INVESTIGATIONS OF TYPE THEORY IN PROGRAMMING LOGICS AND INTELLIGENT SYSTEMS

Robert L. Constable

TR 85-701
September 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853
INVESTIGATIONS OF TYPE THEORY
IN PROGRAMMING LOGICS AND INTELLIGENT SYSTEMS

Robert L. Constable
Cornell University

Abstract

Type theory has become central to computer science because it deals with fundamental issues in programming languages, in programming methodology and specification languages, in automatic theorem proving and programming logic, in natural language semantics and in the foundations of intelligent systems. At Cornell we have been studying a logical theory of types which has influenced the design of programming languages and has become the basis of an implemented program development system. This theory answers many basic questions about data types.

Here we discuss three general questions about this theory: how logical types relate to domains, how they relate to sets, how they organize programming logics and the intelligent systems built around them. These issues are each of independent value, but they also arise naturally as part of a program to provide a computationally meaningful foundation to computing theory.

1. Introduction

Type Theory and Programming

Understanding data types is critical to understanding modern programming languages and their logics. The study of types has become quite deep and increasingly active; there are now several major research efforts both practical and theoretical in this area (e.g. at MIT, CMU, Bell Labs, IBM, Syracuse, Rice, Edinburgh, London, Manchester, Oxford, INRIA, Goteberg, and Stockholm, to name only groups of two or more people with which we have had direct contact last year). Some basic questions are still unresolved or require further study: what is a data type, what are the practical advantages of various definitions, what kinds of polymorphism are possible, how difficult is it to check type-correctness according to the various definitions of type? Some questions were only recently asked: how are the data types of a language related to the logical types, what new types are
suggested by new modes of computation (lazy evaluation, message passing), how are types used to
hide information, both in the computational mechanisms and in the logical mechanisms, how can
large type theories be organized? Answers are still not clear, although the subject is broader than it
was in 1974 - now types can be defined as sets or algebras or categories or retracts or ideals or inter-
vals or logical types! The title "Types are not sets [61]" would not be regarded now as an announce-
ment of new ideas as much as an estimation of the odds.

The study of types has also arisen in programming methodology through a need for precise
specification languages. Systematic means of data type refinement, similar to program refinement,
are being studied. Type theory has also been important in automatic theorem proving [5], in formal
semantics [75] and of course in mathematical logic [72] where the subject arose. The questions,
requirements, and results of all of these areas have converged to make type theory a central concern
of computer science.

*Cornell Work*

The centerpiece of our theoretical research on types is a completely formal constructive theory
of types. This theory has evolved over many years [17,20,21,23]. We think of it now as a fundamen-
tal language for computing in the way the original nonconstructive type theories [87] are a founda-
tion for mathematics. From a purely theoretical point of view, our theory is noteworthy because it is
*comprehensive and uniform*; we claim to be able to naturally formalize any sequential programming
problem, yet there are only 9 basic type constructors (which could be reduced to 6 with some loss of
convenience). These are characteristics shared with the theories to which it is closely related, the
AUTOMATH languages [29,30,85], the intuitionistic type theories of Martin-Löf [56,76], and the
theory of species of Girard [40,81]. But our theory differs from these in several crucial respects
because of its origin as a programming logic (as described in Constable [21]). Some of the differences
are mentioned in this proposal, e.g. treatment of sets, recursive types, recursive functions and func-
tion equality (see sections 3 and 4 below).

One of the most significant aspects of this theory is that it "really works." Around it we have
built a system which provides an environment for writing formal constructive mathematics and for
developing programs. The system, called Nuprl ("new pearl") [23], took four man-years to build. It consists now of over 45,000 lines of Lisp plus 10,000 lines of imported ML code from Edinburgh LCF. Unlike any other equally comprehensive theory, Nuprl is being used to formalize computational problems in mathematics. With it we have already described a number of nonelementary problems, including Cantor’s theorem, a theory of finite discrete sets, a denotational semantics for a simple programming language and the basics of the theory of constructive real numbers. We have also expressed a number of elementary problems in the theory of integers, lists and arrays. The system is also being used at Edinburgh as part of the LCF project.

The type theory of this system has had some impact in programming language design (Russell, Pebble), and in programming methodology. We think that the theory sheds light on the questions raised under the heading *Type Theory and Programming*, and we will discuss these further in sections 3 and 4. It also sheds light on certain basic questions about systems which try to provide deductive and symbolic assistance in problem solving, say *intelligent systems*.

