The AMEN architecture.

Peter Hancock

Abandoned draft of Jan 99

Abstract

There are many combinatorially complete sets of combinators, or 'instruction sets' to which the λ-calculus can be compiled. The most famous are perhaps $\{S, K\}$ and $\{B, C, K, W\}$. Several authors have observed that there is a set of 'arithmetical' combinators $\{A, M, E, N\}$ arising from the Church numerals, which is also combinatorially complete. However their sets of combinators are not absolutely *perfect*. The perfect set of combinators ($\{(+), (\times), (\hat{}), 0\}$ in section-notation) is defined herein.

There is certain 4-instruction machine, with opcodes $\{\texttt{Add}, \texttt{Mul}, \texttt{Exp}, \texttt{Nop}\}$, which is surely the last word in computer architecture. It may be that there is more to this machine than entertainment, as the garbage collector can execute the whole instruction set except addition.

I am thinking of rewriting this under the heading 'exponential calculus', and drying out the humour. Meanwhile, I am rather interested in pursuing a certain idea which is roughly explained in the penultimate section (section 5 on page 10 called 'streams and dreams'. The idea is to express the simply typed λ-calculus with arithmetic combinators, and to extend the type structure with a unary type constructor for infinite streams, with associated constants. The arithmetic operators extend in a pointwise fashion to streams. Then one adds an operator taking streams to streams, which expresses a certain kind of 'cumulative composition', and is closely related to the recursion constant in Gödel's T. I dare to hope that this will lead to a new kind of ordinal analysis for this system.

1 Introduction

Suppose we write application backwards, with the function after the argument, as mathematicians sometimes do. The notation is usually called 'algebraic notation', but I will refer to it as 'exponential notation', for reasons that will soon be clear. Perhaps it is congenial if you visualise data as starting on the left and flowing through successive functions towards the right.

Let us use $\ldots \,\hat{}\, \ldots$ as an infix notation for application written backwards. (Juxtaposition $(\ldots\ \ldots)$ is more often used in real life, but in most functional languages, juxtaposition is already in use for the reverse operator.) The symbol is chosen to evoke 'exponential' associations, as we shall shortly introduce the familiar arithmetical operators $\times$, 1, $+$, 0. We take all binary operators to be right associative, so that for example $a \,\hat{}\, b \,\hat{}\, c$ is bracketed $a \,\hat{}\, (b \,\hat{}\, c)$, corresponding to the left associativity of the normal juxtaposition operator.

In connection with the normal application notation it is supremely natural to introduce the normal composition notation, and something like 1 for the identity. In connection with exponential notation, for mirror-image reasons it is natural to introduce a multiplication operator $\ldots \times \ldots$, which is the reverse of composition.

$$\begin{array}{lcl|lcl} (f \cdot g)\ a & = & f\ (g\ a) & a \,\hat{}\, (g \times f) & = & (a \,\hat{}\, g) \,\hat{}\, f \\ 1\ a & = & a & a \,\hat{}\, 1 & = & a \end{array}$$

What is more, on the arithmetical side there is good use for the $\ldots + \ldots$ operator and 0. I do not know of any common notation for their counterparts in the normal column, so I shall use the operator $\ldots \tilde{+} \ldots$, and constant $\tilde{0}$:

$$\begin{array}{lcl|lcl} (f \,\tilde{+}\, g)\ a & = & f\ a \cdot g\ a & a \,\hat{}\, (g + f) & = & a \,\hat{}\, g \times a \,\hat{}\, f \\ \tilde{0}\ a & = & 1 & a \,\hat{}\, 0 & = & 1 \end{array}$$

Let us take the 'ζ' rule for granted:

$$\frac{x \,\hat{}\, a = x \,\hat{}\, b}{a = b} \text{ where } x \text{ is fresh to } a \text{ and } b$$

This is a harmless form of mild extensionality, or perhaps exponentiality. Using it, we can prove from the definitions the following algebraic facts:

- $(\ldots \times \ldots, 1)$ forms a monoid: i.e. $\ldots \times \ldots$ is associative, and has 1 as a left and right neutral element.
- $(\ldots + \ldots, 0)$ forms a monoid.
- $\ldots \times \ldots$ distributes[1] over $\ldots + \ldots$ from the left:

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

Curiously, these laws[2] are (roughly[3]) the main ones that continue to hold when

[1]is this how one uses 'distributes'?

[2]I have seen such a structure called a 'half-ring'. See [5]. There may be some categorical notion too – some not-necessarily-commutative isotope of a bicartesian closed category?

[3]We do not have $1 \,\hat{}\, a = 1$, nor $0 \times a = 0$.

```
-- 'take over' operators
import Prelude hiding ((*),(^),(+))

-- usual precedence
infixr 8 ^
infixr 7 *
infixr 6 +

m + n       = \f -> f ^ m * f ^ n
m * n       = \f -> (f ^ m) ^ n
m ^ n       = n m
nop f z     = z
```

Figure 1: Haskell code for the arithmetical combinators.

arithmetic is extended to arbitrary orderings (well-founded or not). Other laws that hold only in finite arithmetic fail. For example, the commutativity of multiplication fails because it says that order of composition does not matter. (I have no explanation of this coincidence.)

