The Mathematical Revolution Inspired by Computing. J H Johnson & M J Loomes (eds)
©1991 The Institute of Mathematics and its Applications. Oxford University Press

Computing and Foundations

J. M. E. Hyland
Department of Pure Mathematics, University of Cambridge
16 Mill Lane, Cambridge, CB2 1SB

Abstract

The types and functions which are used in computing are very different from traditional mathematical sets and functions; and yet in practice they are treated as much the same. Doing so in a systematic way raises foundational issues, as it is not clear why the traditional mathematical objects should be given privileged ontological or epistemological status in any such treatment. This paper sketches the mathematical contexts (based on categories known as toposes) in which types and functions in functional programming languages can be treated on a par with ordinary mathematical sets and functions. It argues that the existence of such a perspective undermines the commonly accepted view of foundations based on the primacy of that notion of set analysed in modern set theory.

1 Introduction

1.1 Mathematical Logic

How can there be any connection between the engineering science of computing and the esoteric world of the foundations of mathematics? The suggestion seems like a conscious paradox. The link is provided by mathematical logic. Mathematical or formal logic is concerned with the precise mathematical properties of syntax both in itself and in relation to appropriate semantics. The historical roots of the discipline lie in philosophical enquiry, in particular in questions relating to the philosophy of mathematics. However as computing also requires a precise formal syntax, mathematical logic is in principle and (as it turns out) in practice one of the theoretical bases for the new engineering science.

Now the informed reader may well consider that mathematical logic provides a rather superficial connection between computing and foundations. There is after all nothing unusual in areas of pure mathematics later becoming applicable to a new science. Certainly, if there were nothing more to the story, then we would not be justified in talking of the 'mathematical revolution inspired by computing' in relation to the foundations of mathematics. In this paper I try to describe some of the ways in which conceptual problems raised by computing are rousing us from our dogmatic slumbers about foundations.

1.2 The Foundations of Mathematics

One can distinguish two aspects to the foundations of mathematics: questions of ontology (of what there is or can be) and matters of conceptual organisation (theories of natural kinds). Most mathematicians have lost interest in what they regard as ontological questions. To the extent that they consider the matter, they are generally satisfied with some form of set theory as the foundation for mathematics. However even when the cumulative hierarchy of sets is presented in the best possible light, it must seem philosophically problematic. Its main justification is pragmatic: it does the job. (But what job?) On the other hand mathematicians continue to develop the conceptual foundations of their subject without making any fuss about it. The creation of powerful new notions, which serve both to unify strands of mathematical thought and to solve outstanding problems, is part of the natural activity of the mathematician.

Now there is a considerable tension between (these attitudes to) ontological and conceptual foundations. What seem natural mathematical constructions are 'implemented' in current set theory in an apparently arbitrary fashion. For example the fundamental notion of an ordered pair is standardly 'implemented' by Kuratowski's definition:

$$(x, y) = \{\{x\}, \{x, y\}\}.$$

But there is nothing canonical about this definition. On the other hand there seems to be something behind the idea of conceptual or epistemological priority. Some mathematical notions can naturally be explained in terms of others, but not vice-versa. But what has this to do with ontological priority?

This tension between ontological and conceptual foundations suggests that at bottom the issues involved in them may be two sides of the same coin. The idea that some components of our conceptual equipment are ontologically primary is problematic: but also we feel embarrassed if what is for us conceptually primary is taken in any serious sense to be ontologically secondary.

Now at a trivial level any mathematical discipline must have an effect on the conceptual apparatus of mathematics: it introduces its own basic concepts. But theoretical computer science goes further than that. It can claim an effect on conceptual *foundations* because it challenges the accepted fundamental notions of set and function. This is most clear in the problems associated with the rational design of programming languages, some of which are described below.

1.3 Functions in Computer Science

Many branches of pure and applied mathematics share the feature that some mathematical notion of a function is central; but the notions realised in programming languages are peculiarly problematic. One such notion, which seems to be fundamental, is encapsulated in the pure lambda calculus. This is the calculus which underlies LISP and other modern functional programming languages. In accounting for it, one has to account for quite arbitrary definitions by recursion; and hence, in some sense, for the existence of fixed points for arbitrary functions. This is quite at odds with the classical set theoretic conception of a function. In section 2, I explain a modern view of models of the lambda calculus, and describe the mathematics that is needed to make sense of it.

