The AMEN calculator

April 9, 2018

Contents

1 Benedictus benedicat

Some haskell boilerplate. We are going to play around with the ordinary arithmetical symbols, and versions of these symbols in angle-brackets eg. $< + >$.

```
module Arithmetic where
import Data.Char
import Prelude hiding
  ((×), (∧), (+), (˙)
  , (< ∗ >), (< ˆ >), (< + >), (<<>>)
  , (:∧: ), (: ×: ), (:+: ), (:<>:)
  , pi
  )
infixr 8 ∧
infixr 7 ×
infixr 6 +
infixr 9 ˙
main :: IO ()   -- Do something with this later.
main = let eg = " test $ vc :^: vb :^: va :^: cC "
       in putStrLn
         ("Load in ghci and type something like: " ++ eg)
```

2 Real-world arithmetical combinators

Here are some simple definitions of binary operations corresponding to the arithmetical combinators:

$$
\begin{aligned}
&a \wedge b = b\ a \\
&a \times b = \lambda c \to (c \wedge a) \wedge b \\
&a + b = \lambda c \to (c \wedge a) \times (c \wedge b) \\
&a\ \text{`}\mathit{naught}\text{`}\ b = b
\end{aligned}
$$

Instead of *naught*, one can use the infix operator $(\dot{})$, that looks a little like a '0'. It discards its left argument, and returns its right.

$$
\begin{aligned}
&(\dot{}) = \mathit{naught} \\
&\mathit{zero} = \mathit{naught} \\
&\mathit{one} = \mathit{zero} \wedge \mathit{zero} \\
&\mathit{suc}\ n\ s = n\ s \times s \\
&\mathit{two} = \mathit{suc}\ \mathit{one} \\
&\mathit{three} = \mathit{suc}\ \mathit{two} \\
&\mathit{four} = \mathit{two} \wedge \mathit{two}
\end{aligned}
$$

$five = two + three$
$six \ = two \times three$
$seven = four + three$
$eight = two \wedge three$
$nine = three \wedge two$
$ten \ = two \times five$

The type-schemes inferred for the definitions are as follows:

$(\wedge) :: a \to (a \to b) \to b$
$(\times) :: (a \to b) \to (b \to c) \to a \to c$
$(+) :: (a \to b \to c) \to (a \to c \to d) \to a \to b \to d$
$() \quad :: a \to b \to b$
$one :: a \to a$

2.1 Infinitary operations: streams and lists

For infinitary operations (this may come later) I need these

$pfs :: (a \to a \to a) \to a \to [\,a\,] \to [[\,a\,]]$
$pfs\ op\ ze\ xs = [\,b\ ze \mid (b, \_) \leftarrow pfs'\ xs\ id\,]$
  where $pfs'\ (x : xs)\ b = pfs'\ xs\ (b \circ (op\ x))$
type $EE\ x = x \to x$
$pi :: [\,EE\ a\,] \to [[\,EE\ a\,]]$
$sigma :: [\,a \to EE\ b\,] \to [[\,a \to EE\ b\,]]$
$pi = pfs\ (\times)\ one$
$sigma = pfs\ (+)\ zero$
type $N\ x = EE\ (EE\ x)$
$index :: N\ [\,a\,] \to [\,a\,] \to a$
$index\ n = head \circ n\ tail$
  -- note index 0 is head
type $C\ x\ y = (x \to y) \to y$
$ret :: x \to C\ x\ y$
$ret = (\wedge)$
$mu :: C\ (C\ x\ y)\ y \to C\ x\ y$
$mu\ mm\ k = mm\ (ret\ k)$
 {-The types $x \to C\ x\ x$ and $N\ x$ are isomorphic. -}
 {-The same combinator is used in both inverse directions! -}
$myflip :: (x \to C\ x\ x) \to N\ x$
$myflip = flip$
$myflip' :: N\ x \to x \to C\ x\ x$
$myflip' = flip$
  -- any of the following type statements will do
$mydrop :: C\ (EE\ [\,a\,])\ y$
  -- $mydrop :: (([\,a\,] \to [\,a\,]) \to t) \to t$

```
-- mydrop :: (EE [a] → t) → t
-- mydrop :: N [a] → EE [a]
mydrop n = n tail
mydrop'  = ($tail)
```

pfs is applied only to streams, and returns a stream. Think of it is a stream of finite lists, namely the list of finite prefixes of a stream. Then we fold an operation over each list, starting with a constant.

3 Syntax-world arithmetical combinators

The defining equations above generate an equivalence relation between (possibly open) terms in a signature with eight operators:

- 4 constants $(\wedge)$, $(\times)$, $(+)$ and $()$
- 4 binary operators $\_\hat{}\_$, $\_*\_$, $\_+\_$ and $\_<>\_$

This is the least equivalence relation extending the definitions, congruent to all operators in the signature. This means that equations between open terms can be proved by substituting equals for equals.

One can also allow instances of the following "ζ-rule" in proving equations.

$$x \wedge a = x \wedge b \implies a = b$$

with the side condition that x is fresh to both a and b.

The ζ-rule is a cancellation law. It expresses 'exponentiality': two expressions that behave the same as exponents of a generic base (as it were, a cardboard-cutout of a base) are equivalent. I shall call this equivalence relation ζ-equality. Any equation I assert should be interpreted as ζ-equations, unless I say otherwise.

It may be that to determine the behaviour of an expression as an exponent, we have to supply it with more than one base-variable. Sometimes, 'extra' variables play a role in allowing computations to proceed, and subsequently can be cancelled.

4 Evaluating arithmetical expressions

The arithmetical combinators are rather fascinating, but it is easy to make mistakes when performing calculations. We now write some code to explore on the computer the evaluation of arithmetical expressions, built out of our four/eight combinators.

First, here is a datatype E for arithmetical expressions. The symbols for the constructors are chosen to suggest their interpretation as combinators.

```
infixr 9 :<>:
infixr 8 :∧:
```

```
infixr 7 :×:
infixr 6 :+:
```

```
data E = V String | C String
       | E :∧: E
       | E :×: E
       | E :+: E
       | E :<>: E
       | E :˜: E    -- flip
       | E :&: E    -- pairing
       deriving (Eq)   -- (Show,Eq)
```

We can think of these expressions as fancy Lisp S-expressions, with four different binary 'cons' operations, each with an distinct arithmetical flavour.

It is convenient to have atomic constants identified by arbitrary strings. The constants `"+"`, `"*"`, `"^"`, `"0"`, `"1"` are treated specially.

```
(cA, cM, cE, c0, c1, cC_new, cPair_new, c0')
  = ( V "+"    -- AMEN
    , V "*"
    , V "^"
    , V "0"
    , V "1"    -- de trop combinators
    , V "~"    -- flip
    , V "&"    -- pairing
    , V "<>"   -- discard left-hand argument
    )
```

For each arithmetical operator, we define a function that takes two arguments, and (sometimes) returns a 'normal' form of the expression formed with that operator. (This doesn't allow us to trace reduction sequences – we turn to that later.) The definitions correspond to the transitions of an arithmetical machine.

```
infixr 9 <<>>
infixr 8 <^>
infixr 7 <*>
infixr 6 <+>
```

```
(<+>), (<*>), (<^>), (<<>>) :: E → E → E
a <+> b = case a of V "0"        → b
                    _            → case b of V "0"     → a
                                             b1 :+: b2 → (a <+> b1) <+> b2
                                             _         → a :+: b
a <*> b = case a of _ :∧: V "0"  → b
                    _            → case b of V "0"     → b
```

$$
\begin{array}{lll}
\mathit{b1} :\!+\!: \mathit{b2} & \rightarrow (a < * > \mathit{b1}) < + > (a < * > \mathit{b2}) \\
\mathit{b1} :\!\times\!: \mathit{b2} & \rightarrow (a < * > \mathit{b1}) < * > \mathit{b2} \\
\_ :\!\wedge\!: V\ \texttt{"0"} & \rightarrow a \\
\_ & \rightarrow a :\!\times\!: b
\end{array}
$$

