Tractatus logarithmico-arithmeticus
(the Naperian combinators)

Peter Hancock

March 31, 2018

Contents

List of Figures

Abstract

The λ-calculus, typed or not, has a little-known arithmetical aspect. We arrive at it by considering application, exponentiation and even iteration to be the essentially the same operation.

The first indication that this might make sense is the coincidence between superscript notation f^n for iteration, which applies the exponent or iterator n to unary operation f, and m^n for exponentiation. As Wittgenstein said, a number is an exponent of an operation.

This brings in its train a set of 'AMEN' combinators for addition, multiplication, exponentiation and naught (zero) of Church-numerals. They have some some useful algebraic properties (in the presence of the ζ-rule). Moreover, they are combinatorially complete, in that they support a certain notion of 'logarithm', consistent with but extending some of Napier's laws of logarithms.

The interest of logarithms (to a 'variable' base!) seems to be originally pointed out and explored by Böhm in the late 1970's, in some undeservedly little-known papers from that time. His arithmetic and algebra of combinators has great entertainment value. But there is some intellectual value in Böhm's arithmetical ruminations, that I think bears some repeating.

1 Introduction

What are numbers? Leaving aside the numbers we use when measuring things, specifically, what are the numbers we use when we are counting off operations, as when giving someone change for something bought in a shop, or playing the drums? The answer is that numbers are fundamentally iterators, or templates for keeping track of iterations.

Something similar can be said about elements of any initial algebra. The natural numbers are just a particularly simple case.

The idea of iteration carries within it the idea of exponentiation, or raising one number to the power of another. An iterator a takes an endofunction $f : X \to X$ on a set X, and gives back its a-th iterate $f' : X \to X$, and so is itself an endofunction (on $X \to X$) that can be iterated, say b times. This takes $f : X \to X$ to $f \wedge (\exp_a b)$, and indeed $(+1) : X \to X$ and $o : X$ to $o + a^b$. Exponentiation b^e is itself a form of iteration: it is iterated multiplication by b (ie. $(\times b)$), starting at 1.

Exponential notation for numbers is at least as old as Archimedes, who considered the problem of counting or estimating immensely many grains of sand. He devised an exponential notation in which he expressed an upper bound for the number of grains it would take to fill the universe.

Numbers are among the first abstract things to which we give names, and notation. A handly notation for iteration is superscription, writing the number of iterations as a superscript of the expression for the iterated operation f

$$f^n$$

It is not an accident that we often use the same superscription notation for numerical exponentiation, with the base replacing the operator, and the exponent replacing the iteration

$$\begin{array}{lcl} b^e & = & (\times b)^e(1) \\ b \times e & = & (+b)^e(0) \\ b + e & = & (+1)^e(b) \end{array}$$

In these equations, the first superscription denotes numerical exponentiation, and the others denote iteration.

Superscripts can soon get out of hand at about 2 levels of nesting, so in practice one needs a infix binary operator such as $b \wedge e$. Still, superscription is sometimes helpful in helping to read expressions without undue parenthetical clutter, and take in their grammatical structure. So I'll often use superscription (particularly when the exponent expression is simple) as well as the infix operator,

What this paper is about is the use of exponential notation for function application itself, but with the function as the right operand. Hand in hand with exponential notation comes a useful algebraic calculus ('exponential calculus', or Cantor's laws of exponents) involving the operators $\times$, 1, + and 0 for working with exponential notation:

$$\begin{array}{lll} c^{a \times b} & = (c^a)^b & a^1 = a \\ c^{a+b} & = c^a \times c^b & a^0 = 1 \end{array}$$

A certain part of this calculus, may be re-expressed in terms of logarithms to an 'indeterminate' base x::

$$\begin{array}{lll} (\log_x b) \times c & = \log_x(b^c) & 1 = \log_x x \\ \log_x b + \log_x c & = \log_x(b \times c) & 0 = \log_x 1 \end{array}$$

If one thinks of application as raising the argument to the power of the function, then surely some form of λ-abstraction, or 'bracket abstraction' should correspond to the logarithm. This is indeed the case, as originally shown by Böhm over 3 decades ago, to be re-presented in section 6 on page 10.

2 The AMEN combinators: $+$, $\times$, $\wedge$, 0.

We can read Cantor's laws as *defining* combinators $(\times)$,1,$(+)$,0, using superscript (a^f) or exponential $(a \wedge f)$ notation for application, instead of applicative $f(a)$ notation, as in various Haskell notations like $f\ a$ or $f\ \$\ a$.

In this notation the binary operators are related to the combinators by the arithmetically mind-boggling laws:

$$b \wedge c = c \wedge b \wedge (\wedge)$$
$$b \times c = c \wedge b \wedge (\times)$$
$$b + c = c \wedge b \wedge (+)$$

It is one of the very nice features of Haskell that we can take over and redefine symbols such as for the arithmetical operations $(+)$, $(\times)$, and $(\wedge)$. Three (out of four) cheers for the 'hiding' keyword! However, it doesn't seem that we can take over '0'. [1] This is a slight pity, as it is the only other symbol I need. I have type-set it here as a dot that is almost invisible: with brackets around it looks like a squashed '0'.

```
module Main where
import Prelude hiding ((×), (∧), (+), (˙), ((˙×)), ((˙∧)), ((˙+)), ((˙˙)))
infixr 8∧
infixr 7×
infixr 6+
infixr 9˙       -- yes, there is a symbol there.
```

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

$$a \wedge b = b\ a$$
$$a \times b = \lambda c \to (c \wedge a) \wedge b$$
$$a + b = \lambda c \to (c \wedge a) \times (c \wedge b)$$
$$(\dot{}\,)\ a\ b = b \qquad \text{-- Written infix, } a\ `naught`\ b = b \text{ the operator would be invisible.}$$

Instead of *naught* [2], I have used the almost unnoticable symbol '˙' as an an infix operator, on the grounds that in prefix form '$(\dot{})$' it looks a little like '0'. It throws away its left argument, and returns its right.

$$naught = (\dot{})$$

The type-schemes[3] inferred for the definitions are as follows:

$$(\wedge) :: a \to (a \to b) \to b$$
$$(\times) :: (a \to b) \to (b \to c) \to a \to c$$
$$(+) :: (a \to b \to c) \to (a \to c \to d) \to a \to b \to d$$
$$(\dot{}\,) :: a \to b \to b$$

Equations can be proved by substituting equals for equals. One can also allow instances of the following "ζ" Rule in proving equations.