It is simple to translate arithmetical notation into a Haskell style functional program, without using recursion. The code is in figure 1 on page 3. [4] However, by doing so, we have given ourselves the possibility of using `(+)`, `(*)`, and `(^)` as if they were 'numbers', that represent the arithmetical operations in the sense that the following laws hold:

$$\begin{array}{rcl} b \mathbin{\hat{}} a \mathbin{\hat{}} (\hat{}) & = & a \mathbin{\hat{}} b \\ b \mathbin{\hat{}} a \mathbin{\hat{}} (\times) & = & a \times b \\ b \mathbin{\hat{}} a \mathbin{\hat{}} (+) & = & a + b \end{array}$$

These are the 'arithmetical combinators' of Stenlund [25] page 21, Schwichtenberg [28] page 19, Burge [2] page 35, Rosenbloom [22] page 122, Church [3] page 10 (??), (Barendregt?, Fitch?, Shoenfinkel?, Curry?, Kleene?, Turing? ..) . , except for a quibble. These gentry define the multiplication combinator to be the transpose of $(\times)$, namely $(\cdot)$, which equals the classic 'B' combinator. Their addition combinator is correspondingly different. (It is $\ldots \tilde{+} \ldots$.) My definitions have the authority of Cantor, who appreciated exponential notation, and said (after working with them for a while) that laws such as:

$$a \mathbin{\hat{}} (f \times g) = (a \mathbin{\hat{}} g) \mathbin{\hat{}} f$$

were 'repulsive' (*abstoßende* in German) [5] [6].

[4] Perhaps it is an anomaly in Haskell that one can 'take over' the arithmetical operators for deviant purposes, but that one cannot 'take over' `0` and the other digits in the same way – isn't there some way that could work? Leslie Lamport has figured out something analogous for his language 'TLA'. See `http://www.research.digital.com/SRC/tla/tla.html`, somewhere.

[5] I learnt this from page 120 of Potter's [21].

[6] I suggest that a correct definition of the multiplication combinator can be read into

2 Completeness

That the arithmetical combinators are functionally complete, means that we can introduce λ-abstraction as a façon-de-parler, and compile untyped lambda terms into equivalent applicative terms ('code') with only the arithmetical combinators. Because of the arithmetical setting, it is natural to use logarithmic notation '$\log_x b$' instead of '$\lambda x.b$' .)

To compile into arithmetical combinators, first consider *affine* λ-abstraction $\log_x b$, where the bound variable x may have at most one occurrence in the body b. The following cases are typical. I have used 'section notation', replacing $a \mathbin{\hat{}} (\hat{})$ by $(a\hat{})$, etc. .

- $\log_x x = 1$. (1 can be defined as $\mathsf{Junk} \mathbin{\hat{}} 0$, where Junk is anything, eg 0.)
- $\log_x(x \mathbin{\hat{}} f) = f$, if x does not occur in f. (The 'η' rule.)
- $\log_x(a \mathbin{\hat{}} x) = \log_x(x \mathbin{\hat{}} (a\hat{})) = (a\hat{})$, if x does not occur in a.
- $\log_x(a_k \mathbin{\hat{}} \cdots \mathbin{\hat{}} a_1 \mathbin{\hat{}} M \mathbin{\hat{}} a) = (\log_x M) \times a \times (a_1\hat{}) \times \cdots \times (a_k\hat{})$, if x does not occur in $a_1, \ldots, a_k, a$.
- $\log_x a = 0 \times (a\hat{})$, if x does not occur in a.

Here is arithmetical code for the classical (Shönfinkel[7]) combinators C, B and K.

$$\begin{array}{rcl} C & = & (\times) \times ((\hat{})\times) \\ B & = & (\times) \mathbin{\hat{}} C \\ K & = & 0 \mathbin{\hat{}} C \end{array}$$

Since C plays such a key rôle, it may be worth noting that

$$\begin{array}{rcl} a\hat{}C & = & (\hat{}) \times (a\times) \\ b\hat{}a\hat{}C & = & a \times (b\hat{}) \quad . \end{array}$$

As for the W combinator[8], consider first it's transpose:

$$W' = \log_y(\log_x(y \mathbin{\hat{}} y \mathbin{\hat{}} x))$$

Using the laws above for linear logarithms, one has $W' = \log_y((y\hat{}) \times (y\hat{}))$. Then, by evident properties of (non-linear) logarithms,

$$W' = (\hat{}) + (\hat{}) \quad ,$$

Wittgenstein's Tractatus [33], at or around remarks 6.241, 6.02, and 6.021. Wittgenstein used exponential notation in connection with iteration of operations, and in the 1910's thought of numbers à la Church (to put it anachronistically).

[7] He called them T, Z and C respectively.