$$
\begin{array}{lll}
a < \hat{} > b = \textbf{case}\ b\ \textbf{of} & V\ \texttt{"0"} & \rightarrow b :\!\wedge\!: b \\
 & \mathit{b1} :\!+\!: \mathit{b2} & \rightarrow (a < \hat{} > \mathit{b1}) < * > (a < \hat{} > \mathit{b2}) \\
 & \mathit{b1} :\!\times\!: \mathit{b2} & \rightarrow (a < \hat{} > \mathit{b1}) < \hat{} > \mathit{b2} \\
 & \_ :\!\wedge\!: V\ \texttt{"0"} & \rightarrow a \\
 & \mathit{b1} :\!\wedge\!: V\ \texttt{"\^{}"} & \rightarrow \mathit{b1} < \hat{} > a \quad \text{-- note: destroys termination} \\
 & \mathit{b1} :\!\wedge\!: V\ \texttt{"*"} & \rightarrow \mathit{b1} < * > a \\
 & \mathit{b1} :\!\wedge\!: V\ \texttt{"+"} & \rightarrow \mathit{b1} < + > a \\
 & \mathit{b1} :\!\wedge\!: \mathit{b2} & \rightarrow a :\!\wedge\!: (\mathit{b1} < \hat{} > \mathit{b2}) \\
 & \_ & \rightarrow a :\!\wedge\!: b \\
\_ << >> b = b & &
\end{array}
$$

The following (partial!) function then evaluates an arithmetic expression.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathit{eval} :: E \rightarrow E} \\
\mathit{eval}\ a = \textbf{case}\ a\ \textbf{of} & \mathit{b1} :\!+\!: \mathit{b2} & \rightarrow \mathit{eval}\ \mathit{b1} < + > \mathit{eval}\ \mathit{b2} \\
 & \mathit{b1} :\!\times\!: \mathit{b2} & \rightarrow \mathit{eval}\ \mathit{b1} < * > \mathit{eval}\ \mathit{b2} \\
 & \mathit{b1} :\!\wedge\!: \mathit{b2} & \rightarrow \mathit{eval}\ \mathit{b1} < \hat{} > \mathit{eval}\ \mathit{b2} \\
 & \_ :\!<>\!: \mathit{b2} & \rightarrow \mathit{eval}\ \mathit{b2} \\
 & \_ & \rightarrow a
\end{array}
$$

This first piece of code is an evaluator, that computes the normal form of an expression (with respect to some rewriting rules hard-wired in the code), unless it "hangs", or consumes all the memory in your computer.) Such an evaluator may let look at the normal form of an expression, if it has one. but it doesn't show how this was arrived at. (This is done below.)

There are various systems and reduction strategies of interest. They arise from the algebraic structure: the additive and multiplicative monoids, weak distributivity, etc.

5 Rewriting arithmetical expressions

If an expression does not have a value, then the *eval* function of the last section will not produce one, thank heavens. Nevertheless, one may want to observe finite segments of the sequence of reductions. The second piece of code is for watching the reduction rules in action.

The machinery is controlled by a single (case-)table of top-level reductions, in the function *tlr* below. This maps an expression to the list of expressions to which it can be reduced in one top-level step (rewriting the root of the expression). To vary the details of reduction, one can tinker with the definition of *tlr*.

Although the lists returned here are at most singletons, in other variants there might be overlap: more than one reduction rule might apply. In such a case, the order of pattern matching might matter.

$$
\begin{array}{l}
\mathit{tlr} :: E \to [E] \\
\mathit{tlr}\ e = \textbf{case}\ e\ \textbf{of} \\
\quad \begin{array}{lll}
(a :+: (b :+: c)) & \to [(a :+: b) :+: c\ ] & \text{-- space reuse} \\
(V\ \texttt{"0"} :+: a) & \to [a\ ] & \text{-- drop1} \\
(a :+: V\ \texttt{"0"}) & \to [a\ ] & \text{-- drop1} \\
(a :\times: (b :+: c)) & \to [(a :\times: b) :+: (a :\times: c)] & \text{-- 2 to 3} \\
(a :\times: V\ \texttt{"0"}) & \to [\mathit{c0}\ ] & \text{-- drop1} \\
(a :\times: (b :\times: c)) & \to [(a :\times: b) :\times: c\ ] & \text{-- reuse} \\
(V\ \texttt{"1"} :\times: a) & \to [a\ ] & \text{-- drop1} \\
(a :\times: V\ \texttt{"1"}) & \to [a\ ] & \text{-- drop1} \\
(a :\wedge: (b :+: c)) & \to [(a :\wedge: b) :\times: (a :\wedge: c)\ ] & \text{-- 2 to 3} \\
(a :\wedge: V\ \texttt{"0"}) & \to [\mathit{c1}\ ] & \text{-- drop1} \\
(a :\wedge: (b :\times: c)) & \to [(a :\wedge: b) :\wedge: c\ ] & \text{-- reuse} \\
(a :\wedge: V\ \texttt{"1"}) & \to [a\ ] & \text{-- drop1 -- idle} \\
(a :\wedge: b :\wedge: V\ \texttt{"+"}) & \to [b :+: a\ ] & \text{-- drop1} \\
(a :\wedge: b :\wedge: V\ \texttt{"*"}) & \to [b :\times: a\ ] & \text{-- drop1} \\
(a :\wedge: b :\wedge: V\ \texttt{"\^{}"}) & \to [b :\wedge: a\ ] & \text{-- drop1 -- top 2 swap} \\
(a :\wedge: b :\wedge: V\ \texttt{"0"}) & \to [b :<>: a\ ] & \text{-- drop1 -- indirection} \\
(a :\wedge: b :\wedge: V\ \texttt{"\~{}"}) & \to [b :\tilde{}: a\ ] & \text{--} \\
(a :\wedge: b :\wedge: V\ \texttt{"\&"}) & \to [b :\&: a\ ] & \text{--} \\
(a :\wedge: (b :\&: c)) & \to [c :\wedge: b :\wedge: a\ ] & \text{-- a permutation/swap} \\
(a :\wedge: (b :\tilde{}: c)) & \to [c :\wedge: a :\wedge: b\ ] & \text{-- another permutation/swap} \\
(a :<>: b) & \to [b\ ] & \text{-- drop1} \\
(V\ \texttt{"0"} :\wedge: V\ \texttt{"+"}) & \to [\mathit{c1}\ ] & \\
(V\ \texttt{"1"} :\wedge: V\ \texttt{"*"}) & \to [\mathit{c1}\ ] & \\
(V\ \texttt{"\~{}"} :\times: V\ \texttt{"\~{}"}) & \to [\mathit{c1}\ ] & \\
(V\ \texttt{"\^{}"} :\times: V\ \texttt{"\~{}"}) & \to [V\ \texttt{"\&"}\ ] & \\
\_ & \to [\ ] &
\end{array}
\end{array}
$$

Thought: the associativity laws can be done in place. The distribution laws cannot. Quite a few others can reuse the redex as an indirection node.

To represent subexpressions, we use a 'zipper', in a form in which the context of a subexpression is represented by a linear function. We represent each part e of an expression e' (at a particular position) as a pair (f, e) consisting of the subexpression e there, and a linear function f such that $f\ \ e = e'$. (By construction, the function is linear in the sense that it uses its argument exactly once.) Intuitively you 'plug' the subexpression e into the 'context' f to get back e'.