[1] (PS: It seems that sufficiently advanced ghc-specific voodoo exists that one can actually extricate '0' from ghc's gullet, and treat it as an bona-fide identifier. I have not found an opportunity to try.

[2] I use old-fashioned spelling "naught": the word is in fact etymologically connected with "naughty". A lot of fairly salacious word-play in Shakespeare's plays skates around this. It amuses me that the concept of zero was thought to be "dangerous Saracen magic" in medieval times (William of Malmesbury). According to John Donne, "The less anything is, the less we know it: how invisible, unintelligible a thing is nothing". It Noths, according to Heidegger. I think of its symbol as the first letter of 'Origin'.

[3] Almost certainly some citable publication contains a Hilbert-style axiomatisation of propositional logic of the conditional $(\to)$ equivalent to these type-schemes, at least modulo permutating the antecedents of a conditional.

Of course it is not uncommon to use exponential and other arithmetic notation at the level of types, which amounts to an arithmetic of cardinals. But such a precious notational device as exponentiation should not be too heavily overloaded.

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

with the side condition that x is fresh to both a and b. This is is really a cancellation law. All equations asserted below should be interpreted as ζ-equations.

By using "AMEN" notation for the combinators addition, multiplication, exponentiation and naught (aliases: nil, null, nihil, none, non-entity, nothing, nought, ought'nt ...), we utter, in reverse, from nothingness to abundance, the last word in combinators [4].

$$\begin{aligned}
&N\ a\ b = b\\
&E\ a\ b = b\ a\\
&M\ a\ b\ c = E\ (E\ c\ a)\ b\\
&A\ a\ b\ c = M\ (E\ c\ a)\ (E\ c\ b)
\end{aligned}$$

3 The BWICK combinators: *const*,*id*,(·),*flip*, diagonalisation

A set of combinators and equational laws is combinatorially complete if they suffice to simulate or compile $\lambda-$abstraction with β-contraction into combinatorial code. For precise definition and full discussion, see [Bar84] or [HS86].

It happens that $(+), (\times), (\wedge)$ and $()$ are combinatorially complete. The gist of it is that you can translate, or compile an expression e written using a fresh variable x into an applicative expression $[x]e$ in which that x does not occur, but only the combinators $(+), (\times), (\wedge)$ and $()$, such that for arbitrary a, $([x]e)\,a = e[x \leftarrow a]$. All occurrences of x have been concentrated in a single argument place.

We will see it more directly in section 6, by a remarkable argument due to Böhm, but a simple way to establish completeness it is enough to define the following 'BWICK' combinators in terms of them. This particular set is well-known and probably originally designed to be combinatorially complete. All but one (W) are well known to Haskell programmers, under the names in the left hand column of the following table. The second column gives a possible definition as a λ-term. The third column has the upper case capital letters given as names by Curry or some other authority. The last column contains an aide memoire.

flip	$\rightarrow f\ b\ a$	C	-- swap the arguments of a binary function
(.)	$\rightarrow f\ (a\ b)$	B	-- compose two functions
id	$\rightarrow f$	I	-- identity
const	$\rightarrow f$	K	-- return a function with a single value
	$\rightarrow f\ a\ a$	W	-- this might be called diag, or dupl

For pronouncability, these are the BWICK combinators.

For comparision, here is a similar table for the arithmetical (AMEN) combinators

	$\rightarrow n\ s\ (m\ s\ z)$	A	-- M (m s) (n s)
flip (.)	$\rightarrow n\ (m\ s)$	M	-- 'natural' composition
flip ($)	$\rightarrow n\ m$	E	-- 'natural' application
flip const	$\rightarrow n$	N	-- dispose of an argument

Unfortunately, there seems to no pithy Haskell slang for addition with arbitrary summands. Some sums have a short form though, such as $E + E =$ *flip diag* .

The BWICK combinators are well known (and were perhaps even designed) to be combinatorially complete. (They were introduced in Curry's thesis [Cur30]. With tweaks for efficiency, David Turner used them to implement his seminal function language SASL as described in [Tur79]. These combinators can be viewed as a refinement of the standard 'SKI' combinators, providing for important special *linear* cases of S by using B and C. A definition of S in terms of the arithmetic combinators is not difficult, but there are better and worse ways to explain the idea. A definition

[4] Thanks to Jim Laird for the joke. It is also the first word in the reversal of any prayer.

that isn't blindingly enlightening, is given as a footnote[5], however it is more enlightening to first define S as a binary operator. See section 6 on page 10.

The definitions of the Curry combinators in terms of the arithmetic combinators are quite simple, though it may not be immediately clear where they come from.

$$
\begin{array}{ll}
combC = (\times) \times (\wedge) \wedge (\times) & \text{-- called } \textit{flip} \text{ by fp'ers} \\
combB = (\wedge) \times (\times) \wedge (\times) & \text{-- } (\times) \wedge combC \text{ i.e. composition (.)} \\
combI = evil \wedge () & \text{-- or } () \wedge (), (\wedge) \times (\wedge) \wedge (\times), \text{ inter alia} \\
combK = (\wedge) \times () \wedge (\times) & \text{-- } () \wedge combC \\
combW = (\wedge) \times ((\wedge) + (\wedge)) \wedge (\times) & \text{-- } ((\wedge) + (\wedge)) \wedge combC \\
evil = error\ \texttt{"Unthinkable"} & \text{-- not to be inspected}
\end{array}
$$

We finish this section by deriving these definitions from the definition of each Curry combinator.

