Apologia

Peter Hancock

August 12, 1998

Contents

1 Introduction

This is a draft of the first chapter. Many parts of it are very crudely expressed, or mere placeholders. The organisation is between bad and awful.

This chapter poses the question of what interest there may be from a computer science perspective on the topic in proof theory known variously as 'provable ordinals', or 'ordinal analysis'.

The subject of proof theory and the topic of ordinal analysis within it arose in connection with Hilbert's programme for securing the foundations of mathematics. I begin with a brief survey of its origins. Compared with 100 years ago, the foundations of mathematics are in a more settled state, due primarily to the formulation of satisfactory systems of set theory and hardly at all due to developments in Hilbert's programme. Problems in the foundations of mathematics are less urgent than problems in the foundations of computation. Ideas from proof theory (cut elimination, normalisation, *etc.*) have proved of great relevance in more than one part of computer science. It is natural to try to assess what relevance ordinal analysis has to the study of computation.

In 'Computer Programming and Constructive Mathematics' (need to re–read it) Martin–Löf argues that if we accept the original explanations by Kolmogorov and Heyting of the intuitionistic logical operators, then there is a great deal in common between proving theorems in constructive mathematics and writing functional programs to solve a problem expressed as a type in a certain kind of type system. Such a type system ensures that evaluation of a well–typed program by an ideal computer will eventually terminate. When I first read Martin–Löf's paper, I was a programmer, currently writing functional programs, and had had heavy exposure to Martin–Löf's ideas. My estimation his account of programming was that as a guide for making sense of my experience as a programmer, even a functional programmer, it was highly inadequate. I now think that one inadequacy was shared with all contemporary accounts of functional programming. What was lacking was a clear view of what evaluation of mathematical expressions has got to do with running a real program. This is now much clearer, thanks to Spivey, Moggi, Wadler, and others. In the second section I survey these ideas, and suggest how they might fit with Martin–Löf's account of programming, and dependent types.

Type systems of the kind advocated by Martin–Löf are necessarily incomplete. Any type system designed to exclude programs for which evaluation need not terminate, that takes form of a decision procedure will necessarily exclude some programs that do terminate. By Gödel's incompleteness theorems, for any minimally expressive consistent type theory there are recursive functions which cannot be proved total within it. Merely by cardinality, we know that the capacity of such a system to denote values in an uncountable domain (*e.g.* the set of unary functions over *Nat*, or the set of countable ordinals, or the power set of *Nat*) is necessarily limited. The third section of this chapter argues that among these incompleteness phenomena we may count the limits imposed by a type–system upon the discriminative power of the interactive programs that can be written within it. Ordinal analysis sheds light on how these limits depend on the type system.

Predicativity?

2 Why is what interesting?

In this section I present a synopsis of Hilbert's program, and the results of Gentzen and Ackermann. Extensive surveys and assessments of of Hilbert's program can be found in:

Kreisel, G. (1958) Hilbert's programme. Dialectica, 12, 346–372. Revised version in [benacerraf–putnam].

Benacerraf, P. and Putnam, H. (1983) Philosophy of mathematics: selected readings (2nd ed.). Cambridge University Press.

Kreisel Survey of Proof Theory J.S.L. 33 1968 pp. 321–388.

Wilfried Sieg. 'Hilbert's Programs: 1917–1922'. Published ?? Carefully traces the actual historical development Hilbert's finitist program, from origins in the foundations of mathematics and detailed investigations by Hilbert and Bernays. Analyze the first steps in the new subject of proof theory.

Bernays 1939 ??. Hilbert–Bernays, Grundlagen der Mathematik, Vol II, 5 and 3c.

Gentzen 1943 ??. Provability and nonprovability of restricted transfinite induction in elementary number theory.

Controversial problems raised by the referees of Gentzen's first paper with reduction sequences. Gentzen's proof was suspected to contain implicit use of an intuitionistic principle, namely the fan theorem. There had been fierce debates for decades between leading mathematicians such as Hilbert, Brouwer, Weyl, Lebesgue about the status of such principles. While the controversy wore on, Gentzen took the opportunity to write another proof in which the only finitistically problematic form of reasoning employed was induction on a linear wellordering of the same order type as Cantor's ϵ_0. Check it was Cantor. Weyl (?) remarked that Gentzen was the madman who had justified induction up to ω by using induction up to ϵ_0.

Ackermann ??.

From one point of view, these are very inauspicious beginnings.