The function *sites* returns (in top down preorder: root, left, right ...) all the subexpressions of a given expression, together with the one-hole contexts in which they occur. This includes the improper case of the expression itself in the empty context. I represent the one-hole contexts by a composition of functions that when applied to the contextualised part will return the outermost expression.

$$
\begin{array}{l}
\mathit{sites} :: E \to [(E \to E, E)] \\
\mathit{sites}\ e = (\mathit{id}, e) : \textbf{case}\ e\ \textbf{of}
\end{array}
$$

$$
\begin{array}{lll}
\quad (a :+: b) & \rightarrow h\ (:+:)\ a\ b & \\
\quad (a :\times: b) & \rightarrow h\ (:\times:)\ a\ b & \\
\quad (a :\wedge: b) & \rightarrow h\ (:\wedge:)\ a\ b & \\
\quad (a :<>: b) & \rightarrow \mathit{sites}\ b & \text{-- DANGER! indirection} \\
\quad \_ & \rightarrow [\,] & \text{-- no internal sites}
\end{array}
$$

$$
\begin{array}{l}
\mathbf{where} \\
\quad h\ op\ a\ b = i +\!\!+\ ii \\
\qquad\qquad\qquad \mathbf{where} \\
\qquad\qquad\qquad\quad i\ \ = [((a`op`) \circ f, p) \mid (f, p) \leftarrow \mathit{sites}\ b] \quad \text{-- right operand b first} \\
\qquad\qquad\qquad\quad ii = [((`op`b) \circ f, p) \mid (f, p) \leftarrow \mathit{sites}\ a]
\end{array}
$$

It should be noted that 'far-right' sites come first. This is just a mirror image of the normal situation, where the far-left argument comes first.

Now we define for any expression a list of the expressions to which it reduces in a single, possibly internal step, at exactly one site in the expression. This uses the function *tlr* to get top-level reducts.

$$
\begin{array}{l}
\mathit{reducts} :: E \rightarrow [E] \\
\mathit{reducts}\ a = [f\ a'' \mid (f, a') \leftarrow \mathit{sites}\ a, a'' \leftarrow \mathit{tlr}\ a']
\end{array}
$$

5.1 The tree of reduction sequences, and access to it

We need a structure to hold the reduction sequences from an expression. So-called 'rose' trees, with nodes labelled with expressions seem ideal.

$$
\mathbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ a\ [\mathit{Tree}\ a]\ \mathbf{deriving}\ \mathit{Show}
$$

We define a function which maps an expression to its tree of reduction sequences.

$$
\begin{array}{l}
\mathit{reductTree} :: E \rightarrow \mathit{Tree}\ E \\
\mathit{reductTree}\ e = \mathit{Node}\ e\ [\mathit{reductTree}\ e' \mid e' \leftarrow \mathit{reducts}\ e]
\end{array}
$$

The following function maps a tree to a sequence enumerating the nonempty sequences of node labels encountered on a path from the root of the tree to a (leaf) node without descendants.

$$
\begin{array}{l}
\mathit{branches} :: \mathit{Tree}\ a \rightarrow [[a]] \\
\mathit{branches}\ (\mathit{Node}\ a\ [\,]) = [[a]] \\
\mathit{branches}\ (\mathit{Node}\ a\ \mathit{ts}) = [a : b \mid t \leftarrow \mathit{ts}, b \leftarrow \mathit{branches}\ t]
\end{array}
$$

Putting things together, we can map an expression to a sequence of its reduction sequences. (Hence *rss*.)

$$
\begin{array}{l}
\mathit{rss} :: E \rightarrow [[E]] \\
\mathit{rss} = \mathit{branches} \circ \mathit{reductTree}
\end{array}
$$

The first 'canonical' reduction sequence in my enumeration seems usually to be the one that interest me.

6 Böhm's λogarythm

This code generating the λogarythmof an expression with respect to a variable name.

Böhm's combinators

$$
\begin{array}{l}
\mathit{cBohmA}\ a\ b = \textbf{let}\ g = a :\wedge: V\ \texttt{"\^{}"} :+: b :\wedge: V\ \texttt{"\^{}"}\ \textbf{in} \\
\quad \textbf{let}\ \mathit{curry}\ g = \mathit{cPair} :+: g :\wedge: \mathit{cK}\ \textbf{in} \quad \text{-- Bohm's original} \\
\quad \textbf{let}\ \mathit{curry}'\ g = \mathit{cPair} :\times: \mathit{cM} :\times: (g :\wedge: \mathit{cE}) \quad \text{-- another without additive apparatus} \\
\quad \textbf{in}\ \mathit{curry}'\ (a :\wedge: V\ \texttt{"\^{}"} :+: b :\wedge: V\ \texttt{"\^{}"}) \\
\mathit{cBohmE}\ a\ b = a :\times: \mathit{cPair} :+: b :\times: V\ \texttt{"\^{}"} \\
\mathit{cBohmM}\ a\ b = a :+: b \\
\mathit{cBohm0}\ a = \mathit{c0} :\times: (a :\wedge: \mathit{cE})
\end{array}
$$

These have the crucial properties

$$
\begin{array}{ll}
x \wedge \mathit{cBohmA}\ a\ b & = (x \wedge a) + (x \wedge b) \\
x \wedge \mathit{cBohmM}\ a\ b & = (x \wedge a) \times (x \wedge b) \\
x \wedge \mathit{cBohmE}\ a\ b & = (x \wedge a) \wedge x \wedge b \\
x \wedge \mathit{cBohm0}\ a & = a
\end{array}
$$

used in defining the logarithm.

This code can perhaps be slightly refined to keep the size of its logarithms down. The cases to look at are those where the variable occurs in just one of a pair of operands.

$$
\begin{array}{l}
\mathit{blog}\ v\ e \mid \neg\,(v \in \mathit{fvs}\ e) = \mathit{cBohm0}\ e \\
\mathit{blog}\ v\ e = \textbf{case}\ e\ \textbf{of} \\
\quad a :+: b \rightarrow\ \ \text{\{-cBohmA (blog v a) (blog v b) -\}} \\
\quad\quad \textbf{case}\ (v \in \mathit{fvs}\ a, v \in \mathit{fvs}\ b)\ \textbf{of} \\
\quad\quad\quad (\mathit{False}, \mathit{True}) \rightarrow (\mathit{blog}\ v\ b) :\times: (a :\wedge: \mathit{cA}) \\
\quad\quad\quad (\mathit{True}, \mathit{False}) \rightarrow (\mathit{blog}\ v\ a) :\times: (b :\wedge: \mathit{cA} :\wedge: \mathit{cC}) \\
\quad\quad\quad \_ \qquad\qquad\quad \rightarrow \mathit{cBohmA}\ (\mathit{blog}\ v\ a)\ (\mathit{blog}\ v\ b) \\
\quad a :\times: b \rightarrow \textbf{case}\ (v \in \mathit{fvs}\ a, v \in \mathit{fvs}\ b)\ \textbf{of} \\
\quad\quad\quad (\mathit{False}, \mathit{True}) \rightarrow (\mathit{blog}\ v\ b) :\times: (a :\wedge: \mathit{cM}) \\
\quad\quad\quad (\mathit{True}, \mathit{False}) \rightarrow (\mathit{blog}\ v\ a) :\times: (b :\wedge: \mathit{cB}) \\
\quad\quad\quad \_ \qquad\qquad\quad \rightarrow \mathit{cBohmM}\ (\mathit{blog}\ v\ a)\ (\mathit{blog}\ v\ b) \\
\quad a :\wedge: b \rightarrow \textbf{case}\ (v \in \mathit{fvs}\ a, v \in \mathit{fvs}\ b)\ \textbf{of} \\
\quad\quad\quad (\mathit{False}, \mathit{True}) \rightarrow \textbf{case}\ b\ \textbf{of} \\
\quad\quad\quad\quad V\ v \rightarrow a :\wedge: \mathit{cE} \\
\quad\quad\quad\quad \_ \quad\ \rightarrow \mathit{blog}\ v\ b :\times: (a :\wedge: \mathit{cE}) \\
\quad\quad\quad (\mathit{True}, \mathit{False}) \rightarrow \textbf{case}\ a\ \textbf{of} \\
\quad\quad\quad\quad V\ v \rightarrow b \\
\quad\quad\quad\quad \_ \quad\ \rightarrow \mathit{blog}\ v\ a :\times: b \\
\quad\quad\quad \_ \rightarrow \mathit{cBohmE}\ (\mathit{blog}\ v\ a)\ (\mathit{blog}\ v\ b) \\
\quad V\ \mathit{nm} \rightarrow \textbf{if}\ \mathit{nm} \equiv v\ \textbf{then}\ \mathit{c1}\ \textbf{else}\ \mathit{cBohm0}\ e
\end{array}
$$

