AMEN: arithmetical calculator

Peter Hancock

June 21, 2016

1 The Naperian combinators: $+$, $\times$, $\wedge$, 0.

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

```
module Main where
import Prelude hiding ((×), (∧), (+), (◦)
  , (⟨×⟩), (⟨∧⟩), (⟨+⟩), (<<>>)
  , (:∧:), (:×:), (:+:), (:<>:))
infixr 8 ∧
infixr 7 ×
infixr 6 +
infixr 9 ◦
```

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

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

Instead of *naught*, I shall use the infix operator $\circ$, that looks a little like a '0'. It throws away its left argument, and returns its right.

$$
\begin{aligned}
&\circ = naught\\
&zero = naught\\
&one = zero \wedge zero
\end{aligned}
$$

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

$$
\begin{aligned}
&(\wedge) :: a \to (a \to b) \to b\\
&(\times) :: (a \to b) \to (b \to c) \to a \to c\\
&(+) :: (a \to b \to c) \to (a \to c \to d) \to a \to b \to d\\
&(\circ) \quad :: a \to b \to b\\
&one :: a \to a
\end{aligned}
$$

The definitions above generate an equivalence relation between (possibly open) terms in a signature: 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.

$$x \wedge a = x \wedge b \ \Rightarrow\ a = b$$

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

The ζ-rule is an expression of 'exponentiality': two values that behave the same as exponents of a generic base are the very same value. I shall call this relation ζ-equality. All equations asserted below should be interpreted as ζ-equations.

2 Arithmetical calculus

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 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 | E :∧: E | E :×: E | E :+: E | E :<>: E
       deriving (Eq)   -- (Show,Eq)
```

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

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

$$(cA, cM, cE, c0) = (V\ \texttt{"+"}, V\ \texttt{"*"}, V\ \texttt{"\^{}"}, V\ \texttt{"0"})$$

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
                                             b1 :+: b2     → (a⟨×⟩b1)⟨+⟩(a⟨×⟩b2)
                                             b1 :×: b2     → (a⟨×⟩b1)⟨×⟩b2
                                             _ :∧: V "0"   → a
                                             _             → a :×: b
a⟨∧⟩b = case b of V "0"          → b :∧: b
                  b1 :+: b2      → (a⟨∧⟩b1)⟨×⟩(a⟨∧⟩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
_ <<>> b = b
```

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

$$
\begin{array}{l}
eval :: E \to E \\
eval\ a = \textbf{case}\ a\ \textbf{of}\ b_1 :+: b_2 \to eval\ b_1 \langle + \rangle eval\ b_2 \\
\qquad\qquad\qquad\qquad\quad b_1 :\times: b_2 \to eval\ b_1 \langle \times \rangle eval\ b_2 \\
\qquad\qquad\qquad\qquad\quad b_1 :\wedge: b_2 \to eval\ b_1 \langle \wedge \rangle eval\ b_2 \\
\qquad\qquad\qquad\qquad\quad \_ :<>: b_2 \to eval\ b_2 \\
\qquad\qquad\qquad\qquad\quad \_ \qquad\quad \to a
\end{array}
$$

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) if one exists. Such a thing lets us see the value of an expression, but not how it was arrived at.

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

3 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.

