  - runs on macOS, Windows, Linux
  - runs with 32 and 64 bit words (minimal conditional compilation)
  - runs on i386, x86_64, ARM, RISC-V, S390
- Reasonably small binaries
  - Combinator file size for hello-world: 712 bytes
  - Executable size for hello-world: 75920
  - Compressed (upx) executable size for hello-world: 28408
  - mhs compressed executable: 130k

Why?

Does the world need another Haskell compiler?

😀 I don't care. I'm doing this for fun.

😐 Maybe. GHC isn't known for being minimal.

😕 No. GHC is the defacto standard.

- Goal: Must recompile itself in <10s
- I failed, but there is a solution:
  buy a new laptop

Future work

Whatever I feel like.

I have a long list.

Questions