1.4 Types in Computer Science

The practice of programming generally requires one to distinguish between different *types* of entity: between the data types of Booleans and Lists for example. Similarly in mathematics one distinguishes, for example, real numbers from continuous functions. It is of the essence of type theory to make these distinctions; this results in a more rigidly structured universe than the set-theoretic one. For example, there is no primitive meaning to be attached to the intersection of two types; and one has to provide a function (coercion) to map a natural number to the corresponding real number. However, in this paper I will not stick to any firm distinction between sets and types: the distinction has no effect on the semantics which I discuss in Section 3.

Both in programming and in mathematics, types can seem more important conceptually than sets. However there are problematic issues connected with types in programming languages, in particular with such full blown polymorphic types as occur in the calculus of constructions (this calculus forms the basis for the LEGO proof system described in this volume by Burstall [5]). The most important of these issues are those of the modularity and genericity of programs.

1.5 Modularity and Genericity

Modularity

Large pieces of code must be written by many people collaboratively. Thus one wants to be able to write programs in small pieces, each of which does something identifiable, and then slot the components or modules together to form larger programs; and then one wants to iterate the process. This old philosophical idea, that the meaning of the parts should determine the meaning of the whole, is also the ideal of structured programming. At its simplest it suggests writing programs which define extensional operators (without side effects). The most developed form of this is functional programming, where modularity is most effectively controlled by explicit type systems.

Genericity

The idea is to avoid unnecessary work. One should exploit similarity of structure between routines, by writing very general routines which may be used over and over again in different contexts. A traditional example is that of sorting lists. In principle any of the basic sorting algorithms acts on data consisting of a list of elements (of given type) together with a (decidable) total order (on the type). So they act generically on ordered types. Systematising this idea is the task of polymorphic type systems (a sketch of one such system is given in this volume by Burstall [5]).

The ideas of modularity and genericity are clearly related: in practical terms each makes the other more useful. However there is also a tension: crude modularity suggests making as many distinctions between kinds of code as possible; on the other hand genericity suggests identifying pieces of code in so far as they do essentially the same thing. For this reason, the type systems which organise programming languages in which it is feasible to write general purpose programs in a modular fashion raise difficult technical questions.

1.6 Overview

The rational design of programming languages is now an area of very intense research, but the basic issues are quite accessible. In this paper I attempt to make them clear by focusing on two novel forms of abstract mathematics which have resulted from the needs of computing. These are

1. the problem of describing in a civilised fashion what is a model of Church's lambda calculus;

2. the problem of giving an account of polymorphic functions.

Section 2 is devoted to the pure lambda calculus and section 3 to typed versions. In each case I have tried to indicate how the ideas bring pressure to bear on traditional views of foundations.

Though I have attempted to keep references to category theory to a minimum, this paper makes propaganda for a view of foundations informed by that subject. Category theory is an essential organising principle in modern computer science, and the effect of computing on foundations is best seen in the light of that experience.

2 The Pure Theory of Functions

2.1 The General Notion of a Function

What is a function? With the questionable benefit of hindsight we can see that this question played a significant role in the development of modern mathematics. We associate with Dirichlet the example of the function of a real variable taking the value 1 on rational and 0 on irrational numbers. It seems but a short step from this to the general idea of a function being determined by its graph: that is to the modern set-theoretic notion of function. Probably this is an entirely superficial history of ideas; the example predates serious set theory. However it seems clear that mathematicians in the past did not work with the idea of a function sanctioned by current ideology. We can well imagine that they expected functions to be defined by formulae. (One might liken this to the notion of propositional function as it appears in *Principia Mathematica.*)

In teaching, we are inclined to present the various kinds of function which arise in branches of mathematics as parasitic on the set-theoretic notion of a function as a graph. In analysis we have functions, continuous functions, differentiable functions, analytic functions. In algebra, we have homomorphisms between different kinds of algebraic structure. There is nothing wrong with this set-theoretically based mathematics. Indeed I teach it with pleasure. However we should not suppose that the set-theoretic notion is the only possible basic notion of function. Where we see it under strain (e.g. rational mappings in algebraic geometry, random sequences in probability) we should detect the need for other foundations.