The key thing is to start with the C combinator. The main tool is the ζ-law.

C takes a binary function, and transposes or 'flips' its arguments.

$$
\begin{aligned}
C\ a\ b\ c &= a\ c\ b \ \{\text{- by def of } C \text{ -}\} \\
&= b \wedge c \wedge a \ \{\text{- re-express using exponentiation -}\} \\
&= (c \wedge a) \wedge b \wedge (\wedge) \ \{\text{- by def of } (\wedge) \text{ -}\} \\
&= c \wedge (a \times (b \wedge (\wedge))) \ \{\text{- by def of } (\times) \text{ -}\}
\end{aligned}
$$

So,

$$
\begin{aligned}
C\ a\ b &= a \times (b \wedge (\wedge)) \ \{\text{- by } \zeta \text{ -}\} \\
&= (b \wedge (\wedge)) \wedge a \wedge (\times) \ \{\text{- by def of } (\times) \text{ -}\} \\
&= b \wedge ((\wedge) \times a \wedge (\times)) \ \{\text{- def of } (\times) \text{ -}\}
\end{aligned}
$$

So,

$$
\begin{aligned}
C\ a &= (\wedge) \times a \wedge (\times) \ \{\text{- by } \zeta \text{ -}\} \\
&= (a \wedge (\times)) \wedge (\wedge) \wedge (\times) \ \{\text{- by def of } (\times) \text{ -}\} \\
&= a \wedge ((\times) \times (\wedge) \wedge (\times)) \ \{\text{- by def of } (\times) \text{ -}\}
\end{aligned}
$$

So, $C = (\times) \times (\wedge) \wedge (\times)$, or to use exponential notation: $(\times) \times (\wedge)^{(\times)}$, or to use ordinary applicative notation with the *AMEN* combinators: $M\ M\ (M\ E)$.

The gruesome bit is now over. Most other combinators are plain sailing.

B is the transpose of $(\times)$, *ie* $(\times)^C$. From the middle step in the derivation of C's arithmetic form, we therefore have as a by-product

$$B = (\wedge) \times (\times) \wedge (\times) \ \{\text{- alt. } (\wedge) \times (\times)^{(\times)} \text{ or } M\ E\ (M\ M) \text{ -}\}$$

K is the transpose of $()$, *ie* $()^C$. Therefore, just as for B,

$$K = (\wedge) \times () \wedge (\times) \ \{\text{- alt. } (\wedge) \times ()^{(\times)} \text{ or } M\ E\ (M\ N) \text{ -}\}$$

I is the transpose of $(\wedge)$, *ie* $(\wedge)^C$. So we could take the following.

$$I = (\wedge) \times (\wedge) \wedge (\times) \ \{\text{- alt } (\wedge) \times (\wedge)^{(\times)} \text{ or } M\ E\ (M\ E) \text{ -}\}$$

[5] One possible definition of S:

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

But we can also take any of the following:

$$I = C \times C$$
$$I = evil \wedge ()$$

To explain *evil*, it is anything. Its value does not matter, and never needs to be inspected. But it exists.

W Let's first express its transpose W^C, which is quite easy.

$$\begin{aligned} W^C\ a\ b &= b\ a\ a \ \{\text{- by def of } W \text{ -}\} \\ &= a \wedge a \wedge b \ \{\text{- re-express -}\} \\ &= (b \wedge a \wedge (\wedge)) \wedge a \wedge (\wedge) \ \{\text{- by def of } (\wedge), \text{ twice -}\} \\ &= b \wedge a \wedge ((\wedge) + (\wedge)) \ \{\text{- def of } (+) \text{ -}\} \end{aligned}$$

So by the ζ-rule (twice) $W^C = (\wedge) + (\wedge)$, that is $W = ((\wedge) + (\wedge))^C$. Note how this fits with the definitions of K and I.

$$\begin{aligned} K &= ()^C \\ I &= (\wedge)^C \\ W &= ((\wedge) + (\wedge))^C; \ldots \end{aligned}$$

This completes the derivation of the *BWICK* combinators.

4 Further examples: pairing, S, and fixedpoints.

4.1 The pairing combinator (,) and currying.

Consider first pairing as a binary operator: (a, b). In fact, the Church encoding of pairs gives $(a, b) = (a \wedge (\wedge)) \times (b \wedge (\wedge))$, with the first and second projections of p being given by $(K \wedge (\wedge))$ and $(0 \wedge (\wedge))$ respectively.

One can shuffle the parameters around to obtain a purely arithmetical (and linear) expression for the pairing combinator $(,) = (\wedge) \times (\times) \times (\wedge)^{(\times)}$.

Hand in hand with pairing (and indeed exponentiation) comes currying: $curry\ f\ x\ y = f\ (x, y)$. We can immediately see that $curry\ (a\wedge) = a$, so that $(\wedge)\times\ curry = 1$. *ie* *curry* is a multiplicative inverse of the exponentiation combinator. Among its other strange properties, $id \wedge curry$ is the pairing combinator (,).

Two possible expressions for the combinator *curry* are $B\times(,)^{(\times)}$ (which is linear) and $K\times(,)^{(+)}$ (which is non-linear, since it involves both addition and zero). These can be made purely arithmetic by translating B, K and (,) into arithmetic.

4.2 The combinator S, with $S\ a\ b\ c = a\ c\ (b\ c)$.

Consider S first as a binary combinator, written infix with $\circ$ as in $a \circ b$. Its defining equation is $(a \circ b)x = a\,x\,(b\,x)$. As a matter of fact $a \circ b$ can be expressed as $b^{(,)} + a^{(\wedge)}$.