Oh well, so much for formalism. Any sufficiently powerful formal system that is consistent will inevitably be incomplete. And so much for finitism, because the principles required to prove the consistency of a system such as first order arithmetic go beyond what can be derived in first order arithmetic. It was still interesting to ask if finitistic proof or provability could somehow be characterised: is primitive recursive arithmetic finististically sound? What about diagonalising out of it? What about iterating the process of diagonalisation, and diagonalising over the iterations? Do the finitistically provable arithmetic equations perhaps coincide with those that can be proved in first order arithmetic? Is there a ordinal which serves as a horizon for what can be seen to be well founded by finitistically acceptable reasoning?

Does the surveyability stressed by Hilbert have to do with the feasibility of carrying out a computation, or with the feasibility of writing down an algorithm which in principle could carry out that computation?

Schutte. Feferman. Reductive proof theory. Formalisation of the idea 'predicative relative to the natural numbers'. Measurement of strength.

Metamathematics. Maybe characterise Schütte school as metamathematics. Part of mathematical logic.

But over concern with consistency. Consistency is really not enough. We might have non–terminating computations even though we don't have any term in the empty type???

2.1 Foundational

During the 19–th century, there was great progress in mathematics. To give only some of the most conspicuous examples, there had been progress in the study of the continuum, power–series, calculus, geometry and probability. In particular, Cantor's efforts to understand the convergence of Fourier series had led to the development of transfinite arithmetic, and the formulation of the first of several paradoxes. Despite efforts to place these mathematical developments on a basis of absolute rigour, begun perhaps by Frege, had failed (albeit very fruitfully). The foundation of even the arithmetic of finite cardinals had proved problematic. It is probably not an exaggeration to say that at the beginning of the century the situation in the foundations of mathematics was a matter of dire emergency.

The topic of ordinal analysis arose during the 1930's in connection with Hilbert's program.

Hilbert's program (refer On the infinite, Sieg, Kriesel) was an approach to the foundation of mathematics, specifically the 'infinitistic' mathematics that had emerged during the 19th century, with studies into the continuum, powerseries, integration, the natural numbers, and cardinals and ordinals in general.

The term 'Hilbert's program' is a historical abstraction, representing a bundle of ideas developing around Hilbert. The fundamental idea was to interpret apparently infinitistic mathematics in terms of so–called finitistic (also: combinatorial, syntactic, formalistic, surveyable, visualisable) statements about concrete objects. A paradigm of a finitistic statement is a universally quantified equation between arithmetic expressions, such as the following Π^0_1 formula in the language of first order arithmetic.

$$\forall x \forall y (x + y = y + x)$$

Finitistic statements were held to have an objective computational meaning. Any particular instance of a free variable equation can be verified or falsified by a effective computation, or a verification procedure. What constitutes evidence for such an equation is a surveyable, perspicuous transformation of the computations of the two expressions to a common form. (Kreisel–Krivine Appendix.)

Hilbert proposed to sidestep the problem of giving a meaning to infinitistic constructions.

If a proof of a finitistic statement using infinitistic statements and principles can be systematically transformed into computational evidence for that finitistic statement, then the infinitistic notions can be explained and exploited as reliable 'facon–de–parler'.

He compared this with translating away mention of 'ideal' entities such as points at infinity in geometry.

Hilbert did not apparently ever give a very precise formulation of the concept of a finitist statement, probably because he did not anticipate that a precise formulation would be needed, but expected that the program would succeed on the basis of evident, uncontroversial properties of the informal notion.

To be distinguished from formalism.

By the end of the 1930's (when ordinal analysis began) the urgency of foundations of mathematics was declining. Mathematics could be built on set–theory. Three quarters of a century later, the crisis has become closer to a coma.

It seems worth saying something about Hilbert's programme, because Hilbert's notion of a finitist statement was closey connected with the idea of computation. A finitist statement is a general statement about concrete objects (which we can take to be natural numbers) of which the truth–value of any particular instance can be established by a "surveyable" computation.

Both Gentzen and Ackermann showed that this could indeed by done (in the case of first order classical arithmetic, in Ackermann's case formulated without quantifiers but with the so called ε−symbol, or Hilbert's choice operator). However, the computational content of a proof using infinitistic notions of a finitist statement is (measurably) difficult to "survey". Both Genzten and Ackermann defined essentially while−loop programs for correctness preserving transformations of formal derivations to a form involving only finitist statements, whose termination was established by a use of a construction on well−orderings which many people in fact find decidedly difficult to visualise, namely the lexicographic order of descending chains (of multi−sets) in a given well−order. (Of course, Gentzen had previously submitted a different proof for publication not involving an ordinal−arithmetic assignment, but its structure was not altogether clear.