2.2 Computable Functions

Since the advent of computing, we all think that we know what computable functions are: functions which 'in principle' can be computed on a digital computer. (Of course 'in principle' covers a multitude of sins. The computer must be possessed of unlimited memory and be capable of running for an unlimited time.) Unfortunately our intuitive understanding does not readily give rise to a useful theory of computable functions. Classical recursion theory [18] treats the subject, but its aims are limited; in particular attention is largely restricted to the data type of the Natural Numbers. Furthermore most generalised recursion theory equally fails to address issues which are significant to the practice of computing. Among these issues are those of *Intentionality, Non-termination, Fixed points* and *Effectivity*. (I am unable to provide a comprehensive treatment of these, but I list them to give an impression of the range of the conceptual problems in just one aspect of theoretical computer science.) It seems best therefore to treat computable functions via a theory of functions stripped to its bare bones, that is, via the pure lambda calculus.

2.3 The Syntax of the Lambda Calculus

This formal system was constructed around 1930 by Alonzo Church. He intended it to be a foundation for mathematics based on a universal theory of functions. Church's original system incorporated a system of logic at the same level as the functions, and turned out to be inconsistent [11]. The fundamental observation is that functions in the lambda calculus have fixed points; a fixed point for negation is a proposition equivalent to its own negation, and so is a contradiction. However the part of the theory which deals only with function application and abstraction is (in a suitable sense) consistent. This theory is called the *pure lambda calculus*; it encapsulates a pure theory of functions.

The lambda calculus is a theory of *terms*. We suppose that we are given a (countably infinite) set of variables $x, y, z, \ldots$. The set of terms of the pure lambda calculus is then defined recursively by the following clauses:

(Base clause)	*a variable is a term;*
(Application)	*if s and t are terms then (st) is a term;*
(Abstraction)	*if r is a term and x a variable then $(\lambda x.r)$ is a term.*

In the application clause, we think of s as a function, t as its argument (input) and then (st) is the value of s at t (the output). In the abstraction clause, we imagine that for each possible input x, we have a corresponding output $r(x)$, so that

$$x \longrightarrow r(x)$$

is a function from inputs to outputs; then $(\lambda x.r)$ denotes this function-as-object. It is a feature of this theory that functions of many arguments can be reduced to functions of one. For example

$$(x, y) \longrightarrow f(x, y)$$

reduces to

$$x \longrightarrow (y \longrightarrow f(x, y))$$

and so is represented by the term

$$(\lambda x.(\lambda y.f)).$$

This motivates the standard bracketing convention in the lambda calculus:

$$st_1t_2...t_n \text{ stands for } (...((st_1)t_2)...t_n).$$

(Implicitly here s is a function of n arguments so that we think of $st_1t_2...t_n$ as $s(t_1, t_2, ..., t_n)$.) There is a corresponding convention for iterated abstractions:

$$\lambda x_1x_2...x_n.r \text{ stands for } (\lambda x_1(\lambda x_2(...(\lambda x_n.r)...))).$$

(We think of $\lambda x_1x_2...x_n.r$ as an n-argument function abstracted from r.) We use these conventions and also drop brackets where they do not add anything in the rest of this paper.

2.4 Computing with the Lambda Calculus

The main computation rule for the lambda calculus is the following rule of β-equality

$$(\lambda x.s)t = s[t/x].$$

Here $s[t/x]$ is the result of substituting term t for all free occurrences of x in s, with stipulations which prevent 'dynamic binding' of the variables free in t. So λ acts as a variable binding operator. (There is also a rule of η-equality, which makes for cleaner semantics; but it is not computationally essential, so I will not discuss it.) The computation process which results is called β-reduction. A term u reduces in one step to a term v just when v is the result of replacing a subterm of form $(\lambda x.s)t$ by the corresponding one of form $s[t/x]$ in the term u. We write $u \geq v$ when v is obtained from u by a sequence of one step reductions; in particular $(\lambda x.s)t \geq s[t/x]$.

We illustrate the computation process using the traditional Church numerals. We define inductively terms $f^n(x)$ of the lambda calculus for n a natural number by

$$\begin{aligned} f^0(x) &= x \\ f^{n+1}(x) &= f(f^n(x)). \end{aligned}$$

We can associate with each natural number n the term

$$\lambda f.\lambda x.f^n(x)$$

which encapsulates the notion of n-fold iteration. This term is called the Church numeral for n. We write $\hat{n}$ for the Church numeral $\lambda f.\lambda x.f^n(x)$.

One can compute with Church numerals as codes. We give a formal statement of this result. (The notion of partial recursive function is explained in the classic text book by Rogers [18], and in many more recent books on logic. Simpson's paper in this volume also discusses recursive functions [21]).