The following function returns a list of all the variable names occurring in an expression. The list is returned in the order in which variables first occur in a depth-first scan.

```
fvs e = nodups $ f e []
  where f (V nm) = if nm ∈ ["0", "1", "^", "*", "+", "@", ",", "~", "."]
    then id else (nm:)
    f (a :∧: b)   = f a ∘ f b
    f (a :×: b)   = f a ∘ f b
    f (a :+: b)   = f a ∘ f b
    f (a :<>: b) = f b
```

A Bureaucracy and basic gadgetry

To save typing, names for all single-letter variables

```
(va, vb, vc, vd, ve, vf, vg, vh, vi, vj, vk, vl, vm, vn,
  vs, vt, vu, vv, vw, vx, vy, vz)
  = ( V "a", V "b", V "c", V "d", V "e", V "f", V "g", V "h", V "i", V "j", V "k", V "l", V "m", V
    , V "s", V "t", V "u", V "v", V "w", V "x", V "y", V "z")
```

We code a few useful numbers as expressions.

```
c2, c3, c4, c5, c6, c7, c8, c9, c10 :: E
c2  = c1 :+: c1
c3  = c2 :+: c1
c4  = c2 :∧: c2
c5  = c3 :+: c2
c6  = c3 :×: c2
c7  = c3 :+: c4
c8  = c2 :∧: c3
c9  = c3 :∧: c2
c10 = c2 :×: c5
```

c0 and *c1* have already been defined.

It is time we had an combinator for successor ($[+] \times 1^{[\wedge]}$, by the way).

```
cSuc :: E
cSuc = blog "x" (vx :+: c1)
cN :: Int → E   -- allows inputting numerals in decimal.
cN n = let x = c0 : [t :+: c1 | t ← x] in x !! n
cN_demo = "test $ vz :^: vf :^: cN 7"
```

{-dN :: N a -¿ Int – somehow allow decimal output, when possible. dN e = e (+1) 0 -}

B Displaying

B.1 Expressions

If one wants to investigate reduction sequences of arithmetical expressions by running this code, one needs to display them. To display expressions, we use the following code, which is slightly less noisy than the built in show instance. It should supress parentheses with associative operators. (I think everything is right associative: as with $\wedge$, so with the other operators.) I write the constant combinators in square brackets, which may be considered noisy.

I don't understand precedences very well. I think the following deals properly with associativity of + and $\times$, and their relative precedences (sums of products) but also with the non-associativity of $\wedge$. These nest to the right. The best I can say is that by some miracle it seems to work as I expect.

```
showE :: E → Int → String → String
showE (V "^") _ = ("[^]"++)
showE (V "*") _ = ("[*]"++)
showE (V "+") _ = ("[+]"++)
showE (V ",") _ = ("[,]"++)
showE (V "~") _ = ("[~]"++)
showE (V "&") _ = ("[&]"++)
showE (C "^") _ = ("[^]"++)
showE (C "*") _ = ("[*]"++)
showE (C "+") _ = ("[+]"++)
showE (C ",") _ = ("[,]"++)
showE (C "~") _ = ("[~]"++)
showE (C "&") _ = ("[&]"++)
showE (V str) _ = (str++)
showE (a :+: b) p = opp p 0 (showE a 0 ∘ (" + "++) ∘ showE b 0)
showE (a :×: b) p = opp p 2 (showE a 2 ∘ (" * "++) ∘ showE b 2)
showE (a :∧: b) p = opp p 4 (showE a 5 ∘ (" ^ "++) ∘ showE b 4)
  -- because the below are wierd operator, I make them noisy.
showE (a :<>: b) p = opp p 4 (showE a 5 ∘ (" <!> "++) ∘ showE b 4)
showE (a :~: b) p = opp p 4 (showE a 5 ∘ (" <~> "++) ∘ showE b 4)
showE (a :&: b) p = opp p 4 (showE a 5 ∘ (" <&> "++) ∘ showE b 4)
parenthesize f = showString "(" ∘ f ∘ showString ")"
opp p op = if p > op then parenthesize else id

instance Show E where showsPrec _ e = showE e 0
```

B.2 Trees and lists

Code to display a numbered list of showable things, throwing a line between entries.

$$
\begin{array}{l}
\textbf{newtype}\ \mathit{NList}\ a = \mathit{NList}\ [\,a\,] \\
\textbf{instance}\ \mathit{Show}\ a \Rightarrow \mathit{Show}\ (\mathit{NList}\ a)\ \textbf{where} \\
\quad \mathit{showsPrec}\ \_\ (\mathit{NList}\ \mathit{es}) = \\
\quad\quad (\mathit{composelist} \circ \mathit{commalist}\ (\texttt{'\textbackslash n'}{:}) \circ \mathit{map}\ \mathit{showline} \circ \mathit{enum})\ \mathit{es} \\
\quad\quad \textbf{where}\ \mathit{showline}\ (n, e) = \mathit{shows}\ n \circ \mathit{showString}\ \texttt{": "} \circ \mathit{shows}\ e
\end{array}
$$

B.2.1 General list and stream stuff

Code to pair each entry in a list/stream with its position.

$$
\begin{array}{l}
\mathit{enum} :: [\,a\,] \to [\,(\mathit{Int}, a)\,] \\
\mathit{enum} = \mathit{zip}\ [1\,..]
\end{array}
$$

Code to compose a finite list of endofunctions.

$$
\begin{array}{l}
\mathit{composelist} :: [\,a \to a\,] \to a \to a \\
\mathit{composelist} = \mathit{foldr}\ (\circ)\ \mathit{id}
\end{array}
$$

Code to insert a ‘comma’ at intervening positions in a stream.

$$
\begin{array}{l}
\mathit{commalist} :: a \to [\,a\,] \to [\,a\,] \\
\mathit{commalist}\ c\ (x : (\mathit{xs}'@(\_ : \_))) = x : c : \mathit{commalist}\ c\ \mathit{xs}' \\
\mathit{commalist}\ c\ \mathit{xs} = \mathit{xs}
\end{array}
$$

Remove duplicates from a list/stream. The order in which entries are first encountered is preserved in the output.

$$
\begin{array}{l}
\mathit{nodups} :: \mathit{Eq}\ a \Rightarrow [\,a\,] \to [\,a\,] \\
\mathit{nodups}\ [\,] = [\,] \\
\mathit{nodups}\ (x : \mathit{xs}) = x : \mathit{nodups}\ (\mathit{filter}\ (\not\equiv x)\ \mathit{xs})
\end{array}
$$

B.3 Some top-level commands

The first reduction sequence. This is by far the most useful. One might type something like

$$
\mathit{test}\ \$\ \mathit{vu} :\wedge: \mathit{vz} :\wedge: \mathit{vy} :\wedge: \mathit{vx} :\wedge: \mathit{cS}
$$

$$
\begin{array}{l}
\mathit{test} :: E \to \mathit{NList}\ E \\
\mathit{test} = \mathit{NList} \circ \mathit{head} \circ \mathit{rss}
\end{array}
$$

The normal form. This is occasionally useful when evaluation will obviously terminate. Only the normal form is displayed.

$$
\mathit{eval}\ \$\ \mathit{vz} :\wedge: \mathit{vy} :\wedge: \mathit{vx} :\wedge: \mathit{cS}
$$

Display the n’th reduction sequence in a reduction tree.

```
nth_rs :: Int → E → NList E
nth_rs n = NList ∘ (!!n) ∘ rss
```

Display an entire *NTree a*. Uses indentation in an attempt to make the branching structure of the tree visible. (Actually, this is almost entirely useless, except for very small expressions)

```
newtype NTree a = NTree (Tree a)
instance Show a ⇒ Show (NTree a) where
  showsPrec p (NTree t) =
    f id (1, t) where    -- f :: Show a ⇒ ShowS → (Int, Tree a) → ShowS
      f pr (n, Node a ts)
        = (pr                        -- emit indentation
          ∘ showString "["
          ∘ shows n
          ∘ showString "] "          -- child number
          ∘ shows a                  -- node label
          ∘ showString "\n"
          ∘ (composelist
            ∘ map (f (pr ∘ showString "! "))
            ∘ enum) ts)
```

We can try something like **let** $s = Ntree \circ reductTree$ **in** $s\ (va :\wedge: cS)$.

