The AMEN combinators and their reduction

If one represents natural numbers by Church numerals, there are natural definitions of the arithmetic operators `(+)`, `(*)`, and `(^)` which make sense as definitions of general combinators. It turns out that these combinators are combinatorially complete. This was pointed out in print by Böhm, but may have been previously noticed by Fitch.

In the arithmetical notation arising from this combinatory completeness, the symbol `(^)` for exponentiation becomes the transpose of the normal application operator (usually represented by juxtaposition). It becomes 'backwards application', which is sometimes used in algebra. The symbol `(*)` becomes the transpose of the normal composition `(.)` - ie. 'forwards composition' sometimes written with a semi-colon. The addition operator ia a pointwise 'lifted' version of `(*)`, and is sometimes useful. Interestingly, the fragment without the addition combinator characterises the linear lambda calculus.

In this file I implement a rewriting calculus for arithmetical expressions built up with constructors for addition, multiplication, exponentiation and naught. I've found it useful for experimentation (ie. playing around) with 'arithmetical' versions of standard combinators. The end of the file consists mostly of debris from such experiments.

combinatorial completeness

It is one of the very nice features of Haskell that one can take over and redefine symbols such as for the arithmetical operations `(+)`, `(*)`, and `(^)`. Two cheers for the 'hiding' keyword! However, it doesn't seem that we can take over '`0`'. This is a pity, as it is the only other symbol I need. Instead, for zero, nought, or naught, I use the infix symbol '`<>`', in the form `(<>)`. (It looks a little bit like an '`0`'.)

```
module Main where
import Prelude hiding ((*),(^),(+))

infixr 8  ^
infixr 7  *
infixr 6  +
```

The calculus of iterative exponents can be summarised in the following Haskell definitions. The second and third definitions arise very naturally in connection with exponential notation for the iterates of a function, while the fourth is only a little more recondite. The first can be regarded as introducing the exponential as an alternative notation for application, with the arguments flipped.

```
a ^ b  = b a
a * b  = \c -> (c ^ a) ^ b
a + b  = \c -> (c ^ a) * (c ^ b)
a <> b = b
```

An amusing phenomenon is that `(+),(*),(^),(<>)` are combinatorially complete. To convince oneself of this it is enough to define the 'BWICK' combinators.

```
combC = (*) * (^) ^ (*)              -- called "flip" by fp'ers
combB = (^) * (*) ^ (*)              -- (*) ^ combC, i.e. composition (.)
combI = evil ^ (<>)
combK = (^) * (<>) ^ (*)             -- (<>) ^ combC
combW = (^) * ((^) + (^)) ^ (*)      -- ((^) + (^)) ^ combC
evil  = error"The unthinkable has occurred."
```

You may like to see derivations of some of these. The key thing is to start with the `C` (argument swapping, or) transpose combinator.

```
   C a b c
 = { def }
   a c b
 = { re-express using exponentiation }
   b ^ (c ^ a)
 = { def of (^) }
   (c ^ a) ^ b ^ (^)
 = { def of (*) }
   c ^ (a * (b ^ (^)))
```

So,

```
  C a b
= { by the above, & extensionality }
  a * (b ^ (^))
= { def of (*) }
  (b ^ (^)) ^ a ^ (*)
= { def of (*) }
  b ^ ((^) * a ^ (*))
```

So,

```
  C a
= { by the above, & extensionality }
  (^) * a ^ (*)
= { def of (*) }
  (a ^ (*)) ^ (^) ^ (*)
= { def of (*) }
  a ^ ((*) * (^) ^ (*))
```

So `C = (*) * (^) ^ (*)` .