The results which mark the start of ordinal analysis were due to Gentzen and to Ackermann. They seem to coincide with a much greater sea−change in logic. This larger change was away from conceptual analysis (of notions of set, rule, proof, effective computation ..) and construction of systems (Frege, Peano, Russell−Whitehead, ..) and towards mathematical logic and metamathematics. It is as if what before was the subject matter (namely mathematical reasoning itself) and was called into question, is now taken for granted as the method for studying properties of formal systems, which are the products of conceptual analysis.

If one uses mathematical methods to debug and inform conceptual analysis, then one is doing applied mathematics, in which the structures studied (like formal systems) are relevant to modelling systems of reasoning of some kind.

2.2 Post foundational

Before powerful computers were commonplace, a 'mathematical system' was necessarily something that belonged to the realm of fiction, myth, or pure mathematics. Now, there are actually running embodiments of such systems.

We now have real systems running on computers, rather than notional systems in which mathematics can be 'in principle' encoded, or metamathematical structures modelling features of those systems.

3 What is its interest?

It has been suggested that there is a deep connection between certain kinds of mathematical reasoning, and programming. It is not merely that a mundane form of mathematical reasoning is involved in writing programs, but rather that mathematical reasoning itself, or at least a core, constructive part of it, is itself a specialised form of programming. (Martin Löf. Date 198? Royal Society lectures? Hannover. First publish??)

The origins of this suggestion are in the interpretations by Heyting [??] and Kolmogoroff [??] of intuitionistic logic in terms of expectations and fulfillments, or problems and solutions. One can give a computational interpretation to 'fulfillment' and 'solution'.

The suggestion still appears highly paradoxical. Mathematicians and programmers with whom I have discussed it have found not only risible, but sometimes even offensive. If mathematicians with their mathematical hats on are thereby programmers, not just when they are programming as a sideline, then what computers do their programs run on? What does it even mean to run a proof?

3.1 The pineal gland

(Warning: this is very crudely expressed ..)

In the early 80's there was no clear story about how one wrote a mundane program in a pure functional language, though many people recognised there was a problem, even if they pretended that it was easily solved.

By a mundane program, I mean for example a screen editor, or a program to control or interact with a piece of physical kit like a guided missile, a biscuit machine, a warehouse. Mundane meaning earthly.

Early discussion of this in work on functional operating systems.

One could think of a functional program as an intensionally given continuous function from streams of characters (or other input tokens) to streams of characters (or other output token). The function is continuous in the sense that to obtain a given amount of output, it would be necessary only to provide some finite initial segment of the input. However, we should think of it as given together with the function from output neighbourhoods to open sets of input streams. There is a big difference between the function which returns an endless stream of 1's for all inputs, and that which consumes the first input.

The problem of how to program IO interactions was solved in ML by incorporating impure features. However the ability to write IO programs is one thing, and an account of IO permitting rigorous reasoning about interactive programs is another. Look in ML semantics book.

Spivey. Pointed out monads could be used to structure exception handling in something or other or structure a denotational semantics for catch and fail.

Moggi. View of denotational semantics as translation into a computational meta–language with monadic features; the meta–language is given a denotational semantics (a model) by translation into a meta–language without computational features. Some kind of category supporting the monad you need to interpret computational features (exceptions, state, backtracking, *etc.*). Moggi, E. 1991 Notions of computation and Monads. Look in Haskell report. Andrew Gordon's thesis.

Wadler. Fostered application of Spivey and Moggi's ideas coming from category–theory to functional programming. Wadler, Peyton–Jones, Mark Jones, *et al.* showed how to use monads to organise parsers, interpreters, typecheckers, *etc.*, as well as mundane programs such as internet servers and rough and ready tools. Taken as the basis of Haskell IO.

Wadler wittily describes the IO–problem for pure functional languages as a form of Descarte's problem of how the soul influences and suffers an earthly body. Descartes didn't really figure it out, but he at least located the seat of the actions and passions of the soul in the pineal gland. In Wadler's comparision, the realm of consciousness and spirit is the realm of programs as mathematical abstractions, while the realm of matter is the physical world of metal and flesh. The seat of the actions and passions of the sould the soul that corresponds to the pineal gland is the run–time system.