$$
\begin{array}{l}
tlr :: E \to [E] \\
tlr\ e = \textbf{case}\ e\ \textbf{of}\ (a :+: (b :+: c)) \to [(a :+: b) :+: c] \\
\qquad\qquad\qquad\qquad\quad (a :+: V\ \texttt{"0"}) \to [a] \\
\qquad\qquad\qquad\qquad\quad (V\ \texttt{"0"} :+: a) \to [a] \\
\qquad\qquad\qquad\qquad\quad (a :\times: (b :+: c)) \to [(a :\times: b) :+: (a :\times: c)] \\
\qquad\qquad\qquad\qquad\quad (a :\times: V\ \texttt{"0"}) \to [c0] \\
\qquad\qquad\qquad\qquad\quad (a :\times: (b :\times: c)) \to [(a :\times: b) :\times: c] \\
\qquad\qquad\qquad\qquad\quad (a :\times: V\ \texttt{"1"}) \to [a] \\
\qquad\qquad\qquad\qquad\quad (V\ \texttt{"1"} :\times: a) \to [a] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: (b :+: c)) \to [(a :\wedge: b) :\times: (a :\wedge: c)] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: V\ \texttt{"0"}) \to [c_1] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: (b :\times: c)) \to [(a :\wedge: b) :\wedge: c] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: V\ \texttt{"1"}) \to [a] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: b :\wedge: V\ \texttt{"+"}) \to [b :+: a] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: b :\wedge: V\ \texttt{"*"}) \to [b :\times: a] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: b :\wedge: V\ \texttt{"\^{}"}) \to [b :\wedge: a] \\
\qquad\qquad\qquad\qquad\quad (a :\wedge: b :\wedge: V\ \texttt{"0"}) \to [a] \\
\qquad\qquad\qquad\qquad\quad (a :<>: b) \to [b] \\
\qquad\qquad\qquad\qquad\quad \_ \to [] \\
\quad \text{-- v0 = V "0 "} \\
c_1 = V\ \texttt{"1"} \quad \text{-- } v0 :\wedge: v0
\end{array}
$$

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.

```
sites :: E → [(E → E, E)]
sites e = (id, e) : case e of
            (a :+: b) → h (:+:) a b
            (a :×: b) → h (:×:) a b
            (a :∧: b) → h (:∧:) a b
            _         → []
          where
            h op a b
              = l ++ r where l = [((a`op`) ∘ f, p) | (f, p) ← sites b]
                             r = [((`op`b) ∘ f, p) | (f, p) ← sites a]
```

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.

```
reducts :: E → [E]
reducts a = [f a'' | (f, a') ← sites a, a'' ← tlr a']
```

3.1 holding reduction sequences in a tree

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] deriving Show
```

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

```
reductTree :: E → Tree E
reductTree e = Node e [reductTree e' | e' ← reducts e]
```

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

```
branches :: Tree a → [[a]]
branches (Node a []) = [[a]]
branches (Node a ts) = [a : b | t ← ts, b ← branches t]
```

Putting things together, we can map an expression to its sequence of reduction sequences (the reductions being chosen breadth-first, left-right.

```
rss :: E → [[E]]
rss = branches ∘ reductTree
```

4 Böhm's logarithms

This code is concerned generating the logarithm of an expression with respect to a variable name.

Böhm's combinators

```
cBohmA a b = let g = a :∧: V "^" :+: b :∧: V "^" in
  let t = cPair :+: g :∧: cK in    -- Bohm's original
  let t' = cPair :×: V "*" :×: (g :∧: V "^")   -- another without additive apparatus
  in t'
cBohmE a b = a :×: cPair :+: b :×: V "^"
cBohmM a b = a :+: b
cBohm0 a = V "0" :×: a :∧: V "^"
```

These have the crucial properties

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

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.

```
blog v e | ¬ (v ∈ fvs e) = cBohm0 e
blog v e = case e of
  a :+: b →  {-cBohmA (blog v a) (blog v b) -}
    case (v ∈ fvs a, v ∈ fvs b) of
      (False, True) → (blog v b) :×: (a :∧: cA)
      (True, False) → (blog v a) :×: (b :∧: cA :∧: cC)
      _             → cBohmA (blog v a) (blog v b)
  a :×: b → case (v ∈ fvs a, v ∈ fvs b) of
      (False, True) → (blog v b) :×: (a :∧: cM)
      (True, False) → (blog v a) :×: (b :∧: cB)
      _             → cBohmM (blog v a) (blog v b)
  a :∧: b → case (v ∈ fvs a, v ∈ fvs b) of
      (False, True) → case b of
        V v → a :∧: cE
        _   → blog v b :×: (a :∧: cE)
      (True, False) → case a of
        V v → b
        _   → blog v a :×: b
      _ → cBohmE (blog v a) (blog v b)
  V nm → if nm ≡ v then c1 else cBohm0 e
```

Make a list of all the variables occurring in an expression.