Some basic stats on reduction sequence length. The number of reduction sequences, and the extreme values of their lengths. Be warned, this can take a very long time to finish on even quite small examples.

```
stats_rss :: E → (Int, (Int, Int))
stats_rss e = let (b0 : bs) = map length (rss e)
    in (length (b0 : bs), (foldr min b0 bs, foldr max b0 bs))
data DisplayStats = DisplayStats (Int, (Int, Int))
instance Show (DisplayStats) where
  showsPrec _ (DisplayStats (n, (l, h)))
    = ("There are "++) ∘ shows n ∘ (" reduction sequences"++)
      ∘ (", varying in length between "++)
      ∘ shows l ∘ (" and "++) ∘ shows h
```

```
  -- check that all reduction sequences terminate with the same expression
nf :: E → [E]
nf = map last ∘ rss
  -- might be useful
  -- find the first suffix of a list that begins with a change
fd :: [E] → Maybe [E]   -- first difference
fd [] = Nothing
fd [x] = Nothing
fd (x : xs@(y: _)) | x ≡ y = fd xs
```

$fd\ z@(x : xs@(y : \_)) \mid x \not\equiv y = Just\ z$
-- do not try this on a constant infinite stream.

B.4 IO

We would like to run these programs. My suggestion is to think of the programs as stream processors. The program runs in a state-space consisting of an accumulator register, and an unconsumed stream. Each cycle of the program consumes some initial segment of the input stream, and performs a corresponding action on the accumulator. (This might be to add something to it, or multiply it by something.)

The stream of output produced by the program is then a potentially infinite history of successive accumulator contents.

C Examples

In this section, we encode some naturally occurring combinators as expressions.

C.1 CBKIWSS'

The combinators C, B, K, I and W can be encoded as follows in our calculus.

```
cC, cB, cK, cI, cI', cW, c0 :: E
cC  = V "*" :×: V "^" :∧: V "*"                  -- M to one plus E to the E
cB  = V "^" :×: V "*" :∧: V "*"                  -- M to the C
cK  = V "^" :×: V "0" :∧: V "*"                  -- 0 to the C
cI  = V "@" :∧: V "0"
cI' = V "^" :×: V "^" :∧: V "*"                  -- E to the C
cW  = V "^" :×: (V "^" :+: V "^") :∧: V "*"      -- twice E to the cC
```

The ‘real word’ versions:

```
combC        = (×) × (∧) ∧ (×)           -- flip, transpose.
combB        = (∧) × (×) ∧ (×)           -- (∘), composition. (×) ∧ combC
combI        = naughtiness ∧ ()          -- id. also (∧) × (∧) ∧ (×), inter alia
combK        = (∧) × () ∧ (×)            -- const. () ∧ combC
combW        = (∧) × ((∧) + (∧)) ∧ (×)   -- diagonalisation. ((∧) + (∧)) ∧ combC
naughtiness  = error "Naughty!"
```

As for S, after a little playing around, another combinator emerges. This is S', where $S\ a\ b$ (the normal S combinator) is $W\ (S'\ a\ b)$.

$$S'\ a\ b\ c1\ c2 = a\ c1\ (b\ c2)$$

It turns out that

$S' = (\times) \times ((\times)\times)$

In particular, we have the following remarkable equations:

$$
\begin{array}{lll}
S & = S' \times (\times) \times (W \wedge (\wedge)) & \\
S' & = S' \; (\times) & \\
C & = S' \; (\wedge) & \\
B & = S' \; (\wedge) \; (\times) & = (\times) \wedge C \\
I & = S' \; (\wedge) \; (\wedge) & = (\wedge) \wedge C \\
K & = S' \; (\wedge) \; () & = () \wedge C \\
W & = S' \; (\wedge) \; ((\wedge) + (\wedge)) & = ((\wedge) + (\wedge)) \wedge C \\
S' \; 1 & = (\times) &
\end{array}
$$

One can define the S' and S combinators as follows:

$$
\begin{array}{l}
combS' :: (a \to b \to c) \to (a1 \to b) \to a \to a1 \to c \\
combS :: (a \to b \to c) \to (a \to b) \to a \to c \\
combS' = \mathbf{let}\ x = (\times)\ \mathbf{in}\ x \times (x \wedge x) \\
combS = combS' \times (\times) \times (combW \wedge)
\end{array}
$$

The following expressions code the S and S' combinators combinators.

$$
\begin{array}{ll}
cS, cS' :: E & \\
cS & = cS' :\times: cW :\wedge: cB \\
cS' & = cM :\times: (cM :\wedge: cM)
\end{array}
$$

-- the following is an example for checking that the logarithm apparatus is working.
-- It is a variant of the S combinator that should evaluate in the correct way.

$$
\begin{array}{ll}
cSalt :: E & \\
cSalt & = blog\ \texttt{"y"}\ (blog\ \texttt{"x"}\ ((vx :\times: cPair) :+: (vy :\times: cE)))
\end{array}
$$

Try $test \ \$ \ vy :\wedge: vx :\wedge: vb :\wedge: va :\wedge: cSalt$. It needs two extra variables.

C.2 Sestoft's examples

There is a systematic way of encoding data structures (pairs, tuples, whatnot) in the λ-calculus, sometimes called Church-encoding.

Here are some examples in the list of predefined constants in Sestoft's Lambda calculus reduction workbench at http://raspi.itu.dk/cgi-bin/lamreduce?action=print+abbreviations

The first line shows the definition, the remaining lines show the reduction to arithmetic form.

$$
\begin{array}{ll}
pair\ x\ y\ z & = z\ x\ y \\
& = y \wedge x \wedge z \\
& = (x \wedge z) \wedge (y \wedge) \\
& = (z \wedge (x \wedge)) \wedge (y \wedge) \\
pair\ x\ y & = (x \wedge) \times (y \wedge) \\
& = (y \wedge) \wedge ((x \wedge) \times)
\end{array}
$$

$$
\begin{aligned}
\mathit{pair}\ x &= (\wedge) \times ((x\wedge)\times) \\
&= ((x\wedge)\times) \wedge ((\wedge)\times) \\
&= ((x \wedge (\wedge)) \wedge (\times)) \wedge ((\wedge)\times) \\
\mathit{pair} &= (\wedge) \times (\times) \times ((\wedge)\times)
\end{aligned}
$$

$cPair = V$ `"^"` $:\times:\ V$ `"*"` $:\times:\ (V$ `"^"` $:\wedge:\ V$ `"*"`$)$

Closely related to pairing is the Curry combinator, which satisfies $f \wedge cCurry\ x\ y = f\ (x, y)$. The following are alternate versions of this combinator.

$$
\begin{aligned}
&cCurry\ = cK :\times: (cPair :\wedge: cA) \\
&cCurry' = cB :\times: (cPair :\wedge: cM) \\
&cCurry\_demo = (vz :\wedge: (vy :\wedge: vx :\wedge: cPair) :\wedge: vf) : \& : (vz :\wedge: vy :\wedge: vx :\wedge: vf :\wedge: cCurry)
\end{aligned}
$$

Try $test\ \$\ cK :\wedge: cCurry\_demo$, then $test\ \$\ c0 :\wedge: cCurry\_demo$.

$$
\begin{aligned}
\mathit{tru}\ x\ y &= x \\
&= x \wedge y \wedge () \\
&= (y \wedge ()) \wedge (x\wedge) \\
\mathit{tru}\ x &= () \times (x\wedge) \\
&= (x\wedge) \wedge (()\times) \\
\mathit{tru} &= (\wedge) \times (()\times) \\
&= () \wedge C
\end{aligned}
$$

$$
\begin{aligned}
\mathit{fal}\ x\ y &= y \\
&= y \wedge x \wedge () \\
\mathit{fal} &= ()
\end{aligned}
$$