[8] Judging by Stenlund [25] p 22, this definition is (essentially) due to F. B. Fitch. For amusement's sake, the W combinator can also be coded (dreadfully) as follows.

$$W = (((\hat{}) \times (0\times))+) \times (\times) \times (0 \mathbin{\hat{}} ((\hat{}) \times (\hat{})))$$

(The odds are not good that I have transcribed this correctly.)

so

$$W = ((\hat{}) + (\hat{}))\ \hat{}\ C\ .$$

You can always compile logarithmic expressions to arithmetical code by 'brute force', meaning first translation to affine form by introducing W combinators and then compilation of the resulting affine term. (In particular cases you can sometimes do better.)

Here are some miscellaneous laws.

- $\log_x(a \times M) = (\log_x M) \times (a\times)$ if x does not occur in a.
- $\log_x(M \times a) = (\log_x M) \times (\times) \times (a\hat{})$ if x does not occur in a.
- $\log_x(M \times N) = (\log_x M) + (\log_x N)$.
- $\log_x 1 = 0$.
- $\log_x(b + a \times x) = (a\times) \times (b+)$ if x does not occur in a or b.
- $\log_x(b + x \times a) = (\times) \times (a\hat{}) \times (b+)$ if x does not occur in a or b.
- $\log_x(a \times x + b) = (a\times) \times (+) \times (b\hat{})$ if x does not occur in a or b.
- $\log_x(x \times a + b) = (\times) \times (a\hat{}) \times (+) \times (b\hat{})$ if x does not occur in a or b.
- $(1\times) = (0+) = (\times)\ \hat{}\ (1\hat{}) = (+)\ \hat{}\ (0\hat{}) = 1$.
- ...

Another miscellaneous thing: the usual pairing $(a, b) = \log_c(c\ a\ b)$ comes out as $(a\hat{}) \times (b\hat{})$, with projections $(K\hat{})$ and $(0\hat{})$, which seems to make a weird kind of sense.

It is interesting to speculate about inverses of the arithmetical operations. After all, the logarithm is in a sense an inverse for exponentiation. Can we somehow express subtraction, and division? As for division, I suspect that since multiplication is not commutative, one should look for left and right division operators, say e/x and $x\backslash e$ (which bind the variable x) such that $x \times (x\backslash e) = e$ and $(e/x) \times x = e$ for general (or merely 'interesting') e, while $x\backslash(x \times e) = e$, $(e \times x)/x = e$ for e in which x does not occur free. (These are by analogy with the 'β' law that $x\hat{}\log_x e = e$ for general e, and the 'η' law that $\log_x(x\hat{}e) = e$ for e in which x does not occur free.) But perhaps one also needs remainder operators of some kind? Maybe another tack is to mimic the constructions of the integers and the rationals from the natural numbers, as equivalence classes.

Once one gets speculating, there is no end to it. Is there some way to represent power-series? (For this one would probably have to consider lazy streams of some kind. As indicated later, I think this is a interesting direction for exploration.)

If one completely abandons oneself to the thought of using the λ-calculus as an 'arithmetic', then a host of almost hallucinatory questions arises. What about polynomials? Their roots? Fixed points of polynomials? Rational functions? Formal derivatives? Integration? Trigonometry? Power series? What about change of base in logarithms, or primes and unique factorisation, or Diophantine equations, or having an *ordered* structure, or factorials and Stirling's approximation, or algebraic numbers versus transcendentals, or Fourier transforms? Convolution? Who is to say that all these questions are entirely senseless?

Can these 'numbers' can in some interesting way be connected with the arithmetic of Conway games [4]?

3 The AMEN machine

In this section we sketch in general terms the Arithmetical Logical Unit, or ALU as it is called in computer architecture. Intellectual property considerations dictate that we should restrict ourselves tightly to giving only cryptic hints.

The machine works by graph rewriting. In the graph, the leaf nodes can take on of 5 forms: `(+)`, `0`, `(*)`, `1`, `(^)`. There are 3 forms of inner node. All inner nodes are binary (for the moment), and there is one for each of the binary functions `+,*,^`. The three different kinds of node help conserve space.

The general idea is much as with a combinator machine such as the 'SKI'-machine (Turner [29], Stoye [26]), or a supercombinator machine (Hughes [11], Fairburn ?? Johnsonn-Augustsonn ??, Peyton-Jones [20]). One 'unwinds' the graph in pursuit of the combinator at its head, building up a stack of intermediate nodes. When the combinator is found, if there are sufficiently many arguments stacked, the graph is rewritten so that the node that formerly held an instance of the left-hand-side of its definition now holds the corresponding instance of the right-hand-side.

However, here there are three levels of arithmetic, and so we have an additive stack, a multiplicative stack, and an exponential stack. It is the ordinary thing, but in 3 dimensions. One can think of the machine as being in 3 modes: exponential (the E-axis), multiplicative (the M-axis), and additive (the A-axis).

(I haven't really worked this out, as must be obvious.)

Perhaps one could express this mechanism in the form of an interpreter. In Haskell there might be a state monad, with 3 stacks. Garbage collection could be represented by an operation to 'canonicalise' the graph to the part of potential interest (accessible from a stack), as in breadth-first copying. However, garbage collection should do affine reductions.