In notation without superscripts, one of many possible definitions of S is this horrible specimen:

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

Using superscript notation, it is possibly even more alarming.

$$(\times)^{1+(\times)+1} \times ((\wedge) \times ((\wedge) + (\wedge))^{(\times)})^{(\wedge)}$$

A straightforward way to define S, is first to define a linear version, $S'\ a\ b\ c\ c' = a\ c\ (b\ c')$. For example, $S'\,a\,b = a \times b^{(\times)}$, so $S' = (\times)^{1+(\times)}$). Finally, define S from S' by diagonalisation: $S\,a\,b = (S'\,a\,b)^W$ or $S = S' \times W^B$. The matter of fact mentioned above arises from more arithmetical considerations, and will be explained in section 6 on page 10 on logarithms.

4.3 Curry and Turing fixed-point combinators.

Among endless variations, two fixed-point combinators of historical significance are Curry's ([CF58, p. 178]) and Turing's [Tur37]. 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. 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.

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

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

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

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

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*.

5 Algebra

In the presence of the ζ-rule,

1. $(+, ())$ forms a monoid.
2. $(\times, 1)$ forms a monoid, where $1 = combI$.
3. pre-multiplication $(a\times)$ distributes over the additive monoid: $a \times (b + c) = a \times b + a \times c$ and $a \times () = ()$.
4. exponentiation $(a\wedge)$ maps the additive monoid to the multiplicative monoid: $a \wedge (b + c) = a \wedge b \times a \wedge c$ and $a \times () = ()$. This was Napier's idea: to avoid the 'slippery' errors that can arise in the multiplicative world by expeditiously working instead on more solid ground, in the additive monoid. [6]
5. exponentiation $(a\wedge)$ maps the multiplicative monoid to the composition monoid $(\cdot, id)$ of unary functions over our arithmetical domain: $a \wedge (b \times c) = (a \wedge b) \wedge c$ and $a \wedge 1 = a$, *ie* $\wedge(b \times c) = (\wedge c) \cdot (\wedge b)$ and $(\wedge 1) = id$.

[6] Napier famously wrote in the preface to his "Mirifici logarithmorum canonis descriptio" (1614):

> Since nothing is more tedious, fellow mathematicians, in the practice of the mathematical arts, than the great delays suffered in the tedium of lengthy multiplications and divisions, the finding of ratios, and in the extraction of square and cube roots and in which not only is there the time delay to be considered, but also the annoyance of the many slippery errors that can arise: I had therefore been turning over in my mind, by what sure and expeditious art, I might be able to improve upon these said difficulties.

He wanted to move between the 'slippery' multiplicative world of geometric progressions and the 'expeditious' additive world of arithmetic progressions; to multiply, divide and take roots by adding, subtracting and dividing.

Addition and multiplication are not in general commutative, there need be no subtraction (or division), and post-multiplication need not distribute over sums. I have heard such structures called a 'near-semiring' (with unit). As rings go, this is a bit enfeebled. However, rings do not generally have exponentiation, which makes up for the feeble structure on $+$ and $\times$. Personally, I call the algebraic structure (even without the "funny" numbers $(\wedge), (\times), (+)$, but only 0 and $1 = 0^0 = id$ as constants) an 'elementary arithmetic', or the algebra of elementary arithmetic.

The laws of $(+), (\dot{)}, (\times), 1, 0, (\wedge)$ above are within a gnat's whisker of the basic laws of ordinal arithmetic with (with simple non-commutative addition and multiplication). However the coincidence is not exact. In an ordered world, when a is "numerical", which is to say a "true" number, the following laws obtain.

$$1 \wedge a = 1$$
$$\dot{)} \times a = \dot{)}$$

Matters are different when a is an arbitrary combinatory expression. For example, when $a = K$, 1^K is the constant function whose value is the identity. To find counterexamples to equations, our basic tool is the assumption that no constant function equals the identity function, or equivalently $0 \neq 1$. From this other distinctions follow: $0 \neq K$, $1 \neq K$, $2 \neq K$, *etc.* To distinguish 1^a from 1, clearly, a must not be numerical, but something weird. We can choose $a = K$, since 1^K is ζ-equivalent to 0, which is distinct from 1. A counterexample to the second is a little more circuitous. Again, a must be something non-numerical. Let us try exponentiation itself: $(\wedge)$. Applying both sides to successive variables x then y, the left hand side becomes 1^y, and the righthand side becomes y. So if y is K, the lhs is 0 and the right is K. But $0 \neq K$, since if $0 = K$ then $1 = 1^0 = 1^K = 0$. So exponentiation is one counterexample to the second law.

An intriguing feature of this arithmetical world is that interesting laws (particularly for logarithms) sometimes hold with greater generality than one might expect. [7] Life is not always so serious though, and we can simply enjoy the sheer weirdness of things. For example, call functions f and g converse if $f = g^C$. Converse functions share the same diagonal. It follows immediately that C is not its own converse, *ie* not commutative. However, it is a square-root of identity distinct from identity. Maybe we could consider Gaussian 'numbers', that have with an imaginary part with C as a factor.

Axioms as closed equations As an exercise in pointlessness, one can express the axioms of this structure the monoid laws, the distribution laws and so on, in the form of equations between closed expressions, albeit gigantic and inscrutable. The result is not particularly enlightening, but it calls to mind Curry's finite axiomatisation of the closed consequences of the ζ-rule.

Exponentiality The rule of exponentiality ζ says that if things t are the same as exponentials x^t where the base x is a fresh variable, then they're just the same. Everything is a function, and no more than a function. It is entirely indispensible.