As for `B`, it is clear that it is the transpose of `(*)`, or in other words `C (*)` . From the middle step in the derivation of `C`'s arithmetic form, we therefore have as a by-product

```
B = (^) * (*) ^ (*)
```

As for `K`, it is clear that it is the transpose of `(<>)`, that is `C (<>)` and therefore, just as for `B`,

```
K = (^) * (<>) ^ (*)
```

As for `I`, it is the transpose of `(^)`, modulo extensionality. So we could take the following.

```
I = (^) * (^) ^ (*)
```

But we can also take the following.

```
I = (<>) ^ (<>)
```

As for `W`, first let's express its transpose `CW`

```
  CW a b
= { def }
  b a a
= { re-express using exponentiation }
  a ^ a ^ b
= { def of (^), twice }
  (b ^ a ^ (^)) ^ a ^ (^)
= { def of (*) }
  b ^ (a ^ (^) * a ^ (^))
= { def of (+) }
  b ^ a ^ ((^) + (^))
```

So by extensionality `CW = (^) + (^)`. From this, as in the cases of `B` and `K`, we have

```
W = (^) * ((^) + (^)) ^ (*)
```

This completes the derivation of the `BWICK` combinators.

After a little familiarisation with these combinators, it becomes clear that another combinator plays a central role. This is `S'`, where `S a b` (the normal S combinator) = W (S' a b).

```
  S' a b c1 c2
= { def }
  a c1 (b c2)
```

So,

```
  S' a b c1
= {}
  a c1 . b
= {}
  b * a c1
```

So,

```
  S' a b
= {}
  (b *) . a
= {}
  a * (b *)
```

So,

```
   S' a
 = {}
   (a *) . (*)
 = {}
   (*) * (a *)
```

So,

```
   S'
 = {}
   ((*) *) . (*)
 = {}
   (*) * ((*) *)
```

In particular, we have:

```
S' = S' (*)
C = S' (^)
B = S' (^) (*)
I = S' (^) (^)
K = S' (^) 0
W = S' (^) ((^) + (^))
```

This is at any rate pretty.

If one uses the notation "AMEN" for addition, multiplication, exponentiation and naught, then not only does one have the last word in combinators, the resulting programs resemble a four-letter genetic code. (With lots of irritating single parentheses.) If on the other hand one uses Haskell's section notation, programs in combinatory form resemble an advanced form of eye disease, the ultimate in obfuscation. If I get round to it I will make a postcript picture of a large one.

Origins

The arithmetical combinators above are similar to, but not exactly the same as those defined by Burge (Recursive Programming Techniques, Addison-Wesley 1975, section 1.8), Stenlund (Combinators, lambda-terms and proof theory, Almqvist and Wiksell 1972, sec 2.4), or Rosenbloom (the elements of mathematical logic, Dover 1950 sec 3.4), among other places. Of these people, as far as I know, only Stenlund noticed the combinatorial completeness; others have of course defined them in connection with so-called Church numerals.

These gentry have the wrong argument order. With their definitions, the multiplication combinator is the same as the `B` combinator. My definitions (at least for `*`) can be read into Wittgenstein's Tractatus (around 6.02 and 6.241). They also have also the authority of Cantor, who called the alternative 'repulsive' – abstoßende – having started off using it. (Potter, "Sets" Clarendon Press 1990, p 120.)

In combinatory logic, the observation that they are complete is (according to Stendlund's book, p22) due to F. B. Fitch. (Maybe Fitch never published this observation – Stenlund doesn't give an exact reference.) Or maybe the observation is in

```
[Fitch 1952]
  Fredric Brenton Fitch, Symbolic Logic, the Ronald Press Company, New
  York, 1952.
```

But I haven't looked at it yet.

PS: I was recently (summer 2001) told by Felice Cardone that Böhm (he of the trees) used arithmetical combinators as a basis for combinatory logic, with lambda interpreted as logarithm (better, λogarithm, as Böhm writes sometimes), for example in his papers:

```
@INCOLLECTION{bohm:1979,
      author = "C. B{\"o}hm",
      title = "Un mod{\`e}le arithm{\'e}tique des termes de la logique
      combinatoire",
      BOOKTITLE="Lambda Calcul et S{\'e}mantique Formelle des Langages
      de Programmation.
      Actes de la Sixi{\`e}me Ecole de Printemps d'Informatique
      Theorique, La Ch{\^a}tre, 1978",
      EDITOR="B. Robinet",
      PUBLISHER="LITP et ENSTA",
      ADDRESS="Paris",
      YEAR = 1979,
```

```
        PAGES = "97--108",
        }

@INCOLLECTION{bohm:1980,
        AUTHOR="C. B{\"o}hm",
        TITLE="Logic and computers",
        BOOKTITLE="Modern Logic --- A Survey",
        EDITOR="E. Agazzi",
        PUBLISHER={Reidel},
        ADDRESS="Dordrecht",
        PAGES={297--309},
        YEAR={1980}
        }
```