An observation of Conor McBride: we have a machine which spends practically all of its time performing garbage collection, except once in a while when it gets round to performing an addition. Truely the blame for all forms of garbage may be laid at the feet of addition.

We should have 'varargs' versions of the binary addition and multiplication nodes, as with finite sums and products. Zero and one would be the degenerate, 0-place versions. If a right field in a product is a sum, then expanding will introduce sharing of the left field. This is a typical optimisation.

Wadler [31], how to fix space leaks with a garbage collector. He remarks that certain functions (paradigmatically projections) can safely be evaluated by a garbage collector, as they need no new storage. He probably knew that this holds good for all affine functions.

Purely exponential stack: figure 2

The usual combinators: fig 3.

Exponential:

$\ldots$	$\hat{}$	$\ldots$	$\hat{}$	$\ldots$	$\hat{}$	$\ldots$	$\hat{}$	$(b\,\hat{}\,a)$	$\longrightarrow$	$\ldots$	$\hat{}$	b	$\hat{}$	a
c	$\hat{}$	b	$\hat{}$	a	$\hat{}$	f	$\hat{}$	$(+)$	$\longrightarrow$	$(c\,\hat{}\,b\,\hat{}\,f)$	$\hat{}$	b	$\hat{}$	a
$\ldots$	$\hat{}$	b	$\hat{}$	a	$\hat{}$	f	$\hat{}$	$(\times)$	$\longrightarrow$	$\ldots$	$\hat{}$	$(b\,\hat{}\,f)$	$\hat{}$	a
$\ldots$	$\hat{}$	$\ldots$	$\hat{}$	a	$\hat{}$	f	$\hat{}$	$(\hat{})$	$\longrightarrow$	$\ldots$	$\hat{}$	f	$\hat{}$	a
$\ldots$	$\hat{}$	$\ldots$	$\hat{}$	a	$\hat{}$	f	$\hat{}$	0	$\longrightarrow$	$\ldots$	$\hat{}$	$\ldots$	$\hat{}$	a

Applicative (other way round):

$(a\,b)$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\longrightarrow$	a	b	$\ldots$
$(+)$	f	a	b	c	$\longrightarrow$	a	b	$(f\,b\,c)$
$(\times)$	f	a	b	$\ldots$	$\longrightarrow$	a	$(f\,b)$	$\ldots$
$(\hat{})$	f	a	$\ldots$	$\ldots$	$\longrightarrow$	a	f	$\ldots$
0	f	a	$\ldots$	$\ldots$	$\longrightarrow$	a	$\ldots$	$\ldots$

Figure 2: Stack transitions for the (exponential part of the) arithmetical machine.

$(f\,a)$	$\ldots$	$\ldots$	$\ldots$	$\longrightarrow$	f	a	$\ldots$	UNWIND
B	f	a	b	$\longrightarrow$	f	$(a\,b)$	$\ldots$	$(.)$
C	f	a	b	$\longrightarrow$	f	b	a	SWAP
W	f	a	$\ldots$	$\longrightarrow$	f	a	a	DUPL
K	f	a	$\ldots$	$\longrightarrow$	f	$\ldots$	$\ldots$	POP
I	f	$\ldots$	$\ldots$	$\longrightarrow$	f	$\ldots$	$\ldots$	NO-OP

Figure 3: Usual (BWICK) combinators.

```
a + b = \ l s z -> b l s (a l s z)
a * b = \ l s z -> b l (a l s) z
a ^ b = \ l s z -> b l' (a l) s z
                   where l' xi x = l (\ n -> xi n x)
0 = \ l s z -> z
1 = \ l s z -> s z
```

Figure 4: Arithmetic with a limit operation

4 Types (rough notes)

The exponential function entails a certain shift in type. You can see this if you think of numbers as iterating operations – for example, handing another coin to a shopkeeper. Doing such a thing $n+m$ times is just doing it n times, and then m more. Doing a thing $n \times m$ times is doing m blocks in each of which it it is done n times. Doing a thing n^m times also involves doing something m times, but now the operation is (even) higher order. It is the operation on a concrete operation which iterates it n times. (To be sure, there is also a 'gearing-up' involved in the step from addition to multiplication, since we have to think of the operation as something that can be replaced by blocks or multiples of itself.)

You can see this shift in another way, if you consider what happens when the notion of number is extended (as in figure 4) to include 'limit' numbers of some kind, such as limits of countable sequences. We can extend the arithmetical combinators to commute with taking limits in the appropriate sense. (The domain of the type of sequences plays no rôle.)

The things to remark here:

- See how the brackets creep to the left.
- Look at how `b` is taken to be a 4 place function in the equation for exponentiation.
- We need some notion of limit structure.

Affine terms are always well-typed. (This is a nice fact, since it is merely something to be observed, rather than a technical result.) It is some restriction on W, or +, or S (or other 'repetitor') that enforces a typing discipline. It is to be suspected that in order to get a class of functions which is closed under identification of different arguments (i.e. under the W combinator), some kind of polymorphism is involved.