$cFalse = c0$
$cTrue = c0 :\wedge: cC$ -- cK
$cNot = cC$

$$
\begin{aligned}
\mathit{fst}\ p &= p\ \mathit{tru} \\
&= \mathit{tru} \wedge p \\
&= p \wedge (\mathit{tru}\wedge) \\
\mathit{fst} &= (\mathit{tru}\wedge) \\
\mathit{snd} &= (\mathit{fal}\wedge)
\end{aligned}
$$

$cFst = cTrue :\wedge: cE$
$cSnd = cFalse :\wedge: cE$

$$
\begin{aligned}
\mathit{iszero}\ n &= n\ (K\ \mathit{fal})\ \mathit{tru} \\
&= n\ (() \times (\mathit{fal}\wedge))\ \mathit{tru}
\end{aligned}
$$

$$
\begin{aligned}
&= tru \wedge (() \times (fal\wedge)) \wedge n \\
&= (n \wedge ((() \times (fal\wedge))\wedge)) \wedge (tru\wedge) \\
iszero &= ((() \times (fal\wedge))\wedge) \times (tru\wedge)
\end{aligned}
$$

$$cIszero = cTrue :\wedge: (cFalse :\wedge: cK) :\wedge: cPair$$

C.3 Tupling, projections

We use the usual notation (a, b) for pairs. In general

$$(a1, ..., ak) = (a1\wedge) \times ... \times (ak\wedge)$$

and the projection operators π_i^k have the form

$$(K \wedge i\ (K \wedge j)) \wedge \textbf{where}\ i + j + 1 = k$$

In fact the binary projections are defined using the booleans, and other projections are defined using more general selector terms such as $\lambda x1 \,.\,.\, xn \to xi$. This is done by applying $(\wedge)$ to the selector.

$$\lambda p \to p\ sel = (sel\wedge)$$

The booleans are $K = (K \wedge 0)\ (K \wedge 1)$ and $0 = (K \wedge 1)\ (K \wedge 0)$. Selecting the i'th element of a stack with $i + j + 1$ elements is $(K \wedge i)\ (K \wedge j)$.

It may be interesting to remark that

$$(a, a) = W\ pr\ a = (a\wedge) \times (a\wedge) = a \wedge ((\wedge) + (\wedge))$$

So that $pr \wedge W = (\wedge) + (\wedge) = W \wedge C$

TODO: code some expressions

C.4 Fixpoint operators

Among endless variations, two fixed-point combinators stand out, Curry's and Turing's. Both their fixed point combinators use self application.

This of course banishes us from the realm of combinators that Haskell can type, but what the heck. There is syntax for it. We diagonalise exponentiation.

$$
\begin{aligned}
&cSap :: E \\
&cSap = cE :\wedge: cW
\end{aligned}
$$

We call the self application combinator sap.

$$
\begin{aligned}
sap\ x &= x\ x = W\ (\wedge)\ x = W\ 1\ x \\
So\ sap &= (\wedge) \wedge W = 1 \wedge W
\end{aligned}
$$

We call Curry's combinator simply Y_C.

$$
\begin{aligned}
f \wedge Y &= sap\ (sap \times f) \\
Y &= (sap\times) \times sap \\
&= sap \wedge ((\times) + 1)
\end{aligned}
$$

Y can thus be seen as applying the successor of multiplication to the value sap.

$$cY\_C = cSap :\wedge: cM :\wedge: cSuc$$

Turing's combinator is $T \wedge T$ where $Txy = y(xxy)$.

$$
\begin{aligned}
&T\ x\ y &&= y\ (x\ x\ y) = y\ (sap\ x\ y) = y\ ((sap \wedge C)\ y\ x) \\
&(T \wedge C)\ y\ x &&= y\ ((sap \wedge C)\ y\ x) = (y \circ (sap \wedge C)\ y)\ x \\
&(T \wedge C)\ y &&= ((sap \wedge C)\ y) \times y \\
&(T \wedge C) &&= (sap \wedge C) + 1 \\
&T &&= ((sap \wedge C) + 1) \wedge C \\
& &&= sap \wedge (C \times (+1) \times C)
\end{aligned}
$$

T can thus be seen as applying a kind of dual (with respect to the involution C) of the successor operator to the value sap.

Some expressions for Turing's semi-Y, and his Y.

$$
\begin{aligned}
&cT = cSap :\wedge: (cC :\times: cSuc :\times: cC) \\
&cY\_T = cT :\wedge: cSap
\end{aligned}
$$

Be careful when evaluating these things!

C.5 Rotation combinators

The following linear combinator $combR$ 'rotates' 3 arguments.

$$
\begin{aligned}
&combR\ :: a \to (b \to a \to c) \to b \to c \\
&combR\ = (\wedge) \times (((\times) \times ((\wedge)\times))\times) \\
&combR' = (\wedge) \times (\wedge) \times ((\times)\times)
\end{aligned}
$$

Some such operation is often provided by the instruction set of a 'stack machine', to rotate the top three entries on the stack. It can be seen as a natural extension of the operation that flips (that is, rotates) the top two entries.

It can be encoded as follows:

$$
\begin{aligned}
&cR, cR\_var :: E \\
&cR = cC :\wedge: cC \\
&cR\_var \quad = (V\ \texttt{"\^{}"}) :\times: (V\ \texttt{"\^{}"}) :\times: (V\ \texttt{"*"}) :\wedge: (V\ \texttt{"*"}) \quad \text{-- a variant} \\
&cL :: E \\
&cL = cPair \\
&cR\_demo = test\ \$\ vc :\wedge: vb :\wedge: va :\wedge: cR \\
&cL\_demo\ = test\ \$\ vc :\wedge: vb :\wedge: va :\wedge: cL
\end{aligned}
$$

It has a cousin, that rotates in the other direction. This is actually the pairing combinator.

It so happens that the cC combinator and the cR are each definable from the other.

$$cC' = cR :\wedge: cR :\wedge: cR$$
$$cR' = cC :\wedge: cC$$

To be a little frivolous, this gives us a way of churning out endless variants of the combinators *combR* and *flip*.

$$\mathit{flip}', \mathit{flip}'' :: (a \to b \to c) \to b \to a \to c$$
$$\mathit{flip}' = \mathit{flip}\ \mathit{flip}\ (\mathit{flip}\ \mathit{flip})\ (\mathit{flip}\ \mathit{flip})$$
$$\mathit{flip}'' = \mathit{flip}'\ \mathit{flip}'\ (\mathit{flip}'\ \mathit{flip}')\ (\mathit{flip}'\ \mathit{flip}')$$

C.6 Continuation transform

The function $\wedge$ which takes a to $a\wedge$ pops up everywhere when playing with arithmetical combinators. It provides the basic means of interchanging the positions of variables: $a \wedge b = b \wedge (a\wedge) = b \wedge a \wedge (\wedge)$.

On the topic of the continuation transform, for a fixed result type $R = ()$, the type-transformer CT

$$\textbf{type}\ CT\ a = (a \to ()) \to ()$$

is the well known continuation monad. The action on maps is

$$map\_CT :: (a \to b) \to CT\ a \to CT\ b$$
$$map\_CT\ f\ cta\ k = cta\ (k \circ f)$$
$$map\_CT'1\ f\ cta\ k = cta\ (f \times k)$$
$$map\_CT'2\ f\ cta = (f\times) \times cta$$
$$map\_CT'\ f = ((f\times)\times)$$

$$cmap\_CT :: E$$
$$cmap\_CT = cM :\times: cM$$

The unit *return* and multiplication *join* of this monad have simple arithmetical expressions.

$$\begin{array}{lll} return & :: a \to CT\ a & \\ join & :: CT\ (CT\ a) \to CT\ a & \\ return\ a\ b & = a \wedge b & \text{-- ie. } return = (\wedge) \\ f\ \text{`}join\text{`}\ s & = f\ (return\ s) & \text{-- ie. } join = ((\wedge)\times) \end{array}$$

$$cRet, cMu, cMap :: E$$
$$cRet = cE$$
$$cMu = cE :\wedge: cM$$
$$cMap = blog\ \texttt{"m"}\ (blog\ \texttt{"c"}\ (blog\ \texttt{"k"}\ (vm :\times: vk) :\wedge: vc))$$

We can simply define the bind operator from join and map.