I don't have access to a library at the moment, but look forward to finding these!

algebra

When equality is interpreted as extensional equality,

. `(+,0)` forms a monoid.
. `(*,1)` forms a monoid, where `1 = combI` .
. `a * (b + c) = a * b + a * c` .
. `a * 0 = 0` .
. Addition and multiplication are not in general commutative.

Intriguingly enough, The laws of `(+),0,(*),1` above (which do not mention exponentiation) coincide with the basic laws of the transfinite arithmetic or addition and multiplication, or indeed of the theory of general (not necessarily linear) orderings, well-founded or not. (However the coincidence is not exact. In the combinatory world, we don't have, among other things, `0 * a = 0`. This is because we don't have `1 ^ a = 1`.)

If we think of "real" natural numbers, these also satisfy the following equations. These seem to characterise the "genuinely numerical" part of the model.

```
a + b = a ^ (+ 1) ^ b
a * b = 0 ^ (+ a) ^ b
a ^ b = 1 ^ (* a) ^ b
```

logarithms

One can introduce 'logarithms', as a sort of lambda-abstraction, with `x ^ (log_x e) = e`. Linear logarithms (with only one occurrence of the 'base' `x` in `e`) obey many delightful arithmetico-combinatorial laws, that can be figured out on the average paper napkin. What (for example) is `log_x a x b c`, where `a`, `b`, `c` have no free occurrences of `x`?

```
log_x a x b c  = log_x c ^ b ^ x ^ a
               = log_x ((x ^ a) ^ (b^)) ^ (c^)
               = a * (b^) * (c^)
```

Any linear abstraction is a product (ie. a composite) of "factors" that each look like

```
a * (b1^) * ... * (bk^)
```

A general abstraction (in which the "base" variable may have more than one occurrence) has the normal form

```
((^) + ... + (^)) * (pie^)
where pie = Pi_i (ai * (bi1^) * .. * (bij^))
```

where there is a summand for each occurrence of the variable.

There are many optimisations, such as

```
log_x (a * b) = (log_x a) + (log_x b)
log_x 1       = 0
```

These equations were once familiar to schoolchildren.

subtraction, division, logarithm

Dull would he be of soul who didn't wonder whether he couldn't define subtraction, division and logarithms such that

```
a + x .leq. b     <=>    x .leq. b - a
a * x .leq. b     <=>    x .leq. b / a
a ^ x .leq. b     <=>    x .leq. log_a b .
```

Perhaps ‘floor’ and ‘ceiling’ are worth thinking about too. My wonderings (mainly about what .leq. might mean) have not been productive.

Should one have gadgets like subtraction, division and logarithms, then one will surely have something like Cantor Normal form. This might be interesting.

What might it mean to represent subtraction, division, and logarithms as syntactical operations? ”Oleg” someone (I don’t know his full name) has an interesting collection of web pages at

```
http://www.pobox.com/~oleg/ftp/
```

about these and other matters. One idea of Oleg’s is to use fixed points. For example,

```
    -a,   1/a,   log_a b
```

is a fixed point of

```
\x -> a + x*2,  \x -> x * (a+1) - 1, \x->a ^ x + x - b
```

respectively.

He talks about least fixed points, but I think (integer, ie. cutoff) division and logarithms should (somehow) be greatest fixed points. But ”oleg” has other ideas too. An interesting guy!