A remarkable discovery of Curry's was that finitely many instances of ζ suffice to axiomatise all consequences of ζ. A combinatory algebra that satisfies Curry's equations is (I think, perhaps wrongly) known as an extensional combinatory algebra. Quibbles can arise about whether the term 'extensional' is appropriate, as what is meant is schematic equality as legitimated by the ζ-rule, but whether or not the terminology is appropriate, it is in widespread use. One can read about these algebraic axioms in Hindley and Seldin's book [HS86, Ch. 8], and about more full-blooded, genuine or semantical, forms of extensionality in Selinger's paper, [Sel02].

Curry's equations were written (TODO: check) using the S and K combinators, but they can be written using any combinatorially complete set. A key rôle is played by the K and S combinators, the former being used to throw away an unwanted argument, and the latter to feed arguments to both parts of an expression of exponential form. (Expressions of the form t^K are in facts closed under addition, and indeed multiplication by arbitrary factors, as we shall see shortly.)

[7] For example $\log_x(\alpha^b) = (\log_x \alpha) \times b$, merely in case b does not contain x. It need not be numerical.

In an arithmetical setting, these equations between closed terms can be obtained by 'pointlessly' removing variables from the following laws. (I allow myself to go a bit rough-shod across some fearsome-subtle shenanigans.)

$$\begin{array}{lll}
(\wedge)^K \triangleright x \triangleright y & = y \triangleright x & \\
(\times)^K \triangleright x \triangleright y \triangleright w & = y \triangleright (x \triangleright w) & \\
(+)^K \triangleright x \triangleright y \triangleright w \triangleright z & = y \triangleright w \triangleright (x \triangleright w \triangleright z) & \\
()^K \triangleright x \triangleright y & = y & \\
(x \wedge y)^K & = x^K \triangleleft y^K & \text{homomorphism} \\
x^K \triangleright 1 & = x & \text{universality}
\end{array}$$

Here $\triangleright$ is used as an infix form of the S combinator ; $\triangleleft$ is an infix form of the converse of $\triangleright$; and ^{K} is a superscript form of the K combinator. These have quite pithy arithmetical expressions, given in section 6.

Note that beside $(x \wedge y)^K = x^K \triangleleft y^K$, we also have the laws $(x \times y)^K = x^K + y^K$, and $1^K = ()$. So a lot of structure is mapped around by ^{K}. What is important is that the defining equations for the combinators are preserved.

As for the 'universality' equation, it is still a bit mysterious to me, but it seems to say that the ^{K} homomorphism has some uniqueness property. Note: 1 is not in the image of ^{K}. [8]

TODO: Freyd [Fre89]; Selinger [Sel02]; Statman [Sta14].

Ideals, more or less. A paper by Böhm ([Böh82]) is particularly concerned with the ring-like structure described above. In it, Böhm introduces a notion that is clearly inspired by the notion of an ideal in a ring, A class is called by Böhm a "notion of zero" if it contains $()$, is closed under addition, and for arbitrary a is closed under both $(a\times)$ and $(\times a)$.

One example: $Z =$ the class of terms that can be put in the form a^K, or equivalently $() \times (a\wedge)$ – these represent constant functions, that as powers of any base have the same value. All 'constants' are ζ-equivalent to terms of the form a^K. [9] The following are straightforward to prove with the ζ-rule.

- Closure under addition:
$$\begin{array}{ll} () & = 1^K \\ a^K + b^K & = (a \times b)^K \end{array}$$
- Closure under scaling by arbitrary factors on left and right:
$$\begin{array}{l} b \times (a^K) = a^K \\ (a^K) \times b = (a \wedge b)^K \end{array}$$

  Because of the last scaling law, the class $a^K = () \times (a\wedge)$ already includes the class $() \times (a_1\wedge) \times \cdots \times (a_k\wedge)$.

What does one do with an ideal? Quotienting. So one considers a and b to be 'equal' if a and b differ only by an element of the ideal (*ie* by 'zero'), that is by addition of a 'constant', *ie* $a = b + c^K$. Equivalently, $a = b \times c^B$, which seems to mean that a and b are the same up to scaling by a 'constant' factor.

Perhaps in pursuit of duality, Böhm introduces also the term "notion of infinity" for a set of terms closed under $(a+)$ and $(+a)$ for any a. One example is the set I of terms of the form $a \times K$.

[8] Curry's equations remind me slightly of the axioms for Haskell's applicative functors, if only notationally. With applicative functors (discovered by Conor McBride, among others), it seems that one cares about the linear combinators only, namely B, I and (\$). (One could use $(\wedge)$,$(\times)$.) There is no place, one would guess, for S, K or W among the laws for applicative functors. . A more striking difference is that where Curry has the law $(x^K) \triangleright 1 = x$, Conor has an interchange law (translating it into our more comely notation) $(y\wedge)^K \triangleright x = x \triangleright y^K$. This happens to be true; but more is true. In fact, $(\wedge)^K \triangleright x \triangleright y = y \triangleright x$ is one of the axioms for the linear combinators. For an applicative functor one needs this only when x has the form $\_^K$.

[9] They play a major rôle in Curry's investigation of the ζ-law at section 5.

- Closure under arbitrary addition on left and right:

$$b + (a \times K) = a \times K$$
$$(a \times K) + b = (a \triangleleft b) \times K$$

(Here $\triangleleft$ is infix notation for the converse of the binary 'S' combinator, that was introduced in connection the Curry equations.

Sheer duality might lead one to conjecture that this particular 'notion of infinity' $a \times K$ might be closed under multiplication. It isn't, nor, to be frank is this form of expression of much obvious interest. It might be that some other forms of expression (eg $a + K$, a^B, $(a\wedge) = a^{(\wedge)}$, ...) have interesting closure properties with respect to addition, scalar multiplication, and so on.

Böhm's paper teems with monoidal and cartesian structures: the monoid of lists under concatenation, the monoid $(\times, 1)$ of endofunctions on a set under composition, the monoid $(+, 0)$ of endofunctions of endofunctions under pointwise-lifted composition, products, pairs and their projections. In the next section, we make use of his apparatus to extend Napier's laws of logarithms.