5 Streams, dreams (jumbled remarks)

For a very long time I have had the following 'dream': there is a very simple arithmetical interpretation (in some sense) of a system such as Gödel's theory T of primitive recursive functionals of finite type. As a corollary, which will be scarcely more than a remark, it will follow that T's provable ordinals are below ϵ_0. The dream settled in me when listening around 1971 to a seminar by Jane Bridge about systems of ordinal notations developed by W. Neumer [13] [14] [15] [16] [17], [18] [19]. (Neumer seems to have disappeared without trace. Perhaps he was murdered by a proof theorist?) Neumer's system involved functionals of higher type over the ordinals; the ones that concerned him he called 'facients'. Jane also covered some similar systems developed by Peter Aczel (who was infl;uenced by Neumer's papers, and may have been stimulated by ideas of S. Feferman, reported in a paper [7] called 'Hereditarily replete functionals over the ordinals'). Aczel's systems are described in [1]. The ordinals obtained were not big, but I have always liked Kreisel's 'jingle' (stolen from the advertising campaign of a cigarette company) that with proof-theoretic ordinals, "it's not how big you make them, but how you make them big". The dream received some kind of corroboration from a paper [8] by W. Howard called 'Assignment of ordinals to terms for primitive recursive functionals of finite type', a paper which to this day remains highly mysterious, (despite technical clarifications and improvements by K. Schütte [24] and most recently A. Weiermann [32]). Howard's paper has (obscurely) the 'feel' of being based on a system of functionals.

Some initial investigations showed that the idea of such an 'interpretation' could be carried through for primitive recursive arithmetic with induction restricted to ground types, the corresponding system over the constructive second number class), and indeed some fragments of Gödel's T. What I wanted was to find for the subject of proof-theoretic ordinal analysis something comparable to the method of computability arguments (nowadays called 'logical relations') at the time developed by Tait, Howard, Girard, Martin-Löf, Prawitz, et al.

It is well known that the finite type structure over a ground type O with pairing can be encoded in a 'spine' of pure types: either $O_{n+1} = O_n \to O_n$ or $O_{n+1} = O_n \to O$ will do, and no doubt other 'spine's. I mucked about with this for a while, and tried several notions of hereditary majorisation, trying to show that the terms on Gödel's T were majorised in this sense by Aczel's iteration functionals. It was like stirring up mud in a stream. I gave up, for 20 years.

This 'dream' began to assail/haunt me again when I noticed the completeness of the arithmetical combinators. I became aware of papers by Schwichtenberg, Statman, and Zaionc that were (more or less distantly) related to this phenomenon. (These gents showed *inter alia* that the numerical functions representable *à la* Church in the simply typed λ-calculus are the 'extended polynomials', these being the least closure under composition of the constant and projection functions, the function 0^n, addition and multiplication. Zaoinc obtained further results for other finitary data types besides the natural numbers.) One would like to somehow drastically extend this definability result to stronger λ-calculi, both in the dimension of inductively defined types and their recursors, and in the dimension of transfinite extension of the type structure, as with universes. Ralph Loader, Daniel Leivant, Norman Danner, and Gordon Plotkin too are somewhere on the horizon of this subject. Something I have always felt is very significant is the use of the function $\overline{sg}(n) = 0^n$, which introduces some anti-monotonicity (as well as allowing for definition by cases, truth functions, and other fine infra-structure). (Note that this is a particular case of the exponentiation function whose representation does *not* require a type shift.)

It is unreasonable to expect to go to one's grave having rid oneself of all one's dreams. But human beings are not very reasonable, and I am less reasonable than most.

Of course, everyone knows that the ordinal of Gödel's T is ϵ_0 (except Girard, who with charming eccentricity claims it is the Howard ordinal, ...). The modern technology to prove it was in essence invented by Schütte, and involves the use of infinitary terms (that can be represented in a finitary way). Schütte's technique was developed (in different ways) or re-invented by Tait [27], Hinata, Sanchis [23] and Diller [6]. I have never felt entirely happy with this[9], although I have to admit that probably the use of infinite terms in some guise is essential. But what form?

Let us think of an 'infinitary extension' of the calculus of simply type λ-calculus (or some combinatory isotope). I shall call it $\mathcal{N}$, after Napier (the father of logarithms, from Edinburgh). We introduce a new unary type-constructor $[\cdot]$. If A is a type, then $[A]$ will be a type of (some) infinitely proceeding sequences/vectors/streams of objects of type A. The streams will be built up by a few forms of *corecursion*.

Here (in dribs and drabs) are some rules (of the kind) I have been looking at.

5.1 Construction and deconstruction

We add constants for pushing something on the front of a stream, and operators $\cdot_0$ and $.'$ for taking the head and tail.

$$\frac{a : A \qquad as : [A]}{(a, as) : [A]}$$

$$\frac{a : [A]}{a_0 : A} \qquad\qquad \frac{as : [A]}{as' : [A]}$$

$$\begin{array}{rcl} (a, as)_0 & = & a \\ (a, as)' & = & as \end{array}$$

Nothing particularly funny there.