Arithmetical calculus

Here is a datatype for arithmetical expressions. The symbols for the constructors are chosen to suggest their interpretation as combinators.

```
infixr 8  :^:
infixr 7  :*:
infixr 6  :+:

data E  = V String | E :^: E | E :*: E | E :+: E
          deriving (Show,Eq)
```

We can think of them as like Lisp S-expressions, but with three kinds of ‘cons’ operation.

It is convenient to allow constants identified by arbitrary strings, so we can have constants besides zero.

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.

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

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

a <*> b = case b of
               V "0"         -> b
               b1 :+: b2     -> a ^ (up2 (<+>) (<*> b1) (<*> b2))
               b1 :*: b2     -> a ^ ( (<*> b1) * (<*> b2) )
               _  :^: V "0" -> a
               _             -> a :*: b

a <^> b = case b of
               V "0"         -> b :^: b
               b1 :+: b2     -> a ^ (up2 (<*>) (<^> b1) (<^> b2))
               b1 :*: b2     -> a ^ ( (<^> b1) * (<^> b2) )
               _  :^: V "0" -> a
               b1 :^: V "^" -> b1 <^> a   -- note: destroys termination
               b1 :^: V "*" -> b1 <*> a
               b1 :^: V "+" -> b1 <+> a
 b1 :^: b2    -> a :^: (b1 <^> b2)
               _             -> a :^: b
```

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

```
eval a  = case a of b1 :+: b2    -> eval b1 <+> eval b2
                    b1 :*: b2    -> eval b1 <*> eval b2
                    b1 :^: b2    -> eval b1 <^> eval b2
                    _            -> a
```

A variant. Does it give the same answers?

```
eval' a  = case a of b1 :+: b2    -> b1 <+> eval' b2
                     b1 :*: b2    -> b1 <*> eval' b2
                     b1 :^: b2    -> b1 <^> eval' b2
                     _            -> a
```

A convenient abbreviation, used above for applying a binary function pointwise. For example `(+) = up2 (*)`.

```
up2 f a b c = (c ^ a) `f` (c ^ b)
```

show function for expressions

The parameter "p" below is precedence. I forget how to use it.

```
showE :: E -> Int -> String -> String
showE (V "^") = \p->("(^)"++)
showE (V "*") = \p->("(*)"++)
showE (V "+") = \p->("(+)"++)
showE (V str) = \p->(str++)
showE (a :+: b) = \p->opp p 0 (showE a 1 . (" + "++) . showE b 0)
showE (a :*: b) = \p->opp p 2 (showE a 3 . (" * "++) . showE b 2)
showE (a :^: b) = \p->opp p 4 (showE a 5 . (" ^ "++) . showE b 4)
opp p op f = if p > op then showString"(" . f . showString")" else f
```

Disable it. The default one is just as good.

```
> instance Show E where showsPrec p e = showE e 0
```

reduction sequences

We now define for each expression the list of expressions to which it can be reduced in one toplevel step.

One can tinker here. Be careful of overlap. Although the lists returned here are singletons, there can be overlap between rewriting rules.

```
redex :: E -> [E]
redex e = case e of
           (a :+: (b :+: c))   -> [ (a :+: b) :+: c          ]
           (a :+: V "0")       -> [ a                        ]
           (a :*: (b :+: c))   -> [ (a :*: b) :+: (a :*: c)  ]
           (a :*: V "0")       -> [ v0                       ]
           (a :*: (b :*: c))   -> [ (a :*: b) :*: c          ]
           (a :*: V "1")       -> [ a                        ]
           (a :^: (b :+: c))   -> [ (a :^: b) :*: (a :^: c)  ]
           (a :^: V "0")       -> [ v1                       ]
           (a :^: (b :*: c))   -> [ (a :^: b) :^: c          ]
           (a :^: V "1")       -> [ a                        ]
           (a :^: b :^: V "+") -> [ b :+: a                  ]
           (a :^: b :^: V "*") -> [ b :*: a                  ]
           (a :^: b :^: V "^") -> [ b :^: a                  ]
           -- (a :^: b :^: V "0") -> [ a ]
           _                   -> [                          ]
v0 = V "0"
v1 = V "1"    -- v0 :^: v0
```

To represent subexpressions, we use a clever idea of Mark Jones, Hugo Herberlin, or whoever. We represent each part `e` of an expression `e'` as a pair `(f,e)` consisting of the subexpression `e'` there, and a 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 'socket' `f` to get back `e'`.