Theorem 2.1 *For any partial recursive function f there is a term t_f of the lambda calculus such that*

$$t_f\hat{n} \textit{ reduces to } \hat{m} \textit{ if and only if } f(n) = m.$$

And conversely, if t is a term of the lambda calculus, then the function f defined by

$$f(n) = m \textit{ if and only if } t\hat{n} \textit{ reduces to } \hat{m}$$

is partial recursive.

This was essentially first proved (but for a definition of recursive function in terms of primitive recursion and minimalization) by Kleene, but the most natural proof procedure is that given by Turing [22]. By way of an example, we write down one of the simplest codes for an arithmetical function in the lambda calculus, and perform a computation with it. We define

$$mult = \lambda abf.a(bf),$$

and compute $mult\hat{2}\hat{3}$ as follows. (Here we adopt the convention that a reduction is denoted by $\geq$.)

$$\begin{aligned} (mult\hat{2})\hat{3} &= ((\lambda abf.a(bf))\hat{2})\hat{3} \\ &\geq (\lambda bf.\hat{2}(bf))\hat{3} \\ &\geq \lambda f.\hat{2}(\hat{3}f) \\ &= \lambda f.\hat{2}((\lambda h.\lambda z.h^3(z))f) \\ &\geq \lambda f.\hat{2}(\lambda z.f^3(z)) \\ &= \lambda f.(\lambda g.\lambda x.g^2(x))(\lambda z.f^3(z)) \\ &\geq \lambda f.\lambda x.(\lambda z.f^3(z))^2(x)) \\ &= \lambda f.\lambda x.(\lambda z.f^3(z))((\lambda z.f^3(z))(x)) \end{aligned}$$

$$\begin{aligned} &\geq \lambda f.\lambda x.(\lambda z.f^3(z))(f^3(x)) \\ &\geq \lambda f.\lambda x.f^3(f^3(x)) \\ &= \lambda f.\lambda x.f^6(x) \\ &= \hat{6}. \end{aligned}$$

(I do not expect the reader to be very impressed by this computation.)

2.5 Models for the Lambda Calculus

The good answer to the question 'what is a model of the lambda calculus?' is suggested by our basic intuition: the lambda calculus is a theory of *all* functions. Thus everything must be both a potential argument (input), a potential value (output) and a function from arguments to values; what is more all functions are assumed to occur. Thus a model for the lambda calculus should be a set D which is equal to the set D^D of all functions from D to D. (In fact to model β-equality alone, a retraction from D to D^D will do.) Unfortunately unless the set D has just one element D^D can never be even a retract of D; the cardinality of D^D is too great (this is essentially Cantor's Theorem). Thus there are no non-trivial models of the lambda calculus in the intuitive sense.

For this reason, a bad answer to the question 'what is a model for the lambda calculus?' is used in much of the literature. Typically this amounts to the following: a set D and subset $F \subseteq D^D$ together with maps

$$ap : D \times D \longrightarrow D \qquad \text{and} \qquad rep : F \longrightarrow D$$

such that the equations of the lambda calculus are satisfied when *ap* is used to model application and *rep* to model lambda abstraction. Saying this in any precise form is clumsy and conceptually unilluminating; for we have lost the basic intuition of a theory of all functions. There is a discussion of the various primitive definitions of a model for the lambda calculus in Chapter 5 of Barendregt's book[1].

These conceptual problems are a partial explanation for the fact that during the early history of the lambda calculus the focus of interest was on the purely syntactic properties of the computation rules. Of course the pioneers understood the problems, and implicitly responded to them with the thought that the lambda calculus must be a theory of a countable world of intensional functions. This is a very interesting idea, which in principle should have had a profound effect on foundational questions. However the lack of a traditional semantics for the lambda calculus meant that few took the idea seriously.

What counts as serious semantics for some syntax is not an absolute mathematical question; it is very much a matter of tacit agreement by the mathematical community. Any answer reflects the view of the community as to the natural kinds of structures available in mathematics. Even our picture of the relation between syntax and semantics changes: there is no longer such a clear distinction between them. Indeed for the lambda calculus the computation rule of β-reduction provides what computer scientists (unfortunately) call an 'operational semantics'.