The run time system interprets the program. We can think of this interpretation in terms of successive state transitions in a space of states adequately modelling the program's environment. (Adequate for the purposes of writing a program that when it is interpreted by the run time system ensures that all and only the right things happen.)

To use an architectural metaphor, we can think of the run–time system as comprising a fetch unit and an execution unit. The job of the fetch unit is to fetch IO instructions. They are fetched by evaluating an expression called the program expression (of a certain ground type) to weak head normal form. The instructions are executed by case distinction (despatching, decoding) on the constant at the head of this normal form to processing specific to that constant, with the components (operands) accessible via the fetch unit. The components comprise output operands (*e.g.* prompt strings) that may have to be fully evaluated, and resumption functions that tell the fetch unit what happens next. If there are any results from the execution of the instruction, they are fed back as fully evaluated arguments to these resumption functions so as to form a new program expression (set up a new fetch address) for another machine cycle.

I should like to distinguish between two concepts, which I shall call evaluation, which is the activity of the fetch unit, and computation, which is the activity of the execution unit. Evaluation is what a computer does when we don't care what it is doing beyond that it finishes sometime soon. Running is what it does when it is actually doing something for us or to us, *i.e.* when we interact with it. The distinction between evaluation and running is very obvious in real life when we wonder whether a program is hung – traversing some internal loop or silently halted. A hung program is a dead program, and the opposite of a reactive, live program. Perpetual running is a highly desirable attribute of programs, but perpetual evaluation is its very opposite.

A HALT instruction is very important, and it can be implemented by sending the fetch unit into a loop – in that sense perpetual evaluation can be something very desirable. However a HALT instruction is an explicit 'last gasp' or termination signal.

That a program is running means that it interacts with its environment in the execution phase of successive fetch–execute cycles.

In these senses of evaluation and computation, termination of evaluation is surely of great practical interest, because it is the basis for perpetual computation, which is something very desirable. For precisely inverse reasons, non–terminating evaluation is of great practical interest, because it is something we want to avoid, or detect and get rid of. The class of practically interesting sets is closed under complementation.

It cannot be denied (is anybody going to?) that total correctness is of central importance in programming. Though sometimes computer scientists give the impression that for them a terminating process is an uninteresting process.

It therefore needs no excuse to be interested in type systems for functional programs which at least exclude expressions with no weak head normal form.

The price is that such a system is necessarily incomplete: there will be constructive functions that cannot be written as programs in the system, or true Π^0_2 statements that cannot be proved (except by extending the system).

3.2 Application of mathematics

It is one thing to locate the seat of the passions in the pineal gland, but the problem of accounting for what goes on there just become more intense per unit of volumey. The problem is almost a poetic way of asking how mathematics is applied.

This has not been a popular area in the philosophy of mathematics.

A mathematical value doesn't do anything, but things can be done with it. It isn't an agent of any kind. But we can do things using mathematical values, as when buying something we use finite ordinals to iterate the various processes of handing over coins and bank notes, when tossing coins to decide what to do we use booleans, when running simulations to decide what might happen and what to do about it, when programming the control loop of a missile we use a mathematical model.

When we count out change, we interpret the natural numbers as programs. We can ask to what extent we can think of the application of mathematics as the execution of a program.

3.3 Input Output with dependent types

The notion of 'type' is a bundle of notions with a long history. Some strands come from the foundations of mathematics, and 19th century efforts to formalise mathematical reasoning in a completely rigorous way. There have been type theories in some form starting with Frege's analysis of variable binding operators (*e.g.* the universal and existential quantifiers, series notation) in Begriffschrift[188??] as second order concepts, starting a strand of 'type as a grammatical category'. Other strands come from set theory.

Type as grammatical category. As soon as we have 'function and argument' in a language, as in predicate logic, then we have the notion of applying one thing to another which mightn't 'fit', or 'compute'. The very grammar of predicate logic involves distinctions of type, between formulas and terms, axiom schemes, substitutions, derivations and so on. There is a kind of grammatical calculus implicit in a logical language of any complexity (think of recognising an instance of the elimination rule for the existential quantifier). Because there is calculation, there are types. We can 'fit' a predicate P into the argument place of the universal quantifier $\forall$, but we cannot fit another quantifier into that argument place, or an expression for a term.[1] Perhaps the first person to appreciate the importance of types in logic was Frege.