```
parts, pparts :: E -> [(E->E,E)]
parts e  = (id,e) : pparts e
pparts e = case e of (a :+: b) -> g (:+:) a b
                     (a :*: b) -> g (:*:) a b
                     (a :^: b) -> g (:^:) a b
```

```
    _          -> []
  where g op a b =
          [ ((a `op`) . f,p) | (f,p) <- parts b ]
       ++ [ ((`op` b) . f,p) | (f,p) <- parts a ]
```

Now we define for any expression the list of expressions to which it reduces in a single, possibly internal step.

```
reducts :: E -> [ E ]
reducts a     = [ f a'' | (f,a') <- parts a, a'' <- redex a' ]
```

We need a structure to hold the reduction sequences from an expression. So-called 'rose' trees seem to be ideal.

```
data Tree a = Node a [ Tree a ]
```

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

```
exec :: E -> Tree E
exec e = Node e [ exec e' | e' <- reducts e ]
```

We use the branches in a tree to represent reduction sequences.

```
branches :: Tree a -> [[a]]
branches (Node a []) = [ [a] ]
branches (Node a ts) = [ a : b | t <- ts, b <- branches t]
```

A function for displaying a tree, branch by branch.

```
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 "! "))
                          . number ) ts)
```

A function for displaying a list with numbers.

```
newtype NList a = NList [a]
instance Show a => Show (NList a) where
  showsPrec p (NList es) =
     (composelist . seplist ('\n':) . map showline . number ) es
     where showline (n,e) = shows n . showString ". " . shows e
```

Some expressions for playing with

```
cC    = (V "*") :*: (V "^") :^: (V "*")
cB    = (V "^") :*: (V "*") :^: (V "*")
cK    = (V "^") :*: (V "0") :^: (V "*")
cI    = jnk :^: V "0"
cI'   = (V "^") :*: (V "^") :^: (V "*")
cW    = (V "^") :*: (V "^" :+: V "^") :^: (V "*")
c0    = V "0"

jnk   = V "Junk" -- evil"The unthinkable has happened"
```

Rotation combinators

The following combinator `cR` 'rotates' 3 arguments.

```
vc :^: vb :^: va :^: cR   = va :^: vc :^: vb
```

Such an operation is often provided by an instruction of a 'stack machine', to rotate the top three enties on the stack.

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

```
cR :^: cR :^: cR = cC
cC :^: cC        = cR
```

One upshot of this observation is that we have a way of churning out endless versions of the C combinator, aka `flip`.

```
flip', flip'' :: (a -> b -> c) -> b -> a -> c
flip' = flip flip (flip flip)  (flip flip)
flip'' = flip' flip' (flip' flip')  (flip' flip')
```

We can write the rotation combinator as follows.

```
cR     = cC :^: cC
```

Some other versions

```
combR    =  (^) * (((*) * ((^)*))*)
combR'   =  (^) * (^) * ((*)*)

cR'    = (V "^") :*: (V "^") :*: (V "*") :^: (V "*")
```

Numbers.

```
naught = c0
one    = cI
two    = one   :+: one
three  = two   :+: one
four   = two   :^: two
five   = two   :+: three
six    = two   :*: three
seven  = three :+: four
eight  = two   :^: three
nine   = three :^: two
ten    = two   :*: five
```

S combinator.

The following combinator is useful for defining the S combinator, since `S a b = W (S' a b)` It has quite interesting properties: `S' (*) = S'` , `S' (^) = C` , and so on.

```
cS'    = let x = V"*" in x :*: (x :^: x )
cS     = cS' :*: cW :^: cB

combS' = let x = (*) in x * (x ^ x)
combS  = combS'  *  (*)  * (combW ^)
```

Y combinator.