6 Logarithms

One can daydream of using λogarithmic notation for lambda-abstraction, with $a \wedge (log\_x\ e) = e\ [x \leftarrow a]$. Here the 'base' is a bound variable, but no matter: one obtains Napier's laws of logarithms.[10] For example, whether or not the base occurs in both factors of a product, the product is turned into a sum. If a, b and c are expressions such that c contains no occurrences of x, then we have the following equations, once familiar to schoolchildren.

$$log\_x\ (a \times b) = (log\_x\ a) + (log\_x\ b)$$
$$log\_x\ 1 = (\dot{})$$
$$log\_x\ (a \wedge c) = (log\_x\ a) \times c$$
$$log\_x\ x = 1$$

The last equation is not difficult to remember, and the others can be summed up in the following formula for the logarithm of a product of constant powers.

$$log\_x\ (b_1 \wedge c_1 \times ... \times b_n \wedge c_n)$$
$$= (log\_x\ b_1) \times c_1 + ... + (log\_x\ b_n) \times c_n$$

It should be stressed that this equation holds regardless of whether the coefficients c are numerical.

Now let us consider 'linear' logarithms, where we care about how many times the 'base' x occurs in e. These obey many delightful arithmetico-combinatorial laws, that can be figured out on the edge of a newspaper. The following are easily verified, where $a, b, b_1, \ldots b_k$ are expressions that do not contain any occurrences of the variable x.

$$log\_x\ x = 1$$
$$log\_x\ (a\ x\ b) = log\_x\ (b \wedge x \wedge a)$$
$$= log\_x\ ((x \wedge a) \wedge (b\wedge))$$
$$= a \times (b\wedge)$$
$$log\_x\ (a\ x\ b_1 .. b_k) = a \times (b_1\wedge) \times .. \times (b_k\wedge)$$
$$log\_x\ (a\ x\ x) = ((\wedge) + (\wedge)) \times (a\wedge)$$
$$log\_x\ a = (\dot{}) \times (a\wedge) = a \wedge K$$

In fact, any linear abstraction [11] is a product (ie. a composite) of factors that each look like

$$a \times (b_1\wedge) \times ... \times (b_n\wedge) \quad .$$

[10] It has to be said: but not the laws that deal with change of base.

[11] ie, in which the bound variable occurs exactly once, so not like the left hand terms in the last two equations.

In other words, the following is a normal form for logarithms to a linear base.

$$\prod_{i:[1,m]} (a_i \times \prod_{j:[1,n_i]} b_{ij}^{(\wedge)})$$

The logarithm of the last expression to a fresh (not necessarily linear) base y is then easily attained:

$$\sum_{i:[1,m]} (\log_y a_i + \sum_{j:[1,n_i]} \log_y(b_{ij}) \times (\wedge)) \ .$$

Naperian logarithms deal with non-linear abstraction, but we only know how to deal with multiplicative structure and constant powers. Can we take logarithms of terms of general exponential form, where the 'base' occurs in both the exponent and the iterand? What about logarithms of sums? Böhm [Böh79][12] showed how this works. He was well aware of the formal similarity between λ-abstraction and taking logarithms. In fact, he devised an innovative form of bracket abstraction, by extending the logarithm operation to arbitrary arithmetical expressions, yet preserving its Naperian core.

To take the logarithm of a term of *general exponential form* (where the exponent need not be constant), we re-express it as a product of constant powers. The trick is to use Church's pairing combinator $(,) = \lambda a\ b\ c \to c\ a\ b$ [13]. This satisfies

$$\begin{aligned}(a,b) &= a^{(\wedge)} \times b^{(\wedge)}\\ K \wedge (a,b) &= a\\ (\,) \wedge (a,b) &= b\end{aligned}$$

Note that $(a \wedge b)\ c = b \wedge (a,c)$. So any exponential term $a \wedge b$ can be expressed as a product of constant powers.

$$a \wedge b = (\lambda c \to b \wedge (a,c)) = (a,) \times (b\wedge) = (a \wedge (,)) \times (b \wedge (\wedge))$$

Napier gave us the logarithms of such quantities. Applying his technique, we alchemise the (slippery) product monoid into an (expeditious) sum monoid. We have thus an 'exponential' doppelganger S^C of the S combinator, referred to previously as $\triangleright$ in section 5 in connection with Curry's treatment of the ζ-law. Using the infix notation of that section

$$S^C\, a\, b \ = \ a \lhd b \ = \ a \times (,) + b \times (\wedge)$$

This is what we need to extend logarithms to general exponential form, where the exponent is not constant.

$$\log_x(a \wedge b) \quad = (\log_x a) \lhd (\log_x b)$$

Fortunately, this agrees with the Naperian law when b has no occurrence of x.

How about logarithms of *additive form*? We already know how to do this by brute force, since a sum can be expressed as an exponential to an exponential. Is there something more dainty?

The trick is currying. Let *curry* be some closed expression such that $curry\ f\ x\ y = f\ (x,y)$ [14]. It is direct that $curry\ (a^{(\wedge)}) = a$, and so $curry\ (a^{(\wedge)})\, x = x^a$. But much more obtains:

$$curry(a^{(\wedge)} + b^{(\wedge)})x = x^a + x^b \ .$$

(We even have $curry(a^{(\wedge)} \times b)x = x^a \times b$, regardless of whether b is numerical.) This gives Böhm's amazing formula for the logarithm of a sum:

$$\log_x(a+b) = curry((\log_x a)^{(\wedge)} + (\log_x b)^{(\wedge)})$$

Finally, as you may perhaps guess from looking at the logarithm of a binary sum, the logarithm of zero, to any base, is Curry'd naught, $(\,)\wedge K$. This 'eats' anything, in the sense that $a + (\,)^K = (\,)^K$. Curry'd one, on the other hand, is the pairing combinator $(,)$ For convenience, I've put the laws of Böhmian logarithms in a table in figure 1 on the following page.