Russell and Whitehead, ramified theory of types. Notion of type as the domain of significance of a propositional function. The ramifications were between predicates of the same grammatical level. They were designed to avoid paradoxes. The ramifications came from a concern to avoid impredicative quantification. The ramifications were short circuited with the axiom of reducibility. Ramsey noticed that the paradoxes were avoided already by

[1] I think that Frege would not have thought that there was one 'polymorphic' universal quantifier that could be used for predicates of different levels. It is just conceivable that he would allow that $\forall(\forall)$ could be a wellformed, falsely assertible sentence – saying that the $n+1-th-$order property of a $n-$th order predicate that it holds universally holds of all such predicates. (No predicate is ever false.) However, he demanded that a variable ranging over higher order function should in principle 'carry its arguments', as in $\eta-$long normal form. The argument places are built into the very basic logical grammar of a predicate.

mumble, and that the system of ramifications and the axiom of reducibility could be stripped out, leaving essentially Church's simple type theory.

One strands is from programming. Definition of Algol. Hoare Notes on Data Structuring 1970. Hoare's idea is that types are defined abstractly by basic operations available for manipulating data of that type such as constructors, selectors. The compiler uses type declarations to calculate storage requirements for the representations of abstract data types, and to protect against operation sequences that might break representation invariants (so the implementations of the operations might hang).

A concrete representation will have storage requirements. Essentially the idea Fortran. Declare types to say how many units of storage to reserve to hold the value. DOUBLE.

Types are essentially connected with computation.

The notion of type with which programmers are concerned is in some sense the most fundamental notion of type. Types are certainly connected with logic, but that is because logic is connected with computation.

The most common answer to the question 'what are types for?', meaning 'why do our programming languages have type systems?' would probably be that they are there to prevent type errors. (Hoare. Cardelli's survey paper. Milner: well typed expressions don'nt go wrong.) This suggests that type systems are supposed to prevent certain bad events, like a character string jamming in an orifice designed only for booleans, breaking the mechanism or its warranty. That is certainly true. However, besides ruling out bad events, type systems also have (or can have) at least as interesting a rôle in requiring good events – specifically termination of the computations evoked by running the program on an ideal computer. Lack of termination would be an error 'of omission'.

Type systems are to do with total correctness, not just partial correctness (absence of incorrectness). (Report of Harper.)

Type systems which are designed so that head reduction terminates (assuming an ideal computer).

Wellfoundedness is interesting because liveness is. They are different sides of the same coin. It is because we want programs that live (react, are productive) forever that we want their internal computations to terminate.

Evaluation might be accomplished by a hidden computation. Why we are interested in terminating (non−perpetual) computations: they can be run as transactions, as part of the evaluation of a program expression.

So the types in your program are there not just to stop your program going wrong ('for the nasty things in life'), but also to make it go right. (At least, that is how we could in principle use them.)

Coquand's paper on infinite objects. Coquand sketches a a functional programming language with a type system designed to capture 'productivity' – reduction to constructor form, as well (what does he say?) as wellfounded construction. Refer to Brouwer's 'proof' of the Bar theorem.

Another approach to infinite objects is in Martin–Löf's nonstandard type theory, where it is connected with the Brouwerian notion of a choice sequence.

There seems to be more than one way to understand infinite/cyclic/non–wellfounded objects. In all of them, the wellfounded objects are bound to play a central role.

A formal system with a decidable 'proof–predicate' (type checking algorithm) is necessarily a recursively–enumerable kind of a thing. So there is no hope of representing in a formal system any of the classic uncountable domains, like numerical functions (corresponding to batch programs), or countably branching wellfounded trees (interactive programs).

3.4 Countable ordinals

The ordinals are in a sense sacred, as they are more or less by definition the best kind of set (the $\in$ relation is transitive and wellfounded). The interest of countable ordinals is more profane. They can be interpreted as interactive program, where the messages that are exchanged are discrete, and so can be encoded in natural numbers. This interpretation is possible because (what I mean by) the countable ordinals are (modulo isomorphism) the elements of the initial algebra for the functor $W' = Nat \times (1 + W^{Nat})$ on the category **Set**. (The existence of the initial algebra follows from Aczel's regular extension axiom [??].)

It captures a lot of important cases to model an interaction as an exchange of messages. It is a reasonable idealisation to model the messages exchanged in a real interaction by (fully evaluated) natural numbers. So countably branching structures are especially interesting. It is of course of practical interest to determine the effect of bounding the size of messages.