```
cSap   = V"^" :^: cW
cY     = let x = cSap :^: V"*" in x :^: x :^: cS
```

Although `(cI :^: cY)` runs in constant space (Hugs doesn't run out of stack or heap), matters are different with `(cI' :^: cY)`.

Alternative W combinator

I retain this for sentimental reasons, as it was the first definition I thought of, though clearly a very terrible one.

```
cW'   = t1 :*: V "*" :*: t2
        where t1 = cK :^: V"+"
              t2 = jnk :^: (V"^" :*: V"^")
```

Some mildly interesting functions.

```
ct a       = a :^: V "^"      -- unit of continuation transform
```

Question: what do the continuation transforms look like in arithmetic combinators?

```
line gradient x0 x = x0 :+: gradient :*: x
```

The idea is to look at some schoolroom things like the equations of lines, polynomials, derivates, etc.

```
sgbar n = V "0" :^: n
```

sgbar is the function which is 1 at 0, and 0 everywhere else. It can be used to define sg, which is 0 at 0, and 1 elsewhere by
sg = sgbar . sgbar

It is interesting because it is non-monotone on numerals, and somehow the source of all non-monotonicity.

Some convenient things

```
va    = V "a"
vb    = V "b"
vc    = V "c"
vd    = V "d"
ve    = V "e"
vf    = V "f"
```

```
number :: [a] -> [(Int,a)]
number = zip [1..]
```

```
composelist :: [ a -> a ] -> a -> a
composelist = foldr (.) id
```

```
seplist :: a -> [a] -> [a]
seplist s (x:(xs'@(_:_))) = x:s:seplist s xs'
seplist s xs = xs
```

First difference

```
fdiff :: Eq a => [a] -> [a] -> (a,a)
fdiff (a:as) (b:bs) | a /= b = (a,b)
fdiff (a:as) (b:bs) | True   = fdiff as bs
fdiff _      _               = error"fdiff"
```

Testing and experimenting

```
help = "The tests are: test, number_rss, nth_rs"
```

The first reduction sequence

```
test :: E -> NList E
test = NList . (nth_rs 0)
```

The n'th reduction sequence.

```
nth_rs :: Int -> E -> [E]
nth_rs n = (!! n) . branches . exec
```

The number of reduction sequences

```
number_rss :: E -> Int
number_rss = length . branches . exec
```

Projections

```
p22 = cK
p21 = c0
```

```
p33 = (cK :*: cK) :^: cK
p32 = cK :^: cK
p31 = c0 :^: cK
```

```
p44 = cK :*: cK :*: cK
p43 = (cK :*: cK) :^: cK
p42 = cK :^: (cK :*: cK)
p41 = c0 :^: (cK :*: cK)
```

```
p55 = cK :*: cK :*: cK :*: cK
p54 = (cK :*: cK :*: cK) :^: cK
p53 = (cK :*: cK) :^: (cK :*: cK)
p52 = cK :^: (cK :*: cK :*: cK)
p51 = c0 :^: (cK :*: cK :*: cK)
```

Tests of projections

```
test5 p =  test( ((V"a" :^: V"^") :*:
                  (V"b" :^: V"^") :*:
                  (V"c" :^: V"^") :*:
                  (V"d" :^: V"^") :*:
                  (V"e" :^: V"^"))
                 :^: (p :^: V"^")
               )
```

```
test4 p =  test( ((V"a" :^: V"^") :*:
                  (V"b" :^: V"^") :*:
                  (V"c" :^: V"^") :*:
                  (V"d" :^: V"^"))
                 :^: (p :^: V"^")
               )
```

```
test3 p =  test( ((V"a" :^: V"^") :*:
                  (V"b" :^: V"^") :*:
                  (V"c" :^: V"^"))
                 :^: (p :^: V"^")
               )
```

```
test2 p =  test( ((V"a" :^: V"^") :*:
                  (V"b" :^: V"^"))
                 :^: (p :^: V"^")
               )
```

Bureaucracy

```
main :: IO()
main = error"Meant for interpreter"
```