[12] I am very grateful to Roger Hindley for a copy of this almost unobtainable paper.

[13] For example: $(\wedge) \times (\times) \times (\wedge)^{(\times)}$. This we mentioned before in section 4.1 on page 6.

[14] Two examples are $B \times (,)^{(\times)}$ and $K \times (,)^{(+)}$. The second one comes from expressing $curry\ f\ x$ as $x^{(,)} \times x^{f^K}$, then taking logarithms to the base x to get $curry\ f = (,) + f^K$

$$
\begin{array}{lll}
\text{pairing apparatus: } (a,b) = a^{(\wedge)} \times b^{(\wedge)} & & ;\ curry\, f\, x\, y = f(x,y) \\
\log_x(\prod_i a_i) & = \sum_i (\log_x a_i) & ;\ \log_x 1 = (\dot{}) \\
\log_x(\sum_i a_i) & = curry(\sum_i (\log_x a_i)^{(\wedge)}) & ;\ \log_x(\dot{}) = curry(\dot{}) = (\dot{})^K \\
\log_x(a \wedge b) & = (\log_x a) \times (,) + (\log_x b) \times (\wedge) & \\
\log_x(a,b) & = (\log_x a) \times (\wedge) + (\log_x b) \times (\wedge) &
\end{array}
$$

Figure 1: Table of Böhm's λogarithms

7 Calculators and combinators

In the dawn of functional programming, there were some working implementations of combinator machines, using a complete set of combinators such as SK, $BWCK$, or a variant thereof[15] . If we count both paper and vapour implementations, there were about a dozen. One can daydream about implementing 'in hardware' a machine taking $AMEN$ as a basis. This would be an arithmetical calculator indeed [16], handy to have in a pocket, or on a wrist. [17]

To be remotely sensible, we should bake-into the calculator lots of binary combinators other than $(\wedge)$,$(\times)$,$(+)$ and $(\dot{})$. Many familiar combinators show up everywhere, and deserve names. [18]

It makes no sense to ignore the associativity of addition and multiplication. So in such an arithmetical machine one might use 'polyadic' forms of $\sum$ and $\prod$, taking finite lists as arguments.

$$
\begin{array}{ll}
\sum[a_1,..,a_k] & = a_1 + ... + a_k \\
\prod[a_1,..,a_k] & = a_1 \times ... \times a_k
\end{array}
$$

The apparatus for lists is curious: the Church encoding of the constructor to prefix a given item x to a list l is $l + a^{(\wedge)}$. the Church encoding of the empty list is $(\dot{})$. Once one has lists, one should consider infinite streams, and co-lists. How might $\sum$ and $\prod$ work with infinite streams? One answer might be that the sum of a stream is another stream, consisting of the accumulated finite partial sums, starting with 0. The product might analogically consist of the accumulated finite partial products, starting with 1. The idea is that a stream is turned into a stream of lists, being the finite prefixes of the stream.

If one extends the arithmetical machine with streams, how should the operators behave when one or other of their arguments are streams? All I can say is that it is likely that, the operations $(a+)$, $(a\times)$, $(a\wedge)$ should distribute over streams. The binary arithmetical operators are all continuous in their right-hand arguments.

[15] For example: [Tur79] [Sto85] [Sto83]. For implementing functional languages, attention quickly switched to supercombinators extracted from a particular program, but research on complete sets of combinators still continues, if sporadically. TODO: cite some

[16] In its ancestry there would be those cash-register-resembling desktop 'adding machines' around in the 1960's, and the pocket calculators of the 80's.

[17] Lacking a silicon foundry, I wrote a small program to evaluate arithmetical expressions built up from variables with constructors for addition, multiplication, exponentiation, naught, and various constants. This calculator uses algebraic laws to rewrite expressions to arithmetical normal form. I've found the calculator useful to avoid mistakes when exploring and experimenting with Naperian arithmetic. Not only can one observe what combinators do, but it is also possible to check whether expressions are equal mod-ζ.

[18] A few examples:

(i)	$(\sim)$	$= C = flip$
(ii)	$(,)$	for the pairing combinator
(iii)	$(\cdot), B$	$= (\times)^{(\sim)}$
(iv)	$(+)^{(\sim)}$	
(v)	K	$0^{(\sim)}$
(vi)	$(\$)$	$(\wedge)^{(\sim)}$
(vii)	S	and its flip $S^{(\sim)}$

One observation is that $(\wedge)$, $(\sim)$, $(,)$ and their flips $(\wedge)^{(\sim)}$, $(\sim)^{(\sim)}$, $(,)^{(\sim)}$ are a handy 6-pack of combinators to permute the top 3 positions of the arithmetical stack. Permutations are linear. The combinators $(\dot{})$ and $K = (\dot{})^{(\sim)}$ are affine, or cancelative, *ie* strip some entries from the stack. The combinators $(+)$, W, $S = (\triangleright)$, $S^C = (\triangleleft)$ (among many others, for example those for operating on finite lists) introduce sharing, contraction, diagonalisation, or other duplication: we cannot perform their reduction in constant space, *ie* during garbage collection.

To take this technological fantasy even further, consider graph rewriting with 'garbage collection', or recycling of otherwise wasted storage. The fragment MEI consisting of (exponentiation, multiplication and unity) captures linear abstraction, and MEN captures affine abstraction. It may be an advantage to easily recognise such features of an expression. For one thing, affine functions never introduce sharing, though they may discard garbage. The possibility of sharing requires constant care when rewriting graphs, and administering their storage. In graph reduction, the garbage collector (and not just the mutator) can safely and beneficially perform certain 'affine' forms of calculation – *eg* projections from tuples. See [Wad87].