One good reason (among others) to be interested in locating the 'ordinal' of a system is that the countable ordinals are representative of an important class of interactive processes. The greater the ordinal, the greater the ability of the process it represents to discriminate between different environments. The 'ordinal' of a system is a quantitative limit on the interactive complexity of the programs written in that system, meaning their 'acuity', or discriminative power. (This badly needs a proper definition!!)

In operational terms, what a process does is the following. It issues a command, which we may assume is encoded in a finite sequence of bits. A part of the command indicates whether or not a response is expected. If no response is expected, the program does nothing further so far as its environment is concerned. If a response is expected, it is used it to 'despatch' to the next command. Such a program could form part of the control loop for a piece of machinery (e.g. an aileron of a plane, or a computer peripheral) sending signals to actuators and reading signals from sensors. A command after which the program does nothing further is a shutdown or halt command.

We can define more precisely the syntax of the programs and an operational semantics.

The syntax is given as the initial algebra in the category of sets for the following functor, with the obvious action on maps.

$$\begin{array}{lcl} P & : & \mathsf{Set} \to \mathsf{Set} \\ P(X) & = & \mathit{Nat} \times (1 + X^{\mathit{Nat}}) \end{array}$$

That is to say, syntactically wellformed programs are elements of the initial algebra for this functor. The initial algebra exists provided the category of sets satisfies Aczel's regular extension axiom [??].

The functor P and an adequate data structure with which to represent programs can be written in a programming language such as Haskell as follows.

```
data P x       = P Nat (Maybe (Nat -> x))
data Nat       = Zero | Succ Nat
data Maybe x   = No   | Yes  x   -- a standard functor
instance Functor P where
  map :: (x -> y) -> P x -> P y
  map f (P i m)
    = P i (g m)
      where g No       = No
            g (Yes rd) = Yes (f . rd)
data Prog = Prog (P Prog)   -- a fixed point type
```

Of course, Haskell's data–type `Prog` is more inclusive than the initial algebra in the category of sets, as it contains not only 'bottom', but also infinite or lazy data structures such as the following program `echo`. The program emits `Zero` then perpetually returns the successor of its input:

```
ex :: Prog
ex = e Zero where e i = Prog (P i (Yes (e . Succ)))
```

The syntax of programs might equally well be given as the initial algebra for the following slightly different functor B. The elements of this initial algebra are sometimes known as 'tree ordinals' [Schwichtenburg??], or 'Brouwer ordinals' [Palmgren, Coquand and Dybjer]. The operators of the algebra comprise a constant, a unary operation, and an infinitary operation with 'arity' Nat.

$$\begin{array}{rcl} B & : & \mathsf{Set} \to \mathsf{Set} \\ B(X) & = & 1 + X + X^{Nat} \end{array}$$

We can express the Brouwer ordinals in Haskell using a data type such as

```
data B x        = Halt | Incr x | Read (Nat -> x)
instance Functor B where
  map :: (x -> y) -> B x -> B y
  map f Halt       = Halt
  map f (Incr p)   = Incr (f p)
  map f (Read rd) = Read (f . rd)
data MuB = MuB (B MuB)    -- a fixed point type
```

There is a simple isomorphism between the two initial algebras.

```
b2p :: Prog -> MuB
b2p (Prog (P n m)) = iter (MuB . Incr) n (MuB (x m))
                     where x No       y= Halt
                           x (Yes rd) = Read (b2p . rd)
```

```
p2b :: MuB -> Prog
p2b (MuB b) = Prog (f b)
              where f Halt      = P Zero No
                    f (Incr p)  = let Prog (P n m) = p2b p
                                  in P (Succ n) m
                    f (Read rd) = P Zero (Yes (p2b . rd))
```

(An algebra for P can be used to define a 'corresponding' algebra for for B, but apparently not in the other direction.) A $B-$ordinal makes its output available in the smallest possible steps, one for each successor, and a $P-$ordinal 'optimises' consecutive successors together into a natural number.

We may define an operational semantics for the Brouwer ordinals, interpreting them as consuming a stream of input, and producing a finite trace or log of the interactions. A possible interpretation is expressed as a Haskell program in figure 3.4.

A program of this kind interacts with its environment by issueing a command, then either halting or reading a response. I call a program which eventually halts a wellfounded program.