The first mathematical model for the lambda calculus was discovered by Dana Scott [19]. (For more on the background see 2.2 of Johnson's paper in this volume [9]). Scott's approach was typical of modern conceptual pure mathematics; he first found an appropriate category in which to work. In these categories continuity is used as an analogue of effectivity, and this keeps control of the structure of function spaces. Other kinds of structure have since been used for this purpose, and most of the 'categories of domains' that provide 'denotational semantics' for functional programming languages contain objects D with categorical function space D^D isomorphic to D.

2.6 Topos Theoretic Models

There is a sense in which the semantics initiated by Scott is unsatisfactory. We have to be conscious of types as 'sets with structure'. The necessary structure is not itself represented in the programming language, but rather (at best) is a reflection of some intuition about how computations are carried out. This conflicts with our original intuition of types-as-sets.

The right response seems to be to adopt the more flexible notion of set inherent in the 'universes of constructive mathematics' called toposes. A category consists of 'objects' and 'maps' (or 'morphisms' or 'arrows') from one object to another; there are 'identity maps' for a notion of 'composition' with obvious axioms but no further structure. (Mac Lane's book [14] remains the best introduction to category theory for the mathematically educated.) Intuitively a topos is a category equipped with such structure as to make it an abstract category of sets and functions. Formally a topos may be defined to be a category equipped with (i) all finite limits, and (ii) power objects. One can think of the objects of a topos as (constructive) sets and the morphisms (or maps or arrows) as (constructive) functions between sets. The epithet constructive is in order as the internal logic of the topos is intuitionistic logic; the lattices of subobjects of an object may be Heyting algebras. The finite limits provide some finitary constructions on sets, while the power objects provide a full power set (set of all subsets) with associated membership relation. This gives a very rich essentially set-theoretic or (perhaps better) type theoretic structure. The standard reference to topos theory is still Johnstone's book [10], but see Bell's book [2] for a good account of the logician's perception of a topos as a world of constructive mathematics (model of type theory or local set theory).

Now, if we are prepared to take an undogmatic approach to foundations, we can readily recapture our basic intuitions about the lambda calculus. For the cardinality problems associated with Cantor's Theorem do not bite in constructive mathematics. There it is perfectly consistent that there be sets D which are equal to (or better, isomorphic to) the set D^D. In such models we have the technical and conceptual advantage of being able to argue (albeit constructively) as if we were dealing with arbitrary sets and functions. A description of the simplest kind of topos, in which one can find objects (that is constructive types) D with D^D isomorphic to D is given in [20]. (They are 'toposes of presheaves': they capture a very primitive notion of 'variable set'.)

Now topos theory does more than make our mathematical models run more smoothly. For once we can conceive of a world of sets in which there exist non-trivial models of the the pure lambda calculus, we come naturally to regard it as a defect of the classical universe of sets that it contains no such model. Considerations of this kind may lead to quite radical forms of relativism. (See the preface of the book by Lambek & Scott [13] for a hint of such a position.) But we need not go that far. Once we have an interest in alternative formal systems which can serve as a foundation for mathematics, the absolute status of the classical set-theoretic foundations must come into question.

3 The Theory of Types

3.1 Typed Programming Languages

The value of types in a programming language is that they provide a basic guide to the programmer who needs to structure complex programs. In particular in many typed languages it is possible to detect syntax errors at compile time, that is effectively as the program is being constructed. The usual analogy is with dimensional analysis in physics. We can write

$$\begin{array}{rcl} x & \in & L \\ v & \in & LT^{-1} \\ m & \in & M \\ F & \in & MLT^{-2} \end{array} \tag{3.1}$$

to signify that x is a length, v a velocity, m a mass and F a force. Then we can deduce that

$$\frac{1}{2}mv^2 \in ML^2T^{-2}$$

and that

$$Fx \in ML^2T^{-2}$$

so that

$$\frac{1}{2}mv^2 = Fx$$

makes sense, while

$$\frac{1}{2}mv^2 = F$$

does not.

3.2 The Simple Typed Lambda Calculus

For our purposes we only need consider the type structure of this theory, and not the associated computation rules which give it meaning. (These are in fact just the rules of the pure lambda calculus restricted to typed terms.)

We start with some basic types and their associated constants and functions. For the simplest kind of programming, these might be the types *Nat* and *Bool* of Natural Numbers and Booleans, together with some basic constants and functions and with definition by cases (*if...then..., else...*). But the details are not important. The collection of all *Simple Types* is then generated by the single rule:

if A and B are types then so is $(A \to B)$.