[9] I should however mention a recent paper by Voda [30], which expresses some thoughts I find very congenial – though I haven't penetrated his own approach.

5.2 Zippy things

I refer to 'zip' in functional programming. By 'zippiness' I mean to include 'bang', and lifting functions to sequences. There are various choices here, which are perhaps to be made on grounds of tidiness or brevity. This is not yet an issue.

$$\frac{a : A}{a! : [A]}$$

$$\frac{as : [A] \qquad fs : [A \to B]}{as[\hat{\ }]fs : [B]}$$

$$\frac{as : [A \to B] \qquad bs : [B \to C]}{as[\times]bs : [A \to C]}$$

$$\frac{as : [A \to B \to C] \qquad bs : [A \to C \to D]}{as[+]bs : [A \to B \to D]}$$

The last three rules above merely extend the binary arithmetical combinators to sequences, in a 'coordinate-wise' fashion. One may perhaps replace the whole block instead with the combinators known to functional programmers as 'map', 'zipWith' and 'repeat'. (I learnt from Daniel Fridlander and Mia Indrika the observation that these combinators are really manifestations of the same phenomenon. Indeed, one may make do with just ! and [ˆ].) Some suitable rules may be:

$$\frac{a : A}{\text{repeat}(a) : [A]}$$

This is just $a!$.

$$\frac{f : A \to B \qquad as : [A]}{\text{map}(f, as) : [B]}$$

This is just $as[\hat{\ }]f!$.

$$\frac{f : A \to B \to C \qquad as : [A] \qquad bs : [B]}{\text{zipWith}(f, as, bs) : [C]}$$

This is just $bs[\hat{\ }]as[\hat{\ }]f!$. And so on.

Equations

$$\begin{array}{lcl@{\qquad}lcl}
(a!)_0 & = & a & (a!)' & = & a! \\
(as[\hat{}]fs)_0 & = & as_0\hat{}fs_0 & (as[\hat{}]fs)' & = & as'[\hat{}]fs' \\
(as[\times]bs)_0 & = & as_0 \times bs_0 & (as[\times]bs)' & = & as'[\times]bs' \\
(as[+]bs)_0 & = & as_0 + bs_0 & (as[+]bs)' & = & as'[+]bs'
\end{array}$$

Nothing particularly funny there, either.

5.3 Cumulative product

$$\frac{fs : [A \to A]}{\Pi fs : [A \to A]}$$

Equations.

$$\begin{array}{lcl}
(\Pi fs)_0 & = & 1 \\
(\Pi fs)' & = & (fs_0!)[\times](\Pi fs')
\end{array}$$

The equation for Πf may be a little easier to understand if one contemplates:

$$\Pi(f_0, f_1, f_2, \ldots) = (1, f_0, f_0 \times f_1, f_0 \times f_1 \times f_2, \ldots)$$

and bears in mind that $f \times g$ equals $g \cdot f$. Notice that if one defines

$$\begin{array}{lcl}
f(0) & = & a \\
f(n+1) & = & b(n, f(n))
\end{array}$$

then

$$\begin{array}{lcl}
f(0) & = & a \\
f(1) & = & b(0, a) \\
f(2) & = & b(1, b(0, a)) \\
\ldots & & \\
f(n) & = & b(n-1, b(n-2, \ldots, b(0, a)) \ldots)) \\
 & = & (b(n-1) \cdot b(n-2) \cdot \ldots \cdot b(0))(a) \\
 & = & a\hat{}(b(0) \times \cdots \times b(n-1))
\end{array}$$

If we order the arguments of the recursion combinator (in Gödel's T) as follows:

$$R : (N \to A \to A) \to N \to A \to A$$

with equations

$$\begin{array}{lcl}
R(b, 0, a) & = & a \\
R(b, S(n), a) & = & b(n, R(b, n, a))
\end{array}$$

then we can rewrite the equations

$$\begin{array}{lcl}
R(b, 0) & = & 1 \\
R(b, S(n)) & = & b(n) \cdot R(b, n)
\end{array}$$

but also as

$$\begin{array}{lcl}
R(b, 0) & = & 1 \\
R(b, S(n)) & = & R(b \cdot S, n) \cdot b(0) \\
 & = & b(0) \times R(S \times b, n)
\end{array}$$

5.4 Cumulative sum

It is natural to propose also (or at least hold in reserve) a cumulative sum operator.

$$\frac{fs : [(A \to A) \to A \to A]}{\Sigma fs : [(A \to A) \to A \to A]}$$

Equations.

$$\begin{array}{rcl} (\Sigma fs)_0 & = & 0 \\ (\Sigma fs)' & = & (fs_0!)[+](\Sigma fs') \end{array}$$

Surely if anything deserves to be called ω it is $\Sigma(1!)$.

5.5 Now what?

What about the 'interpretation'? The one criterion I have is that it should be almost obvious. It should be the sort of thing one calls an 'observation' rather than a 'result'. The difficulty will be to think of it, rather than verify it. And it should follow, as a simple corollary, that the proof theoretic ordinal of Gödel's T is $\leq \epsilon_0$.