```
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, some names for 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 "n"
    , 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  = c2 :+: c3
c6  = c3 :×: c2
c7  = c3 :+: c4
c8  = c2 :∧: c3
c9  = c3 :∧: c2
c10 = c2 :×: c5
```

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. I don't understand precedences very well. I think the following deals properly with associativity of + and ×, and their relative precedences (sums of products) but also with the non-associativity of ∧. 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 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)
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 an *NTree a* that indentation in an attempt to make the branching structure of the tree visible. (Actually, this is entirely useless.)

```
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)
```

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

```
newtype NList a = NList [a]
instance Show a ⇒ Show (NList a) where
  showsPrec _ (NList es) =
    (composelist ∘ commalist ('\n':) ∘ map showline ∘ number) es
    where showline (n, e) = shows n ∘ showString ": " ∘ shows e
```

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

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


composelist :: [a → a] → a → a
composelist = foldr (∘) id
```

Code to insert a 'comma' at intervening positions in a list.

```
commalist :: a → [a] → [a]
commalist c (x : (xs'@(_: _))) = x : c : commalist c xs'
commalist c xs = xs


nodups [] = []
nodups (x : xs) = x : nodups (filter (≢ x) xs)
```

B.3 Some interesting things to use

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

```
test $ vu :∧: vz :∧: vy :∧: vx :∧: cS
```

and see in response:

```
1 : u ∧ z ∧ y ∧ x ∧ ((×) × (×) ∧ (×) × ((∧) × ((∧) + (∧)) ∧ (×)) ∧ ((∧) × (×) ∧ (×)))
2 : u ∧ z ∧ y ∧ (x ∧ ((×) × (×) ∧ (×))) ∧ ((∧) × ((∧) + (∧)) ∧ (×)) ∧ ((∧) × (×) ∧ (×))
3 : u ∧ z ∧ y ∧ (x ∧ ((×) × (×) ∧ (×))) ∧ (((∧) × ((∧) + (∧)) ∧ (×)) ∧ (∧)) ∧ (×) ∧ (×)
4 : u ∧ z ∧ y ∧ (x ∧ ((×) × (×) ∧ (×))) ∧ ((×) × ((∧) × ((∧) + (∧)) ∧ (×)) ∧ (∧))
5 : u ∧ z ∧ y ∧ ((x ∧ ((×) × (×) ∧ (×))) ∧ (×)) ∧ ((∧) × ((∧) + (∧)) ∧ (×)) ∧ (∧)
6 : u ∧ z ∧ y ∧ ((∧) × ((∧) + (∧)) ∧ (×)) ∧ (x ∧ ((×) × (×) ∧ (×))) ∧ (×)
7 : u ∧ z ∧ y ∧ (x ∧ ((×) × (×) ∧ (×)) × (∧) × ((∧) + (∧)) ∧ (×))
8 : u ∧ z ∧ (y ∧ x ∧ ((×) × (×) ∧ (×))) ∧ ((∧) × ((∧) + (∧)) ∧ (×))
9 : u ∧ z ∧ ((y ∧ x ∧ ((×) × (×) ∧ (×))) ∧ (∧)) ∧ ((∧) + (∧)) ∧ (×)
10 : u ∧ z ∧ (((∧) + (∧)) × (y ∧ x ∧ ((×) × (×) ∧ (×))) ∧ (∧))
11 : u ∧ (z ∧ ((∧) + (∧))) ∧ (y ∧ x ∧ ((×) × (×) ∧ (×))) ∧ (∧)
```

$12 : u \wedge (y \wedge x \wedge ((\times) \times (\times) \wedge (\times))) \wedge z \wedge ((\wedge) + (\wedge))$
$13 : u \wedge (y \wedge x \wedge ((\times) \times (\times) \wedge (\times))) \wedge (z \wedge (\wedge) \times z \wedge (\wedge))$
$14 : u \wedge ((y \wedge x \wedge ((\times) \times (\times) \wedge (\times))) \wedge z \wedge (\wedge)) \wedge z \wedge (\wedge)$
$15 : u \wedge z \wedge (y \wedge x \wedge ((\times) \times (\times) \wedge (\times))) \wedge z \wedge (\wedge)$
$16 : u \wedge z \wedge z \wedge y \wedge x \wedge ((\times) \times (\times) \wedge (\times))$
$17 : u \wedge z \wedge z \wedge y \wedge (x \wedge (\times)) \wedge (\times) \wedge (\times)$
$18 : u \wedge z \wedge z \wedge y \wedge ((\times) \times x \wedge (\times))$
$19 : u \wedge z \wedge z \wedge (y \wedge (\times)) \wedge x \wedge (\times)$
$20 : u \wedge z \wedge z \wedge (x \times y \wedge (\times))$
$21 : u \wedge z \wedge (z \wedge x) \wedge y \wedge (\times)$
$22 : u \wedge z \wedge (y \times z \wedge x)$
$23 : u \wedge (z \wedge y) \wedge z \wedge x$