The idea is that $(A \to B)$ is the type of all functions from elements of type A to elements of type B. Corresponding to this are the rules

($\to$-elimination) if $s \in (A \to B)$, and $t \in A$ then $(st) \in B$;

($\to$-introduction) if $r \in B$ given $x \in A$, then $\lambda x.r \in (A \to B)$.

Now we are forced to distinguish $f \in (Nat \to Bool)$ from $g \in (Nat \to Nat)$ and can apply neither to $t \in Bool$.

The reader may find it useful to check that in this calculus each of the Church numerals of 2.4 can be given the type

$$(A \to A) \to (A \to A)$$

independently of what A may be. This is a hint of genericity which cannot be explicitly handled by the simple typed lambda calculus.

The mathematical structure needed to model this calculus with full $\beta\eta$-equality is that of a cartesian closed category. The connection is explained in full detail in the book by Lambek and Scott [13]. There is a plentiful supply of cartesian closed categories, including the familiar category of sets. Thus the semantics of the simple typed lambda calculus is unproblematic. (Of course, it is not so straightforward to find models useful in computer science.)

3.3 The Syntax of the Second Order Lambda Calculus

It is altogether more problematic to describe the semantics of a system in which the genericity of functions in the simple typed lambda calculus is made explicit. Here is a brief sketch of the simplest such extension of the simple typed lambda calculus described above.

To the rules for *Simple Types* we add type variables $X, Y, Z, \ldots$,and a further rule of type formation:

if A is a type and X is a type variable, then $\Pi X.A$ is a type

This gives us the collection of *Second Order Types*. The idea of the new rule of type formation is as follows. Call types with no free type variables constant types, and suppose for simplicity that A has at most the variable X free. Then for each constant type B there is a (constant) type $A[B/X]$ where B has been substituted for X in A. Then $\Pi X.A$ is the type of all (choice) functions from the collection of all constant types B which pick an element of the corresponding type $A[B/X]$. (So of course X is bound in $\Pi X.A$).

We simultaneously add further operations on terms:

(2nd order application) if s is a term and B a type then (sB) is a term;
(2nd order abstraction) if r is a term and X a type variable then $(\lambda X.r)$ is a term.

Then there are the typing rules:

(Π-elimination) if $s \in (\Pi X.A)$, and B is a type then $(sB) \in A[B/X]$;
(Π-introduction) if $r \in A$ with X a type variable, then $\lambda X.r \in (\Pi X.A)$.

The intended meaning of the term-forming operations should be clear enough from the typing rules. The meaning of second order application is that $s \in (\Pi X.A)$ denotes a function from types X to elements of $A(X)$, and then $(sB) \in A[B/X]$ denotes its value at B. To understand second order abstraction, we imagine that as X varies over types, r takes values in the corresponding types $A(X)$; then $\lambda X.r \in (\Pi X.A)$ denotes the corresponding function-as-object. (The new notions of abstraction over types and of application of terms to types give rise to new computational rules of $\beta\eta$-equality, but we do not go into these here.)

The extended system of types and terms which we have just described is the *second order lambda calculus*. It was first considered by Girard in the course of proof theoretic investigations [6] and rediscovered by Reynolds in the context of computer science [16].

3.4 Computations in the Second Order Lambda Calculus

In the second order lambda calculus we explicitly have generic types. For example, for every natural number n, we have

$$\lambda X.\hat{n} \in \Pi X.(X \to X) \to (X \to X),$$

using the notation of 2.4. In fact we can regard $\Pi X.(X \to X) \to (X \to X)$ as an implementation of the natural numbers, and so write $\mathbb{N}$ for this type. And we write $\bar{n}$ for $\lambda X.\hat{n}$.

Theorem 3.1 *Up to $\beta\eta$-equality the only terms of type $\mathbb{N}$ are of the form $\bar{n}$.*

This result says in effect that there are no non-standard natural numbers. The first explicit statement and proof that I know is rather late [3]. However the great expressive power of the second order lambda calculus was known to Girard [6] which includes (amongst many other things) a characterisation of the computations which can be coded as terms of type ($\mathbb{N} \rightarrow \mathbb{N}$).