Wellfounded programs are practically interesting because they can safely be run serially in a batch stream as isolated transactions.

```
data Str a      = Push a (Str a)
type Input      = Str Nat

type Behaviour a = Input -> (a,Input)  -- a monad

data Trace      = Trace Nat (Maybe (Nat, Trace))

behave :: muB -> Behaviour Trace
behave (muB p) inp = case p of
  Halt     -> (Trace Z No, inp)
  Incr p'  -> let (Trace n mnt, unconsumed) = behave p' inp
              in  (Trace (Succ n) mnt, unconsumed)
  Read f   -> let Push i inp'       = inp
                  (t, unconsumed)   = behave (f i) inp'
              in  (Trace Zero (Yes (i,t)), unconsumed)
```

Figure 1: Interpretation of Brouwer ordinals as terminating interactive programs

Wellfounded programs are theoretically interesting, because ... all programs are for mortal users approximated by wellfounded programs.

The countable ordinals represent wellfounded programs that interact with their environment by exchanging discrete messages, with no bound on size.

Continuity.

An ordinal is a parser.

Infinite objects. Coquand's 'Infinite Objects in Type Theory'. Hallnäs notion of convergent.

4 Incompleteness

Type systems such as those advocated by Martin–Löf for constructive mathematics are necessarily incomplete.

There is a quantitative relationship between the type constructors available in a type system, and the power we have to express interactive behaviour.

It is like an inverse square law in physics. (Except not inverse but direct, and squaring is only the beginning of an arithmetical hierarchy.)

We should try to understand this abstractly.

The discriminative power of an interactive program. The ability to distinguish between externally produced sequences. Interactive programs represented by proofs in Heyting arithmetic that a ordinal notation is accessible cannot distinguish between certain inputs.

Suppose that we have some encoding of a natural ordinal notation system in the natural numbers. It should be decidable whether a notation given by a natural number is zero, a successor, or a limit, and these should be exhaustive. With a successor notation e, there should be associated an immediate predecessor $pd(n)$. With a limit notation e, there should be associated a sequence $fs(e, n)$ enumerating the immediate predecessors of that limit. (This is nasty to explain properly because we are in Heyting arithmetic.)

By a proof in a system *e.g.* Heyting arithmetic that a notation e is accessible I mean a proof of the formula $Acc(e)$ in the following extension of Heyting arithmetic. We add a new predicate constant Acc to the language, and new rules

$$\frac{Zero(e)}{Acc(e)} \qquad\qquad \frac{Successor(e) \qquad Acc(pd(e))}{Acc(e)}$$

$$\frac{Limit(e) \qquad \begin{array}{c} \vdots n \\ Acc(fs(e,n)) \end{array}}{Acc(e)}$$

The schematic variable e stands for an expression. The third rule binds occurrences of the variable n, which should not occur free in any premise of the subproof (the part above the horizontal rule).

We can interpret a (closed) proof of a formula $Acc(e)$ as an interactive program.

First, reduce the proof to head normal form. The head must be an occurrence of one of the rules (one of the three constructor forms of a Brouwer ordinal). If the proof ends with an application of the last rule, then read a natural number, form the corresponding numeral and substitute it for the free variable in the sub−proof.

It contains the structure of a Brouwer ordinal.

4.1 Weakness/Strength

'Real' systems (actually implementable) versus 'thought' systems (to which 'in principle' mathematics can be reduced).

With real systems, feasibility is an urgent issue.

Related to the phenomena studied in Schutte−school proof theory is an issue stressed by Godel, namely shorter proofs. (Read what he says.) If we strengthen a system (*e.g.* by allowing cuts of greater rank), then not only can we prove more things, but the things we could already prove may have (logarithmically) shorter proofs.

In "The Sand–Reckoner" (Psammites) Archimedes shows how to remedy an inadequacy of the Greek numerical notation system, so that he can write down the number of grains of sand that it would take to fill the universe[2] It would have taken a similarly immense amount of parchment to write down the number using only the Ionian system of numerical notation known to Archimedes. Archimedes devised a place–value system of notation with a base of 10^8.

'Scientific notation', *e.g.* 2.1783×10^{3141} is in a sense the invention of Napier. A 'sparse' representation designed to accomodate numbers with a few consecutive significant digits, of varying orders of magnitude.