*Type Theory and Intelligent Systems*

Our experience in expanding the Nuprl implementation from a first-order theory of numbers and lists to a higher-order theory of types revealed an interesting phenomenon — that some of the algorithms supporting the first-order theory generalized considerably in the higher-order context and thereby amplified the computing power applied to the system. The algorithms used to decide equality and to fill in trivial steps in an argument are a good example; the same tactics used for trivial reasoning also manage to prove elementary facts in higher-order logic and facts about data types. There are other examples of this phenomenon. It seems to depend on the uniformities of the theory. (In Nuprl sets, types and propositions are variants of the same concept.) The techniques would not apply if the system were made expressive by putting together separate pieces, e.g. predicate calculus, then numbers, then sets, then algorithms. It is noteworthy that the phenomenon does not appear at all if we look only at simple systems.

Type theory also creates a quantum jump in the capabilities of the full Nuprl system over its first-order version because it can conveniently describe important parts of its own metalanguage. In
this case the phenomenon is again natural; part of the metalanguage is the programming language ML from Edinburgh LCF. This language is implicitly contained in the type theory.

We have mentioned two ways in which the use of type theory as the language of an intelligent system leads to improvements in the performance of such a system. We think that this theory will help us understand the important question of how language effects the organization, capabilities and performance of such systems generally. In section 5 we propose specific studies.

Organization of the Paper

This is a discussion of some of the theoretical questions that have arisen during explorations with Nuprl, especially while considering the relationship of our theory to others and while pondering the answers that Nuprl offers to long standing issues. In the remainder of the paper we discuss four aspects of type theory; its role in foundations of computing theory, its relationship to constructive theories of sets, its relationship to domain theory, and its role in intelligent systems. Specific technical questions are asked in each category, labeled as Q1, Q2, ..., Q9.

2. Constructive Foundations for Computing Theory

In computer science, we are principally interested in computation. We seek efficiently computable functions and feasibly constructible objects. We worry about how these objects are represented and manipulated. The meaning of our languages, even their declarative parts, is ultimately to be found in computer implementations and in the information processing power of machines.

The foundations of the subject, however, are laid on ordinary noncomputational mathematics. So computable functions are defined by restricting the class of all functions. In denotational semantics, for example, although Dana Scott has been sensitive to giving computational explanations, his influential theories are nearly always interpreted noncomputationally [36,42,51,82]. So partial recursive functions, say from natural numbers, \( \mathbb{N} \), to \( \mathbb{N} \) are defined denotationally as total noncomputable functions on the domain \( \mathbb{N} \cup \{ \perp \} \). Data types are frequently defined as restrictions of some inherently infinitary notion of set, and people are confused as to whether there can be such a thing as "real infinite sets" used computationally. In some cases we see people advocating that "programs"
be written first without regard for computability [2], say in a language such as ZF set theory (specifically in the language Z used at Oxford [1]). Such steps are taken in the belief that only classical mathematical formalisms like ZF allow sufficiently high levels of abstraction [2].

For some time I have been interested in whether a computationally meaningful foundation can be laid for computer science and whether providing it would be beneficial. There is strong evidence that this is possible and desirable. First, there is the direct evidence that we use very robust informal concepts of algorithm, data type, representable object, and provable formula in our informal work. These ideas are very clear without translation into set theory. As indirect evidence we know that constructive mathematics is a coherent subject which can in fact provide a meaningful explanation of large parts of classical mathematics. It seems likely that it will explain all of those parts needed in computer science. (I know of no counter example.) Starting with the constructive ideas will lead more directly to the concepts important for computer science, and will minimize the confusion caused because ordinary mathematical language loses track of computational meaning.

Whether or not this program of seeking a computational foundation for computing theory is philosophically appealing, it has led to a number of results which answer basic questions, such as how to compute with infinite sets, how to reason about partial functions (compare [22,26] and [51]), and how to organize a theory of data types. It has also provided computationally meaningful formalisms as rich as Z or as constructive set theory [62] for defining programming problems. In such formalisms one gains the advantage of great abstraction without sacrifice of computability. Indeed all functions in the theory are computable. It is noteworthy that these overtly constructive solutions are different than the classical ones, e.g. those in LCF or in Z, and they suggest alternative classical solutions.