Theorem 3.2 *For any function f, provably recursive in analysis, there is a term $t_f \in (\mathbb{N} \rightarrow \mathbb{N})$ of the second order lambda calculus such that*

$$t_f\bar{n} \text{ reduces to } \bar{m} \text{ if and only if } f(n) = m.$$

And conversely, if $t \in (\mathbb{N} \rightarrow \mathbb{N})$ is a term of the second order lambda calculus, then the function f defined by

$$f(n) = m \text{ if and only if } t\bar{n} \text{ reduces to } \bar{m}$$

is provably recursive in analysis.

Here, as usual in logic, 'analysis' refers to any formalisation of second order arithmetic with full comprehension. The provably recursive functions of analysis form a large class of functions containing all those total recursive functions which seem likely to arise in practice (and many many more).

3.5 Models of the Second Order Lambda Calculus

The question of what is a good notion of model for the second order lambda calculus is quite different in detail from the corresponding question for the pure lambda calculus, which we considered in section 2. However it is similar in impact and causes the same kind of heartache. There is an answer in accord with our intuitions: essentially we must have a collection of sets and (all) functions closed under sufficiently large products. (For the natural interpretation of $(\Pi X.A)$ is as the product of all sets $A(X)$.) But there turn out to be straightforward cardinality problems with this idea within the world of classical set theory. (A more general problem with models of the second order lambda calculus emerges from the analysis in a paper by Reynold [17].) As with the pure lambda calculus, the result is that many people work with a conceptually unilluminating answer. (The details are too awful to be worth sketching here; they can be found in the paper by Bruce & Meyer [4].) But again this is a situation in which constructive mathematics comes to the rescue. As first suggested by Eugenio Moggi, sufficiently complete collections of sets and functions do exist in suitable constructive universes. (For a general construction of such toposes see Pitts' paper [15].)

One particular kind of constructive mathematics (based on 'realizability') in which we can model the second order lambda calculus is very attractive. One of the least appealing features of traditional category theory (as developed in Mac Lane [14] for example) is the need for size restrictions (the solution set condition) in the fundamental Adjoint Functor Theorems. But these are not necessary for small categories. The problem for traditional category theory is that the only small complete categories are preordered sets. However, there are toposes containing very rich small complete categories and for these, the Adjoint Functor (and related) Theorems can be exploited in the very simple form appropriate to small categories. I give a sketch of this perspective is in [8].

Overall, the effect of the constructive view of models of the second order lambda calculus is much the same as that of the constructive view of the pure lambda calculus. We are driven emotionally from any view which gives the less conceptually rich world of classical set theory any primary status.

4 Conclusions

4.1 Universes of Constructive Mathematics

In this paper I have described how the ideas in a particular area of computer science make non-standard constructive worlds of mathematics seem very attractive. I am myself struck by how closely some of these mirror views about foundations sketched long ago by Kolmogorov in 1932 [12].

It is worth emphasising that the way we usually present toposes presupposes a definite classical set theory. However this is in no way essential and cannot be used to give any primacy to classical set theory. (Equally, I believe there are no strong arguments for any other kind of foundations.)

Quite generally, the concepts of classical set theory are inappropriate as organising principles for much modern mathematics and dramatically so for computer science. The basic concepts of category theory are very flexible and prove more satisfactory in many instances. As rightly stressed in 2.3 of Johnson's paper in this volume [9] much category theory is essentially computational and this makes it particularly appropriate to the conceptual demands made by computer science.

It is true that category theorists (in particular topos theorists) have for the last twenty years been engaged in activities that are antipathetic to traditional foundations. However since in most cases these activities have involved the exploitation of nothing more subtle than the possibility of doing algebra in a topos, the effect on our thinking about foundations has been small. The honourable exception is the study of Synthetic Differential Geometry initiated by Lawvere, where spaces of functions are exploited in a natural and elegant fashion. Regretably this subject remains a minority interest. By comparison, the radical conceptual demands made by the concrete practice of computing have given the process of rethinking foundational questions a quite definite focus.

The emerging view does not make set theory redundant; it may remain a crucial component of the foundations of mathematics. But its place in the scheme of things looks radically different. In particular, the idea that there should be some definite foundation for mathematics in the traditional sense looks less secure.

4.2 Other Aspects of a Revolution?

I should emphasise that the story that I tell here is both partial and far from over. In the first place I have had to omit serious discussion of a number of areas of mathematical logic which have recently been transformed by questions arising in computer science. (In particular I regret not being able to discuss Girard's linear logic, which has recently challenged our view of what logic is. The curious reader might like to compare Girard's paper [7] with traditional books on logic.) And secondly we are in the midst of very exciting times in the development of an abstract mathematical view of logic. Mathematicians may still refer to category theory as abstract nonsense, but the IT revolution has transformed this abstract nonsense into a serious form of applicable mathematics. In so doing, it has revitalized logic and foundations. The old complacent security is gone. Does all this deserve to be called a revolution in the foundations of mathematics? If not maybe it is something altogether more politically desirable: a radical reform.