So, let us take Gödel's T in the following form.

Type structure:

$$N \qquad\qquad (A \to B) \qquad\qquad O$$

Constants:

$$\begin{array}{lcl} (\hat{\ }) & : & A \to (A \to B) \to B \\ (\times) & : & (A \to B) \to (B \to C) \to A \to C \\ (+) & : & (A \to B \to C) \to (A \to C \to D) \to A \to B \to D \\ 0 & : & A \to B \to B \\ R & : & (N \to A \to A) \to N \to A \to A \\ Z & : & N \\ S & : & N \to N \\ ZO & : & O \\ ZS & : & O \to O \\ ZL & : & (N \to O) \to O \end{array}$$

Rules (of 'cut'):

$$\frac{a : A \qquad\qquad b : A \to B}{a\hat{\ }b : B}$$

$$\frac{b : A \to B \qquad\qquad c : B \to C}{b \times c : A \to B}$$

$$\frac{c : A \to B \to C \qquad\qquad d : A \to C \to D}{c + d : A \to B \to D}$$

(It is not essential to have binary operators $\times$ and $+$.) Let $1 = 0\hat{}0$.

Equations (rewriting rules):

$$
\begin{array}{lcl}
b\hat{}a\hat{}(\hat{}) & = & a\hat{}b \\
b\hat{}a\hat{}(\times) & = & a \times b \\
b\hat{}a\hat{}(+) & = & a + b \\
b\hat{}a\hat{}0 & = & b \\
c\hat{}(a \times b) & = & (c\hat{}a)\hat{}b \\
c\hat{}(a + b) & = & (c\hat{}a) \times (c\hat{}b) \\
c \times (a + b) & = & (c \times a) + (c \times b) \\
c \times 0 & = & 0 \\
a + b + c & = & (a + b) + c \\
a + 0 & = & a \\
0 + a & = & a \\
a \times b \times c & = & (a \times b) \times c \\
a \times 1 & = & a \\
1 \times a & = & a \\
Z\hat{}b\hat{}R & = & 1 \\
(n\hat{}S)\hat{}b\hat{}R & = & (Z\hat{}b) \times (n\hat{}(S \times b)\hat{}R)
\end{array}
$$

Define the standard interpretation to be the model in which $\rightarrow$ is the full function space constructor, N is the natural numbers, O is the countable ordinals, ZO is the ordinal zero, SO is the ordinal successor function, LO is the operation $(a_0, a_1, \ldots) \mapsto \sup_n a_n$. Define the ordinal of T to be $\sup_n |t_n|$ where t_n is an enumeration of t with $t : O$, and $|a|$ is the value of a in the standard interpretation.

Now what? We might try to map a type A of T to one A^∞ of $\mathcal{N}$ by something like:

$$
(A \rightarrow B)^\infty = \begin{cases} [B^\infty] & \text{if } A = N \\ A^\infty \rightarrow B^\infty & \text{otherwise} \end{cases}
$$

We might attempt to map a term $a : A$ of T somehow 'homomorphically' to one $a^\infty : A^\infty$ of $\mathcal{N}$. But it is not at all obvious what that should mean.

We might take some inspiration from realisability. Or a logical relations argument. Or a majorisation argument [9]. But probably none of these things.

6 detritus

I suspect that proof-theorists (with some honourable exceptions: Peter Aczel, Feferman) have been unduly obsessed with well-foundedness, or in other jargon, initiality. If, in 1998, in a computer-science department, one reads the original papers by Tait [27], Martin-Löf [12], Howard [10], .. about infinite terms, or indeed (from a Curry-Howard perspective) infinite proofs, it seems reasonable to ask “should this really be expressed in terms of streams?”.

What is the ‘strength’, in any reasonable sense, of (some simple) principles of co-induction (say, as on streams, which seems somehow dual to induction on the natural numbers)? I have *no* idea! On the other hand, from programming in lazy functional languages, I know that coinduction is an enormously useful form of recursion. (It seems to arise whenever you want to deal with something ‘stateful’.)

Addition, multiplication (not necessary), exponentiation, and zero are 4 of the 5 ingredients of Cantor normal form. What could be ω?

$$\begin{array}{lcl} a \mathbin{\hat{}} (b : bs) \mathbin{\hat{}} \omega &=& a : (a \mathbin{\hat{}} b) \mathbin{\hat{}} bs \mathbin{\hat{}} \omega \\ \omega \times hd &=& 0 \\ \omega \times tl &=& hd + tl \times \omega \end{array}$$

Maybe. (: is an infix cons.) Something else that seems to crop up is *bang* – a kind of infinite repetition.

$$\begin{array}{lcl} a \mathbin{\hat{}} bang &=& a : a \mathbin{\hat{}} bang \\ bang \times hd &=& 1 \\ bang \times tl &=& bang \end{array}$$

And then, there are pointwise operations (map, zipWith, ...).

How to make something precise of this?