As this program has matured over the years, it has led to tangible results from a number of quarters. A constructive approach to recursive function theory [18] for example, influenced independence results in complexity theory [34]. A constructive approach to programming methodology led

\[\text{\textsuperscript{4}In fact in my experience with undergraduates in computer science, I find that the translation into set theory is much less clear than direct contact with programs. Programs offer a much better formalization of computable functions than their translation into relations and then into sets and then into Turing machines.}\]
to the notion of a *programming logic* [19] as an alternative to Hoare logic, the wp-calculus, etc. Certain theoretical insights independently arrived at from this point of view [19,68] have been subsumed in "dynamic logic" [43,64]. In the last four years these ideas have led to a substantial programming system, Nuprl [69] in which one mode of programming consists of writing constructive proof outlines for $\forall$E formulas [7,69].

We suggest pursuing this program further. The specific topics discussed in sections 3 and 4 contribute to it, but in addition there are two direct questions along these lines. In the first place, LCF [42] can be seen as a foundational logic for computer science based on Scott's domain theory. But it is a classical logic with classical semantics. We have defined a constructive version of LCF [25] and have shown it to be realizable in Kleene's sense [47], so it is a consistent and sensible theory.

(Q1) We suggest examining constructive LCF further - to develop some basic theorems in it, to show that LCF can be recovered by adding the axiom PνP, and to give a type theoretic semantics to it. These results will succinctly illustrate the contrast between a classically justified theory of recursive functions and a computationally justified theory.

(Q2) In the second place, David McCarty [58] and Dana Scott have recently considered intuitionistic Zermelo/Fraenkel (IZF) set theory as a foundation for domain theory. We too have studied a computational denotational semantics. N.P. Mendler has defined a denotational semantics for while-schemes in Nuprl. This semantics is at once "denotational" and computable since the theory in which he described it is constructive. Moreover since Nuprl offers a set constructor, it appears fruitful to compare in detail Nuprl and IZF (see section 4) as a basis for "computable denotational semantics".

3. Domains and Types

The concept of a domain, say as a cpo [65,77,78,82], has become fundamental in the semantics of programming languages. Most basically, the concept of a partial recursive function (of any finite type) is usually defined on domains as the least fixed point of a functional. This essentially handles all control structures. Then certain universal domains such as a $P\omega$ or $T\omega$ have been used to give a
rigorous account of data types including those recursively defined over the constructors \( + \) (disjoint union), \( \# \) (cartesian product) and \( \to \) (function space). For instance, domain theory can account for types such as \( \text{define } t = t \to t \), and it can define models of the lambda calculus \([65,75]\). Programming languages such as ML \([42]\) and Russell \([32,33]\) provide data types based on domains. We shall loosely refer to these types as \textit{domains}.

Another important definition of type, which for the present we will call \textit{logical type}, has emerged from studies in the foundations of mathematics. Starting with Bertrand Russell and recently taken up for example in automated mathematics and in proof theory and in the study of constructive mathematics. These efforts too have now produced type theories suitable for programming. There is the work of Per Martin-Löf \([56]\) which has led to efforts to design a programming language \([63]\) and is the object of lively interest in logic \([8,35]\). There is the closely related work at Cornell \([20,21,23]\) which has already led to a fully implemented programming language and logic \([23,69]\) and has been the subject of study in logic \([9]\) and computer science \([44]\). We shall say that these languages are based on the idea of a \textit{logical type}.

The "logical" function space \( A \to B \) consists entirely of \textit{total} computable functions. This means that it is not possible to build recursive data types such as \( \text{define } t = t \to t \) with this \( \to \) constructor. It also seems to imply (but see \([22]\)) that partial recursive functions, and hence many natural programs, are not represented among the objects of the theory. This state of affairs seems to place languages based on such a type theory at a disadvantage over those based on domains.

I have argued that there is no theoretical disadvantage to theories based on logical types over theories based on domains because suitable notions of partial function and recursive type can be defined in the former as well. Nevertheless, there is a practical disadvantage because these definitions are not as convenient, for instance partial functions are not given directly as they appear in standard programming languages. Thus there has been a considerable effort to bridge the gap between domains and logical types. Martin-Löf has proposed a \textit{constructive theory of domains} \([55]\); however, he complicates considerably the structure of logical types and gives up on the propositions-as-types principle completely, which is a central concept of his type theory. He is also considering
such notions as "free choice sequences" [83] which are among the most subtle parts of Intuitionism. A number of other computer scientists are attempting to bridge this gap (Gordon Plotkin, Samson Abramsky and Mike Smyth for example). So far few definite results are known.