$test :: E \to NList\ E$
$test = NList \circ hd \circ rss$
$hd\ (x{:}\ \_) = x$

The n'th reduction sequence.

$nth\_rs :: Int \to E \to NList\ E$
$nth\_rs\ n = NList \circ (!!n) \circ rss$

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.

$stats\_rss :: E \to (Int, (Int, Int))$
$stats\_rss\ e = \mathbf{let}\ (b_0 : bs) = map\ length\ (rss\ e)$
  $\mathbf{in}\ (length\ (b_0 : bs), (foldr\ min\ b_0\ bs, foldr\ max\ b_0\ bs))$

$nf :: E \to [E]$
$nf = map\ last \circ rss$
$fd :: [E] \to Maybe\ (E, [E])$
$fd\ [\,] = Nothing$
$fd\ [x] = Nothing$
$fd\ (x : xs) = \mathbf{let}\ e = [t \mid t \leftarrow xs, t \not\equiv x]$
  $\mathbf{in\ if}\ e \equiv [\,]\ \mathbf{then}\ Nothing\ \mathbf{else}\ Just\ (x, e)$

C Examples

In this section, we give the arithmetical form of some naturally occurring combinators.

C.1 CBKIWSS'

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

$cC, cB, cK, cI, cI', cW, c0 :: E$
$cC = V\ \texttt{"*"} :\times: V\ \texttt{"\^{}"} :\wedge: V\ \texttt{"*"}$ -- M to one plus E to the E
$cB = V\ \texttt{"\^{}"} :\times: V\ \texttt{"*"} :\wedge: V\ \texttt{"*"}$ -- M to the C
$cK = V\ \texttt{"\^{}"} :\times: V\ \texttt{"0"} :\wedge: V\ \texttt{"*"}$ -- 0 to the C
$cI = V\ \texttt{"@"} :\wedge: V\ \texttt{"0"}$
$cI' = V\ \texttt{"\^{}"} :\times: V\ \texttt{"\^{}"} :\wedge: V\ \texttt{"*"}$ -- E to the C
$cW = V\ \texttt{"\^{}"} :\times: (V\ \texttt{"\^{}"} :+: V\ \texttt{"\^{}"}) :\wedge: V\ \texttt{"*"}$ -- twice E to the cC

The 'real word' versions:

$combC = (\times) \times (\wedge) \wedge (\times)$ -- *flip*, transpose.
$combB = (\wedge) \times (\times) \wedge (\times)$ -- $(\circ)$, composition. $(\times) \wedge combC$
$combI = evil \wedge ()$ -- *id*. also $(\wedge) \times (\wedge) \wedge (\times)$, inter alia
$combK = (\wedge) \times () \wedge (\times)$ -- *const*. $() \wedge combC$
$combW = (\wedge) \times ((\wedge) + (\wedge)) \wedge (\times)$ -- diagonalisation. $((\wedge) + (\wedge)) \wedge combC$
$evil = error$ `"Naughty"`

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

$$S'\ a\ b\ c_1\ c_2 = a\ c_1\ (b\ c_2)$$

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:

$combS' = \mathbf{let}\ x = (\times)\ \mathbf{in}\ x \times (x \wedge x)$
$combS = combS' \times (\times) \times (combW \wedge)$

The following arithmetical code implements variants of the S and S' combinators.

$cS, cS' :: E$
$cS \quad = cS' :\times: cW :\wedge: cB$
$cS' \quad = cM :\times: (cM :\wedge: cM)$
-- the following serves as a check that the logarithm apparatus is working.
-- this should evaluate in the correct way.
$cSalt \quad = blog$ `"y"` $(blog$ `"x"` $((vx :\times: cPair) :+: (vy :\times: cE)))$

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.