References

[1] Barendregt, H. P., (1981). *The Lambda Calculus, its Syntax and Semantics.* North-Holland, Amsterdam.

[2] Bell, J. L. (1988). *Toposes and Local Set Theories.* Clarendon Press, Oxford.

[3] Böhm, C. and Berarducci, A. (1985). 'Automatic synthesis of Typed Λ-Programs on Term Algebras' *Theoretical Computer Science* 39, 135-154.

[4] Bruce, K. B. and Meyer, A. R. (1984). 'The semantics of second order polymorphic lambda calculus.' In: G. Kahn et al (eds), Semantics of Data Types, *Lecture Notes in Computer Science 173,* (Springer-Verlag, Berlin), 131-144.

[5] Burstall, R., (1991) 'Computer assisted proof for mathematics: an introduction using the LEGO Proof System', in *The Mathematical Revolution Inspired by Computing,* J.H. Johnson & M.J. Loomes (eds), Oxford University Press, (Oxford)

[6] Girard, J.-Y. (1972). 'Interprétation Fonctionelle et Elimination des Coupures dans l'Arithmétique d'Ordre Supérieur.' Thése de Doctorat d'Etat, (Paris).

[7] Girard, J.-Y. (1987). Linear Logic. *Theoretical Computer Science* 50, 1-102.

[8] Hyland, J. M. E. (1988). 'A small complete category.' *Annals of Pure and Applied Logic,* 40, 135-165.

[9] Johnson, J. H., (1991) 'An introduction to the mathematical revolution inspired by computing', in *The Mathematical Revolution Inspired by Computing,* J.H. Johnson & M.J. Loomes (eds), Oxford University Press, (Oxford)

[10] Johnstone, P. T. (1977). *Topos Theory* Academic Press, (London).

[11] Kleene, S. C. and Rosser, J. B. (1935). 'The inconsistency of certain formal logics'. *Annals of Mathematics* (2) **36**, 630-636.

[12] Kolmogorov, A. N. (1932). 'Zur Deutung der intuitionistischen Logik'. *Mathematische Zeitschrift* **35**, 58-65.

[13] Lambek, J. and Scott, P. J. (1986). *Introduction to higher order categorical logic.* Cambridge University Press, (Cambridge)

[14] Mac Lane, S. (1971). *Categories for the Working Mathematician.* Springer-Verlag, (Berlin)

[15] Pitts, A. M. (1987). 'Polymorphism is Set Theoretic, Constructively'. In: D. H. Pitt et al (eds), *Category Theory and Computer Science, Lecture Notes in Computer Science 283,* Springer-Verlag, (Berlin), 12-39.

[16] Reynolds, J. C. (1974). 'Towards a Theory of Type Structure.' In: Programming Symposium, *Lecture Notes in Computer Science 19,* Springer-Verlag, (Berlin), 408-425.

[17] Reynolds, J. C. (1984). 'Polymorphism is not set-theoretic.' In: G. Kahn et al (eds), Semantics of Data Types, *Lecture Notes in Computer Science 173,* Springer-Verlag, (Berlin), 145-156.

[18] Rogers, H. (1967). *Theory of Recursive Functions and Effective Operations.* McGraw Hill, (New York)

[19] Scott, D. S. (1972). 'Continuous Lattices.' In: F. W. Lawvere (ed), *Toposes, Algebraic Geometry and Logic, Lecture Notes in Mathematics 274*, Springer-Verlag, (Berlin), 97-136.

[20] Scott, D. S. (1980). 'Relating theories of the lambda calculus.' In: J. R. Hindley & J. P. Seldin (eds), *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism* (Academic Press, London), 403-450.

[21] Simpson, D. (1991) 'A Euclidean Basis for Computation', in *The Mathematical Revolution Inspired by Computing*, J.H. Johnson & M.J. Loomes (eds), Oxford University Press, (Oxford)

[22] Turing, A. M. (1937). 'Computability and λ-definability.' *Journal of Symbolic Logic,* **2**, 153-163.