Recently at Cornell, N.P. Mendler and I created a theory of recursive types and partial functions in the context of a theory of logical types [26]. In fact our concepts can be used to extend either M-L82 or Nuprl without abandoning the propositions-as-types principle and without adding the complexities of approximations found in Martin-Löf's account [54,55]. Our approach is to first define recursive types over +, # and → (even over the dependent # and →), restricting types built with → to be "strongly positive" (e.g. the defined type occurs only as a consequent). Then we can define types which are the domains of partial functions. We introduce the idea of a domain predicate, dom(f), for a partial function f. With this we can define the type of partial functions from A to B, written A ~> B. Then we can use ~> to define recursive types such as rec(t. t~>t) which are the analogue of define t = t → t in domain theory. We have shown that the resulting theory is consistent relative to Nuprl.

The concepts we have formulated will allow us to pursue two important investigations which we plan as specific tasks under this proposal. We describe them next.

(Q3) We plan to extend the concepts of recursive type to include those with infinite objects, e.g. streams, infinite trees, etc. Such types will fit well with the lazy evaluation strategy of Nuprl.

(Q4) We think it is possible to use these ideas to give a constructive semantics to LCF which is simpler than the realizibility semantics mentioned in (Q1) and is an alternative to Martin-Löf's proposal for a constructive LCF (presented in his invited lecture at the International Symposium on Semantics of Data Types in Sophia-Antipolis, France, June 1984).

There are a number of thorny technical problems in this line of inquiry. For example, we want to allow a wide variety of types, not only streams, say definition Strm = A # Strm but these things; definition S = S # S. Now there will be no proper objects in S and only infinite objects in Strm. Among the infinite objects of Strm will be terms such as gen(x.<a,x>) which generates the stream <a,a,a,...>. There will also be then generators like gen(x.<x,x>) in s which are "improper"
because they do not produce any real elements. Are we to exclude such improper generators? If we allow them, as domain theory would, then we allow objects, such as definition \( S = S \# S \) which may contradict the intuitive semantics of logical types. Our approach to partial recursive functions suggests a line of attack, but the details are unclear. However if we succeed we will have taken another significant step to reconcile domain theory and type theory.

4. Sets and Types

Set theory has been a remarkably good language for expressing noncomputational mathematics. There are several commonly known formalizations of set theory which are thought to be adequate for transcribing extant mathematics, e.g. Zermelo/Fraenkel (ZF), Bernays Gödel (BG), Bourbaki set theory [Bour]. The language of sets is especially good for dealing with generalities; for instance the idea of an inductive definition of a subset of \( S \) can be equated with a monotone operator on the power set of \( S \), regardless of the structure of \( S \). Since sets are objects they can be arguments to and values of functions. Sets also provide simple encodings for many basic concepts such as Dedekind cuts, algebraic structures, quotient structures, etc.

But why does set theory work so well? Perhaps if we really knew we could transplant the mechanisms which work into the realm of types and programming. We think that one mechanism making set theory work is a uniform way of “hiding information.” For instance, the separation axiom and its associated construct \( \{ x : A \mid P(x) \} \) specifies information about members. When we prove \( a \in \{ x : A \mid P(x) \} \), the proof of \( P(a/x) \) records the information and is kept available, but is not shown explicitly until it is needed. Attempts to capture the same idea in type theory seem cumbersome because the information is explicit. For example in Martin-Löf’s theory M-L82 [56], the set of objects of type \( A \) satisfying predicate \( B \) is written \( \Sigma x : A . B \) and denotes those pairs \( < a , b > \) such that \( a \in A \) and \( b \in B(a/x) \). So the proof component \( b \) is explicit, and this information is carried around with \( a \).

In the Nuprl type theory, there is an explicit separation constructor, \( \{ x : A \mid B \} \) whose purpose is to hide the information \( b \). Roughly the idea is that \( b \) is available in proofs but not in computations. This idea is implemented by keeping track of dependency information. But the concept of a set type
such as \( \{x:A \mid B\} \) is very subtle in the presence of the other constructors which make information explicit. New ways of showing inconsistency exist in theories like Nuprl where information can disappear and reappear, so the details of such a theory are important.\(^\dagger\)

The study of the set constructor in Nuprl has revealed a number of interesting insights about "information flow" in logical theories. One result by Stuart F. Allen [4] is an alternative propositions-as-types principle [45]. His version accords more intuitively with the usual rules of logic. The idea is to treat a proposition \( P \) as the type \( \{ \cdot \mid \neg P \} \) where \( \cdot \) is some inhabited singleton type such as \( \{0\} \). Thus all proofs of \( P \) belong to \( \{ \cdot \mid \neg P \} \), but we can reason about \( P \) as a true (proved) proposition without knowing the particular proofs. We hope that this approach may eventually allow someone (of us?) to see another mapping of classical logic into constructive logic since this view of propositions is closer to the classical conception yet has a rigorous computational semantics. Moreover, this view of propositions plays an important role in formalizing analysis.