6.1 Hereditary orderings

Very vague ideas.

$$f <_{\overline{A}\to\iota} g = \exists y : \overline{A}.\forall x : \overline{A}.y <_{\overline{A}} x \to x\hat{}f <_\iota x\hat{}g$$

$(y_1, .., y_k) <_{\overline{A}} (x_1, .., x_k)$ means $\leq$ in each coordinate, but strict in at least one coordinate?

Define monotone, $\leq$ first. Then define strict, $<$.

References

[1] P. Aczel. Describing ordinals using functionals of transfinite type. *Journal of Symbolic Logic*, 37:35–47, 1972.

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

[3] A. Church. *The Calculi of Lambda Conversion.* Princeton Univ. Press, 1941.

[4] J.H. Conway. *On Numbers and Games.* London Mathematical Society monographs,6. Academic Press, London, 1976.

[5] A. S. Davis. Half-ring morphologies. In M.H. Löb, editor, *Proceedings of the Summer School in Logic, Leeds, 1967*, volume 70 of *Lecture Notes in Mathematics*, pages 253–268, Berlin, Heidelberg, New York, 1968. Springer-Verlag.

[6] J. Diller. Zur berechenbarkeit primitiv-recursiver functionale endlicher typen. In Schütte K. Thiele H Schmidt, H.A., editor, *Contributions to Mathematical Logic*, pages 109–120. North-Holland, Amsterdam, 1970.

[7] S. Feferman. Hereditarily replete functionals over the ordinals. In Myhill J. Vesley R.E. Kino, A., editor, *Intuitionism and Proof Theory: Bufallo N.Y. 1968*, Amsterdam, 1970. North-Holland.

[8] W.A. Howard. Assignment of ordinals to terms for primitive recursive functionals of finite type. In Myhill J. Vesley R.E. Kino, A., editor, *Intuitionism and Proof Theory: Bufallo N.Y. 1968*, pages 443–458, Amsterdam, 1970. North-Holland.

[9] W.A. Howard. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *Lecture Notes in Mathematics*, chapter Hereditarily majorizable functionals of finite type, pages 454–461. Springer-Verlag, Berlin, Heidelberg, New-York, 1973.

[10] W.A. Howard. Ordinal analysis of terms of finite type. *Journal of Symbolic Logic*, 43(3):355–374, 1980.

[11] R.J.M. Hughes. *The design and implementation of programming languages. PRG-40.* PhD thesis, Programming Research Group, Oxford University Computing Laboratory, September 1984.

[12] P. Martin-Löf. Infinite terms and a system of natural deduction. *Compositio Mathematica*, 24:93–103, 1972.

[13] W. Neumer. Zur konstruktion von ordnungszahlen, i. *Mathematishe Zeitschrift*, 58:319–413, 1953.

[14] W. Neumer. Zur konstruktion von ordnungszahlen, ii. *Mathematishe Zeitschrift*, 59:434–454, 1954.

[15] W. Neumer. Zur konstruktion von ordnungszahlen, iii. *Mathematishe Zeitschrift*, 60:1–16, 1954.

[16] W. Neumer. Zur konstruktion von ordnungszahlen, iv. *Mathematishe Zeitschrift*, 61:47–69, 1954.

[17] W. Neumer. Zur konstruktion von ordnungszahlen, v. *Mathematishe Zeitschrift*, 64:435–456, 1956.

[18] W. Neumer. Algorithmen für ordnungszahlen und normalfunktionen, part i. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 3:108–150, 1957.

[19] W. Neumer. Algorithmen für ordnungszahlen und normalfunktionen, part ii. *Zeitschrift für mathematische Logik und Grundlagen der Mathematik*, 6:1–65, 1960.

[20] S.L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International, 1987.

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

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

[23] L.E. Sanchis. Functionals defined by recursion. *Notre Dame Journal of Formal Logic*, 8:161–174, 1967.

[24] K. Schütte. *Proof Theory*. Springer, Berlin, Heidelberg, New York, 1977.

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

[26] Stoye. The skim microprogrammer's guide. Technical Report 31, University of Cambridge Computer Lab, October 1983.

[27] W.W. Tait. Infinitely long terms of transfinite type. In Dummett M.A.E. Crossley, J.N., editor, *Formal Systems and Recursive Functions*, Studies in Logic and the Foundations of Mathematics, pages 176–185, Amsterdam, 1965. North-Holland.

[28] Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*, volume 43 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1996.

[29] D.A. Turner. A new implementation technique for applicative languages. *Software - Practice and Experience*, 9:31–49, 1979.

[30] Paul J. Voda. A simple ordinal recursive normalisation of gödel's t. In *csl????*, 199?

[31] Philip Wadler. Fixing a space leak with a garbage collector. *Software Practice and Experience*, 17(9):595–608, September 1987.

[32] A. Weiermann. A proof of strong uniform termination for gödel's T by methods from local predicativity. *Archives for Mathematical Logic*, 1997.

[33] L. Wittgenstein. *Tractatus Logico-Philosophicus.* Routledge and Kegan Paul, London, 1961. First German edition in Annalen der Naturphilosophie, 1921.