$cBind :: E$
$cBind = \textbf{let}\ arg = vm :\wedge: vc :\wedge: cMap\ \textbf{in}\ blog\ \texttt{"m"}\ (blog\ \texttt{"c"}\ (arg :\wedge: cMu))$

You may be interested in fancy control operators (like 'abort'), and flirtations with classical logic. The following may reduce your cravings.

Peirce's law: $((a \to b) \to a) \to a$ is interesting because it is a formula of minimal logic. It involves only the arrow, and not 0. *Yet* you can prove excluded middle *aornota* from it, where negation is relativised to a generic type r, as $\neg\ a = a \to r$. So when defining something, one has unrestricted access to these two cases.

Peirce's law postulates the existence of an algebra for a certain monad, called 'the Peirce monad' by Escardo&co.

When 'true' 0 (including efq) and hence true negation is added, to minimal logic, Peirce's law implies not just excluded middle, but full classical logic with involutive negation, *ie.* $\sim(\sim A) = A$.

To suppose the negation of Peirce's law leads to an absurdity. (We don't need efq for this.)

$$\begin{aligned}
\sim Peirce &= \sim a \mathbin{\&} ((a \to b) \to a) \\
&\Rightarrow \sim a \mathbin{\&} \sim(a \to b) \\
&= \sim a \mathbin{\&} \sim b \mathbin{\&} a
\end{aligned}$$

We cannot hope to prove Peirce's law, but we might expect to prove it's transform by the continuation monad $((a \to CT\ b) \to CT\ a) \to CT\ a$, in which all the arrows $a \to b$ have been turned into Kleisli arrows $a \to CT\ b$. Or maybe even $((a \to b) \to CT\ a) \to CT\ a$. We can ask what such a thing would look like expressed with arithmetical combinators. I once made an effort to do this, and got the following result (though it is pretty certain there are some errors here):

$$(((\times) \times ((\wedge) \times 0^{(\times)})^{(\wedge)})^{(\times) \times (\wedge)}) \times ((\wedge) + (\wedge))^{(\times)}$$

Well, if that's an arithmetical expression of classical logic, it's neither very enlightening or beguiling! One can however see some additive features here, namely $(\wedge) + (\wedge)$ and 0. I find that reassuring.

One can ask the same questions with respect to the Peirce monad.

The monadic apparatus can be encoded as follows

$$\begin{aligned}
&ct && :: E \to E \\
&ct\ a && = a :\wedge: V\ \texttt{"\^{}"} \quad \text{-- unit} \\
&cb && :: E \to E \to E \\
&cb\ m\ f && = V\ \texttt{"\^{}"} :\times: (f :\wedge: V\ \texttt{"*"}) :\times: m \quad \text{-- bind}
\end{aligned}$$

$$\begin{aligned}
&cCTret, cCTjoin, cCTbind :: E \\
&cCTret \quad = V\ \texttt{"\^{}"} \\
&cCTjoin = cCTret :\wedge: V\ \texttt{"*"} \\
&cCTbind = blog\ \texttt{"m"}\ (blog\ \texttt{"f"}\ (cb\ (V\ \texttt{"m"})\ (V\ \texttt{"f"})))
\end{aligned}$$

It may be interesting to point out that $(a, b) = a^{[\wedge]} \times b^{[\wedge]} = \eta a \times \eta b$ where η is the unit of CT.

C.7 Peano numerals

Oleg Kiselyov was once and may still be interested in what I think he calls or once called ‘p-numerals’.These are (so to speak) related to primitive recursion as the Church numerals are related to iteration. So the successor of n is not

$$\lambda f, z \to f\ (n\ f\ z)$$

as it is with Church numerals, but rather

$$\lambda f, z \to f\ n\ (n\ f\ z)$$

I have heard other people than Oleg express an interest in this encoding. It’s not un-natural.

So, letting the variable n vary over p-numerals, one has

$$n\ b\ a = b\ (n-1)\ (b\ (n-2)\ ...\ (b\ 1\ (b\ \dot{()}\ a))...)$$

Using the combinators of this paper, one can derive

$$\begin{aligned} n\ b &= b\ \dot{()} \times b\ 1 \times ... \times b\ (n-2) \times b\ (n-1) \\ &= b \wedge (\dot{()}\wedge) \times b \wedge (1\wedge) \times ... \times b \wedge ((n-2)\wedge) \times b \wedge ((n-1)\wedge) \\ &= b \wedge ((\dot{()}\wedge) + (1\wedge) + ... + ((n-2)\wedge) + ((n-1)\wedge)) \end{aligned}$$

By exponentiality (ζ),

$$n = (\dot{()}\wedge) + (1\wedge) + ... + ((n-2)\wedge) + ((n-1)\wedge)$$

In fact, if $Osucc$ is Oleg’s successor, we have $Osucc\ n = n + (n\wedge)$.

$$\begin{aligned} &\dot{()}\ \_p = \dot{()} \\ &1\ \_p = (\dot{()}\wedge) \\ &2\ \_p = (\dot{()}\wedge) + ((\dot{()}\wedge)\wedge) \\ &3\ \_p = (\dot{()}\wedge) + ((\dot{()}\wedge)\wedge) + (((\dot{()}\wedge) + ((\dot{()}\wedge)\wedge))\wedge) \\ &... \end{aligned}$$

One may be reminded here of von-Neumann’s representation for ordinals, which has $n \mapsto n \cup \{n\}$.for its successor operation, and the empty set $\{\}$ for its origin.

$$\begin{array}{rcl} 0 & = & \{\} \\ 1 & = & \{\{\}\} \\ 2 & = & \{\{\}, \{\{\}\}\} \\ 3 & = & \{\{\}, \{\{\}\}, \{\{\}, \{\{\}\}\}\} \\ & & ... \end{array}$$

Clearly the operation of raising to the power of exponentiation (that takes n to $(n\wedge)$) plays the role of the singleton operation $n \mapsto \{n\}$.

$$\begin{aligned} &cOsuc :: E \to E \\ &cOsuc\ e = e :+: (e :\wedge: V\ \texttt{"\^{}"}) \end{aligned}$$

$cOzero :: E$
$cOzero = V$ "0"
$cO :: Int \to E$ -- allows inputting numerals in decimal.
$cO\ n = \textbf{let}\ x = cOzero : [\,cOsuc\ t \mid t \leftarrow x\,]\ \textbf{in}\ x\ !!\ n$

C.8 sgbar, *&co.*

sgbar, or exponentiation to base zero, is the function which is 1 at 0, and 0 everywhere else. In other words, it is the characteristic function of the zero numbers.

$sgbar = ((){\wedge})$
$cSgbar = c0 :\wedge: cE$
$sgbar' :: EE\ (N\ a)$
$sgbar'\ n\ s\ z = n\ (const\ z)\ (s\ z)$
$cSgbar' = \textbf{let}\ v1 = vz :\wedge: vs$
$\quad ef = vz :\wedge: cK$
$\quad \textbf{in}\ (blog$ "n" $(blog$ "s" $(blog$ "z" $(v1 :\wedge: ef :\wedge: vn))))$

Using *sgbar*, we can define *sg*, which is 0 at 0, and 1 elsewhere (the sign function, or the characteristic function of the non-zero numbers).

$sg = sgbar \times sgbar$
$sg' :: EE\ (N\ a)$
$sg'\ n\ s\ z = n\ (const\ (s\ z))\ z$
$cSg = cSgbar :\times: cSgbar$
$cSg' = \textbf{let}\ v1 = vz :\wedge: vs$
$\quad ef = v1 :\wedge: cK$
$\quad \textbf{in}\ (blog$ "n" $(blog$ "s" $(blog$ "z" $(vz :\wedge: ef :\wedge: vn))))$

It may be clearer to write it $sg\ a = () \wedge () \wedge a$. Think of double negation.