Clearly we can iterate the process of taking logarithms which underlies 'scientific notation'. The invention of a numerical notation system in which the idea of sparse representation of large numbers is iterated is due to Cantor.

$$
\begin{aligned}
&CNF_b(n) = b^{l_1} \times c_1 + \ldots + b^{l_k} \times c_k \\
&b > c_i > 0 \\
&l_1 > \ldots > l_k \geq 0 \\
&k \geq 0.
\end{aligned}
$$

How to write down 2^{256}, a la Archimedes, Napier and Cantor. We do it in a simply typed functional language, using only λ–abstraction and application, and (inessentially) polymorphic local definitions. (*i.e.* in minimal logic.) (The arithmetic operators have the usual precedence and are right associative.)

$$
\begin{array}{lrcl}
\textbf{let} & a \wedge b & = & b(a) \\
& a \times b & = & \lambda\, c\,.\,(c \wedge a) \wedge b \\
& a + b & = & \lambda\, c\,.\, c \wedge a \times c \wedge b \\
& one & = & \lambda\, c\,.\, c \\
& zero & = & \lambda\, c\,.\, one \\
& two & = & one + one \\
& eight & = & two \wedge (two + one) \\
& it & = & two \wedge two \wedge eight \\
\textbf{in} & \multicolumn{3}{l}{0 \wedge (+1) \wedge it}
\end{array}
$$

The reason we can write this down (without any use of recursion) is that the type system of minimal logic accomodates the operation $X \mapsto X^X$ which takes a set to the set of unary functions from that set to itself. This makes it feasible to write down (and no doubt to 'survey' in Hilbert's sense) a completely infeasible computation.

[2] According to the heliocentric system of Aristarchus of Samos, it would take about 10^{63} (very much less than 2^{256}) grains.

There is an elegant hierarchy of functions from [Knuth??] [Conway??] based on arithmetical exponentiation *base* ^ *exponent*.

$$\begin{array}{lcl} a \wedge\!\wedge\, b & = & \underbrace{a \wedge \ldots \wedge a}_{b\ \text{`}\wedge\text{'s}} \\ a \wedge\!\wedge\!\wedge\, b & = & \underbrace{a \wedge\!\wedge \ldots \wedge\!\wedge\, a}_{b\ \text{`}\wedge\wedge\text{'s}} \\ \ldots & = & \ldots \end{array}$$

It seems intuitively clear that there is no feasible way to write in the same type system a program to count 2 ^^ (2 ^ 256) grains of sand.

Now what has this to do with interactive programs?

We can 'in principle' write in Gödel's T a program with the same interactive complexity (where is the precise definition??) as the countable ordinal ω ^^ (2 ^ 256) (an exponential tower of 'ω's 2^{256} deep). We can do it in practice (on a sheet of A4 paper) if we add a universe. (Where do I do this??) (*cf.* Godel's remarks on shorter proofs in stronger systems.)

This example shows in perhaps the simplest possible form how a type system limits the programs we can write. We show in a later chapter how by making essential use of dependent type structure, we can write a program to count to 2 ^^ (2 ^ 256).

It is interesting to know the 'asymptotic' limits imposed by a type system upon the terminating interactive programs it will allow, using programs of perhaps astronomical size. This seems to be somehow reflected in terms of how programs of feasible size...

If we know how to make a strong enough type–system, than any program that deserves to be written can be 'in principle' written on a sheet of A4.

5 Iteration, exponentiation

The idea in Wittgenstein's Tractatus, and in the lambda–calculus, that a number is an iterative exponent. (Frege refers to someone in the 19th century.) More generally that a thing is what it does, it is its behaviour in a typical context.

Ordinals are for iterating. One strand in the notion of 'number': iterative exponent. Cantor's exponential notation, arithmetic of iterative exponents.

How to exhaust the strength of a system [3]. Schwichtenburg's result that the numerical functions expressible a la Wittgenstein in the simply typed lambda–calculus are the extended polynomials. This proceeds by analysis of

[3] Archimedes was the inventor of 'the method of exhaustion'

the ($\eta-$long) normal forms. (Independently proved by Statman? Extended by Marek Zaoinc to other data types??)

Feferman [Buffalo 1968] Hereditarily replete families of functionals over the ordinals. Question raised by Feferman: 'What is a suitable generalisation of iteration to transfinite types' (p 296). Sketches how it might go. 'Obviously this process can be iterated, but it is not yet clear how to organise this in a coherent way.'

Neumer. Known to Aczel. Rehabilitation by Simmons. Aczel's extension to transfinite types [JSL 1972].