Separation is only one of several set-forming mechanisms used in the standard axiomatizations of set theory. Another is collection, associated with the constructor \( \{f(y) \mid y \in A\} \). This provides a different kind of power, a way of creating new sets from which subsets can be separated. As we work within type theories such as Nuprl and ML82 we are seeing situations where a collection construct might be useful. For example, a collection would provide a way of building types whose elements were types without resort to higher universes. (Universes are the means of collecting together types.) We have examined an interesting form of collection in type theory which is sensible. We need to know if it will really be useful.

Another type theory constructor original with us is the quotient, \( A // E \), of a type \( A \) by an equivalence relation \( E \). This constructor captures the idea of a quotient structure from algebra [11] — a concept easily defined in set theory, but requiring special treatment in type theory. As we have used the apparatus of quotient types in formalizing analysis, we have become aware that it provides a mechanism for abstraction which appears to be more flexible than the usual formulation of abstract

\(^\dagger\)We also use a novel notational device to hide \( b \) implicitly even when we use \( \Sigma x \in A. B \). For example the user can define a new notation such as \( \{x:A \mid B\} \) to mean \( \Sigma x \in A. B \) and write \( a \in \{x:A \mid B\} \) as an abbreviation for \( <a,b> \in \Sigma x \in A. B \). An operator such as \( \text{proof}(a) \) can pick out \( b \) when it is needed.
types or packages. With a quotient we can define operations which are guaranteed not to have access to representations, but we need not decide in advance exactly which operations will be so protected. New representation independent operators can be added at any time.

The following specific questions are suggested by our preliminary investigations into the general question of why set theory works.

(Q5) Using separation and collection in type theory, can we build a model of constructive set theory, say in particular of Myhill's CST [62]. Peter Aczel [3] has built a model of CST without either separation or collection in the type theory; one might hope that these operations would give a more natural model. In such a theory we could compute with infinite sets, and it would be interesting to compare such an account with attempts to compute with sets in domain theory via approximations.

(Q6) Can we discover an exact relationship between the representation hiding capabilities of quotients and those of abstract types (say as presented in [42])?

(Q7) An important issue in type theory is polymorphism or type ambiguity. Already in Principia Mathematica Whitehead and Russell used notation which suppressed type information. Thus their concepts and proofs were polymorphic in the sense that they made sense for "many types" simultaneously. This style of polymorphism is used in the programming language ML and is present in Nuprl and M-L82; for example \(\lambda x.x\) is a function belonging to any type \(A \to A\). It is however not possible in these theories to define the collection of polymorphic functions. We cannot, for example, define the polymorphic functions from \(N \to N\) as those \(f\) for which for all types \(A\), \(f \in A \to A\).

Definitions like this could be made possible in Nuprl in a number of ways. One way is to change the formation rules for the logic so that a proposition such as \(\exists x : A.B\) is well-defined provided \(B(a)\) is well-defined for some \(a\) in \(A\). Consider for example the proposition \(\exists x : \text{integer}.(1/x=1/x)\); is this well-defined? Presently it is not; one needs \(\exists x : \{i : \text{int} \mid i \neq 0\}.(1/x=1/x)\), but we can consistently amend the rules to allow the former. When we do this, we also gain the ability to define types such as \(\{f : \text{int} \to \text{int} \mid \forall A : \text{type}.(f \in A \to A)\}\) which expresses polymorphism.
There are other ways of expressing polymorphism which are based on set theoretic mechanisms. If for example types are regarded as sets over a domain of individuals which include all of the "constructive objects" including functions, then a function is polymorphic when it belongs to many sets. This idea can be approached in several ways, one of which is to use the collection constructor over a universe and a set-like union construction. We would like to pursue polymorphism in these ways and compare them to approaches based on MacQueen, Sethi, Plotkin [50] or Cartwright [15].