Using *sg* and *sgbar*, we can implement a form of boolean conditionals. *IF* $b = 0$ *THEN* a *ELSE* c can be defined as $a \times sg\ (b) + c \times sgbar\ (b)$.

In fact we have forms of definition by finite cases.

C.9 A simple parser

Is it even worth thinking about this? The interpreter gives a fine language for defining expressions, using let expressions, etc.

Something changed in ghc 7.10.2 making it a fuss to write simple parsers. Applicative is bound up with monads, and they have stolen $< * >$. If I hide that, I hide monads, and can't use do notation.

-- PARSERS.
-- t is the token type, v the parsed value.
newtype $Parser\ t\ v = Parser\ \{prun :: [t] \to [(v, [t])]\}$

```
sat :: (t → Bool) → Parser t t
sat p = Parser f where f (t : ts) = if p t then [(t, ts)] else []
           f []                    = []
lit :: Eq t ⇒ t → Parser t t
lit t  = sat (≡ t)
  -- composes a sequence of N parsers that return things of the same type A
  -- into a parser that returns a list in A* of length N.
fby :: Parser t a → Parser t [a] → Parser t [a]
fby p q
  = Parser (λs → [((v : vs), s'') | (v, s') ← prun p s, (vs, s'') ← prun q s'])
fby2 :: Parser t a → Parser t b → (a → b → c) → Parser t c
fby2 p q f
  = Parser (λs →
        [(f v v', s'') | (v, s') ← prun p s
                       , (v', s'') ← prun q s'])
grdl :: Parser t a → Parser t b → Parser t b
grdr :: Parser t a → Parser t b → Parser t a
grdl p q
  = Parser (λs → [(b, s'') | (_, s') ← prun p s, (b, s'') ← prun q s'])
grdr p q
  = Parser (λs → [(a, s'') | (a, s') ← prun p s, (_, s'') ← prun q s'])
paren p
  = Parser (λs → [(a, s''') | (_, s') ← prun (lit Lpar) s
    , (a, s'')                   ← prun p s'
    , (_, s''') ← prun (lit Rpar) s''])
alt :: Parser t a → Parser t a → Parser t a
alt p q = Parser (λs → prun p s ++ prun q s)
alts :: [Parser t a] → Parser t a
alts ps = Parser (λs → concat [prun p s | p ← ps])
empty :: Parser t [a]
empty = Parser (λs → [([], s)])
rep, rep1 :: Parser t a → Parser t [a]
rep p = rep1 p `alt` empty
rep1 p = p `fby` rep p
repsep :: Parser t a → Parser t b → Parser t [a]
repsep p sep = p `fby` rep (sep `grdl` p)
  -- SCANNER.
  -- turns a stream of characters into a stream of tokens.
  -- import Data.Char
is_alphabetic c = 'a' ⩽ (c :: Char) ∧ c ⩽ 'z' ∨ 'A' ⩽ c ∧ c ⩽ 'Z'
is_digit    c = '0' ⩽ (c :: Char) ∧ c ⩽ '9'
is_idchar c = c ∈ "_." ∨ is_alphabetic c ∨ is_digit c
is_space   c = c ≡ ' '
```

```
is_par    c = c ∈ "()"
is_symch c = ¬ (is_par c ∨ is_space c)
data Tok = Id String | Num Int |
  Sym String | Lpar | Rpar
  deriving (Show, Eq)
tokens :: String → [Tok]
tokens [ ] = [ ]
tokens (c : cs) | isSpace c = tokens cs
tokens (inp@('(' : cs))
   = Lpar : tokens cs
tokens (inp@(')' : cs))
   = Rpar : tokens cs
tokens (c : cs) | is_alphabetic c
   = id_t (c:) cs where
     id_t b [ ] = [Id (b [ ])]
     id_t b (c : cs) | is_idchar c = id_t (b ∘ (c:)) cs
     id_t b inp             = Id (b [ ]) : tokens inp
tokens (c : cs)
   = id_t (c:) cs where
     id_t b [ ] = [Sym (b [ ])]
     id_t b (c : cs) | is_symch c = id_t (b ∘ (c:)) cs
     id_t b inp = Sym (b [ ]) : tokens inp
  -- GRAMMAR
variable :: Parser Tok E
variable = Parser p where
  p (Id st : ts) = [(V st, ts)]
  p _ = [ ]
constant :: Parser Tok E
constant = Parser p where
  p (Sym st : ts) = [(V st, ts)]
  p _ = [ ]
atomic       = variable `alt` constant `alt` paren expression
additive     = Parser (λs → [(fo (:+: ) x xs, s')
                        | ((x : xs), s') ←
                          prun (repsep multiplicative (lit (Sym "+"))) s])
multiplicative = Parser (λs → [(fo (:×: ) x xs, s')
                        | ((x : xs), s') ←
                          prun (repsep exponential (lit (Sym "*"))) s])
exponential = Parser (λs → [(fo (:∧: ) x xs, s')
                        | ((x : xs), s') ←
                          prun (repsep atomic (lit (Sym "^"))) s])
expression   = additive
discard      = Parser (λs → [(fo (:<>:) x xs, s')
                        | ((x : xs), s') ←
```

```
                      prun (repsep atomic
                         ((lit (Sym "!"))
                            `alt` lit (Sym "<>"))) s
                    ])
```

-- foldr1 ?

$fo\ op\ fst\ [\,] = fst$
$fo\ op\ fst\ (x : xs) = fst\ `op`\ fo\ op\ x\ xs$

-- instance Read E where
-- readsPrec d = prun expression . tokens

$rdexp :: String \to E$
$rdexp = fst \circ head \circ prun\ expression \circ tokens$

D Benedicto benedicatur

Once used, with *test*, to demo some evaluations.

$demo1Add = \mathbf{let}\ d = va :\!+\!: vb$
$\quad \mathbf{in}\ vc :\!\wedge\!: d$
$demo1Zero = \mathbf{let}\ d = V\ \texttt{"0"}\ \mathbf{in}\ va :\!\wedge\!: d$
$demo1Mul = \mathbf{let}\ d = va :\!\times\!: vb\ \mathbf{in}\ vc :\!\wedge\!: d$
$demo1One = \mathbf{let}\ d = V\ \texttt{"1"}\ \mathbf{in}\ va :\!\wedge\!: d$

-- show that the logarithm of an exponential behaves as expected

$demoExp = \mathbf{let}\ d = (va :\!\wedge\!: cPair) :\!\times\!: (vb :\!\wedge\!: V\ \texttt{"\^{}"})$
$\quad \mathbf{in}\ vc :\!\wedge\!: d$
$demoExp' = \mathbf{let}\ d = (va :\!\times\!: cPair) :\!+\!: (vb :\!\times\!: V\ \texttt{"\^{}"})$
$\quad \mathbf{in}\ vd :\!\wedge\!: vc :\!\wedge\!: d$

-- two equivalent codings

$demoAdd = \mathbf{let}\ c = (va :\!\wedge\!: V\ \texttt{"\^{}"}) :\!+\!: (vb :\!\wedge\!: V\ \texttt{"\^{}"})$
$\quad d = cPair :\!\times\!: V\ \texttt{"*"} :\!\times\!: (c :\!\wedge\!: V\ \texttt{"\^{}"})$
$\quad \mathbf{in}\ ve :\!\wedge\!: vd :\!\wedge\!: vc :\!\wedge\!: d$
$demoAdd' = \mathbf{let}\ c = (va :\!\wedge\!: V\ \texttt{"\^{}"}) :\!+\!: (vb :\!\wedge\!: V\ \texttt{"\^{}"})$
$\quad d = cPair :\!+\!: (c :\!\wedge\!: cK)$
$\quad \mathbf{in}\ ve :\!\wedge\!: vd :\!\wedge\!: vc :\!\wedge\!: d$

$demoNaught = \mathbf{let}\ d = V\ \texttt{"0"} :\!\times\!: V\ \texttt{"0"} :\!\wedge\!: V\ \texttt{"\^{}"}\ \mathbf{in}\ d$