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

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

$$cPair = V\ \texttt{"\^{}"} :\times: V\ \texttt{"*"} :\times: (V\ \texttt{"\^{}"} :\wedge: V\ \texttt{"*"})$$

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

$$
\begin{aligned}
cCurry &= cK :\times: (cPair :\wedge: V\ \texttt{"+"}) \\
cCurry' &= cB :\times: (cPair :\wedge: cM)
\end{aligned}
$$

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

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

$$
\begin{aligned}
cTrue &= V\ \texttt{"0"} :\wedge: cC \\
cFalse &= V\ \texttt{"0"}
\end{aligned}
$$

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

$$
\begin{aligned}
cFst &= cTrue :\wedge: V\ \texttt{"\^{}"} \\
cSnd &= cFalse :\wedge: V\ \texttt{"\^{}"}
\end{aligned}
$$

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

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

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 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:

$$\begin{aligned} &cR, cR' :: E \\ &cR \quad\ = cC :\wedge: cC \\ &cR' \quad = (V\ \texttt{"\^{}"}) :\times: (V\ \texttt{"\^{}"}) :\times: (V\ \texttt{"*"}) :\wedge: (V\ \texttt{"*"}) \end{aligned}$$

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

$$\begin{aligned} &cR :\wedge: cR :\wedge: cR = cC \\ &cC :\wedge: cC = cR \end{aligned}$$

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

$$\begin{aligned} &flip', flip'' :: (a \to b \to c) \to b \to a \to c \\ &flip' = flip\ flip\ (flip\ flip)\ (flip\ flip) \\ &flip'' = flip'\ flip'\ (flip'\ flip')\ (flip'\ flip') \end{aligned}$$

C.5 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

$$CT\ a = (a \to R) \to R$$

is the well known continuation monad. The unit *return* and multiplication *join* of this monad have simple arithmetical expressions.

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

TODO: Is this for real? The bind operator $\gg\!=$ is not quite as simple.

$$\begin{aligned}
&m \gg\!= f = m \circ (f \wedge C) \\
&\quad = (f \wedge C) \times m \\
&\quad = (\wedge) \times (f\times) \times m \\
&(\gg\!=) \wedge C\ f\ m = (\wedge) \times (f\times) \times m \\
&(\gg\!=) \wedge C\ f = ((\wedge)\times) \circ ((f\times)\times) \\
&\qquad\qquad = ((f\times)\times) \times ((\wedge)\times) \\
&f \wedge ((\gg\!=) \wedge C) = ((f \wedge (\times)) \wedge (\times) \times (\wedge) \wedge (\times) \\
&(\gg\!=) \wedge C \quad = (\times) \times (\times) \times (((\wedge) \wedge (\times)) \wedge B) \\
&(\gg\!=) \wedge C \quad = (\times) \times (\times) \times (\wedge) \wedge ((\times) \times B) \\
&(\gg\!=) \qquad = ((\times) \times (\times) \times (\wedge) \wedge ((\times) \times B)) \wedge C
\end{aligned}$$

You may interested to see what *callCC* looks like. That's the function whose type is the Kleislified version of Peirce's law: $((a \to CT\ b) \to CT\ a) \to CT\ a$. Unless I'm missing something, it is not very beguiling:

$$(((\times) \times ((\wedge) \times 0 \wedge (\times)) \wedge (\wedge)) \wedge (\times) \times (\wedge)) \times ((\wedge) + (\wedge)) \wedge (\times)$$

Well, that's classical logic for you.

The monadic apparatus can be encoded as follows

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

C.6 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

$$n\ b\ a = b\ (n-1)\ (b\ (n-2)\ ...\ (b\ 1\ (b\ (')\ a))...)$$

Using the combinators of this paper, one can derive

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

By exponentiality (ζ),

$$n = ((')\wedge) + (1\wedge) + ... + ((n-2)\wedge) + ((n-1)\wedge)$$

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

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

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.

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

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

C.7 sgbar

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.

$$sgbar = ((')\wedge)$$

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).

$$sg = sgbar \times sgbar$$

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.