It is interesting to note that Nuprl and M-L82 can directly express a predicative version of the so-called parametric polymorphism of Reynolds [71] and Girard [40]. For instance, a polymorphic sorting routine might be defined in Nuprl as

$$\forall A : U_1. \forall R : A \times A \rightarrow U_1 (R \text{ a linear order relation on } A) \Rightarrow$$

$$\exists f : \text{list}(A) \rightarrow \text{list}(A) . f \text{ sorts list}(A) \text{ wrt } R$$

which says that for every type A in any universe, if R orders A, then there is a function that sorts lists over A. The possibility is open here that the operation f depends on A. By expressing the first style of polymorphism we could rule this out, saying in addition

$$\forall B : U_1. \forall Q : B \times B \rightarrow U_1 (Q \text{ is a linear order on } B) \Rightarrow$$

$$\exists f : \text{list}(B) \rightarrow \text{list}(B) . f \text{ sorts list}(B) \text{ wrt } Q).$$

Thus in one theory we could express several variants of the concept of polymorphism (including even nonparametric polymorphism where f does depend on A). We already know of simple ways to accomplish this (e.g. altering the formation rules), and we want to explore the consequences of the alternatives on other aspects of the theory, such as the need for other uses of sets.

5. A Unified Reflexive Logic - A Foundation for Intelligent Systems

In our applied work on Nuprl, we make extensive use of the polymorphic programming language ML. It serves as the metalanguage to Nuprl in which users write procedures to search for proofs. The type discipline of ML is extremely useful in this capacity. ML can in principle be defined as a sublanguage of Nuprl. Considerable practical advantage would be gained if this were actually done. First, there would be a uniform way of reasoning about the theory inside the theory itself; as a consequence some of the proof building tactics could be proved to work and would not
have to be executed. Second, the syntactic structure of terms would be available for use in defining such concepts as the computational complexity of functions (see below).

Merging ML into Nuprl is part of a general process of building a partial reflexive closure of Nuprl, that is, a language which can talk about its own syntax and its own provability conditions. This is an exceedingly complex and delicate task because one ventures close to paradox. But the first steps of the process were already carried out in the course of building the system interface between Nuprl and ML. We would like to complete the process. However because it is so delicate, we have decided to study it theoretically for simpler languages first.

As a preliminary step in this investigation we have studied the process of closing a class of total computable functions with a type structure similar to Nuprl's. The results might be considered surprising. Constable [24] shows that it is possible to have a subrecursive class of total computable functions, each represented as a data object, such that an evaluation function gives the meaning of the object. Although this construction appears to violate basic results in computing theory, in fact it does not, because the type structure is so complex that no internal functions can enumerate the function spaces.

(Q8) We propose to investigate these closure results further. While the above results offer guidance in the task of building the closure of Nuprl, we need more information. For instance, we need to study the closure over proofs as well as over functions, and we need to treat types as data objects as well.

One of the most important consequences of such a study might be the methods provided for dealing with issues of computational complexity. We have been keenly aware of this issue from the beginning and have studied ways of treating functions intensionally [20,21]. We think of functions as algorithms, so two functions $f$ and $g$ in $A \rightarrow B$ need not be equal when $f(x) = g(x)$ for all $x$ in $A$. They are equal only if their "structures are the same."

In the case of Nuprl, we have not yet succeeded in describing the conditions of structural equivalence between functions as we did for our earlier type theory, V3 [20]. The mechanisms that make this possible in V3, typed combinators, complicate writing ordinary mathematics. So we are
trying a different approach. The reflexive closure result would provide a definite method for describing the structure of functions in Nuprl.

(Q9) We also propose to investigate alternative definitions of function structure and computational complexity along these new lines. The internal representation of a function will be taken to define its structure. This will be sensible because equality of functions is not extensional. The computational behavior of a function can be dependent on properties that will be revealed by syntactic analysis of notation for that function. This is, of course, not possible for all functions in classical mathematics.

References


[26] Constable, Robert L. and Mендler, N.P. Recursive definitions in type theory. (To appear Nov. 1984.)


[32] Demers, A.J. and Donahue, J. 'Type-completeness' as a language principle. 7th ACM Symp. on Principles of Programming Languages, January 1980, 234-244.


[37] Feferman, S. Constructive Theories of Functions and Classes. Logic Colloquium '78, North-Holland, Amsterdam, 1979, pp. 159-224.


[66] Plotkin, Gordon D. Notes for post-graduate course in semantics, Computer Science Department, Edinburgh, 1983.


[69] PRL Staff. Implementing mathematics with the Nuprl proof development system. Computer Science Department, Cornell University, January 1985.