In some sense, an arithmetical calculator (a combinator machine for the arithmetical combinators) would spend most of its time in multiplicative (affine) mode, performing garbage collection. From time to time it would perform an addition, introducing sharing, and thereby provoking more 'multiplicative' work for the garbage collector. Sums and not products would be the major headache for the designer of the AMEN machine. [19]

8 Origins and heresies

Slight variants of the arithmetical combinators abound in the literature. Some examples are in Rosenbloom ([Ros50, sec. 3.4] Stenlund ([Ste72, sec. 2.4] and Burge ([Bur75, sec. 1.8] They were surely defined by Church, and – to stretch a point – Wittgenstein (of whom, more anon). But in some sense we all know the definitions, if we understand the concept of iteration at all. They are in our brain-stems or DNA.

Of the authors mentioned above, as far as I know, only Stenlund notes the combinatorial completeness of the arithmetical combinators. In his thesis from 1972 he attributes the result to F. B. Fitch. He thinks he was made aware of it in correspondence with some of Fitch's students. Apparently independently, Böhm was by 1977 aware of the combinatorial completeness ([Böh79],[Böh81] – written in 1977). As already mentioned, publication [Böh81] (and presumably [Böh79]) notes the analogy between λ-abstraction and logarithms referred to in section 6, and in fact wrote the logarithm function as λogarithm.

Repulsive variants of arithmetical notation: Many definitions of arithmetical combinators get the argument order slightly wrong, in this author's opinion. They make the multiplication combinator the same as the B composition combinator. This is a mistake. It should take arguments the other way round. The business end of an arithmetical operation should always be the second operand. This mistake affects in turn the argument order for addition, which is taken to be a pointwise-lifting of composition.

To establish the scriptural correctness of my version, allow me to assert that my definitions (at least for $\times$) can be read into Wittgenstein's Tractatus (around 6.02 and 6.241). The argument order has also the authority of Cantor according to whom the alternative is 'repulsive' – abstoßende. In fact Cantor started off using repulsive argument order, but soon thought better of it. (This is according to Michael Potter, [Pot90, p. 120].) In any case, the notation on which Cantor settled coheres attractively with exponential notation for iteration, if we remember that multiplication is composition in reverse:

$$\begin{array}{lll} f^{\alpha\times\beta} & = (f^{\alpha})^{\beta}, & f^{1} = f, \\ f^{\alpha+\beta} & = f^{\beta}\cdot f^{\alpha} = f^{\alpha}\times f^{\beta}, & f^{0} = 1\,. \end{array}$$

We leave the last word to Napier, from his note at the end of chapter 2 of book 1 of his *Descriptio* :

> Indeed I await the judgment and censure of the learned men concerning these tables, before advancing the rest to be published, perhaps rashly, to be examined in the light of envious disparagement.

[19] This was pointed out to me by Conor McBride.

References

[Bar84] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[Böh79] Corrado Böhm. Un modèle arithmétique des termes de la logique combinatoire. In B. Robinet, editor, *Lambda Calcul et Sémantique Formelle des Langages de Programmation. Actes de la Sixième Ecole de Printemps d'Informatique Theorique, La Châtre, 1978*, pages 97–108. LITP et ENSTA, Paris, 1979.

[Böh81] Corrado Böhm. Logic and computers: Combinatory logic as extension of elementary number theory. In E. Agazzi, editor, *Modern logic – a survey*, pages 297–309. Reidel, Dordrecht, 1981.

[Böh82] Corrado Böhm. Combinatory foundation of functional programming. In *LFP '82: Proceedings of the 1982 ACM symposium on LISP and functional programming*, pages 29–36. ACM Press, New York, NY, USA, 1982.

[Bur75] William H. Burge. *Recursive Programming Techniques*. The Systems Programming Series. Addison-Wesley, Reading, Massachusetts, 1975.

[CF58] Haskell B. Curry and Robert Feys. *Combinatory Logic*, volume 1. North Holland, 1958. Second edition, 1968.

[Cur30] Haskell B. Curry. Grundlagen der kombinatorischen logik. *Amer. J. Math.*, 52:509–536, 789–834, 1930.

[Fre89] Peter Freyd. Combinators. In J. W. Gray and A. Scedrov, editors, *Categories in Computer Science and Logic: Proc. of the Joint Summer Research Conference*, pages 63–66. American Mathematical Society, Providence, RI, 1989.

[HS86] J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-calculus*, volume 1 of *London Mathematical Society – Student Texts*. Cambridge University Press, 1986.

[Pot90] Michael D. Potter. *Sets, An Introduction*. Clarendon Press, Oxford, New York, Tokyo, 1990.

[Ros50] Paul C. Rosenbloom. *The elements of mathematical logic*. Dover, New York, 1950.

[Sel02] Peter Selinger. The lambda calculus is algebraic. *Journal of Functional Programming*, 12(6):549–566, 2002.

[Sta14] Rick Statman. Near semi-rings and lambda calculus. In *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 410–424, 2014.

[Ste72] Sören Stenlund. *Combinators, λ-Terms and Proof Theory*, volume 42 of *Synthese Library*. Reidel, Dordrecht, 1972.

[Sto83] William Stoye. The SKIM microprogrammer's guide. Technical Report UCAM-CL-TR-40, University of Cambridge, Computer Laboratory, October 1983.

[Sto85] William Stoye. The implementation of functional languages using custom hardware. Technical Report UCAM-CL-TR-81, University of Cambridge, Computer Laboratory, December 1985.

[Tur37] Alan M. Turing. The p-function in $\lambda - K$-conversion. *J. Symb. Log.*, 2(4):164–164, December 1937.

[Tur79] David A. Turner. A new implementation technique for applicative languages. *Softw., Pract. Exper.*, 9(1):31–49, 1979.

[Wad87] Philip L. Wadler. Fixing some space leaks with a garbage collector. *Software Practice and Experience*, 17(9):595–609, 1987.