Theoretical Computer Science 121 (1993) 411–440
Elsevier

A type-theoretical alternative to ISWIM, CUCH, OWHY

Dana S. Scott
Carnegie-Mellon University, Pittsburgh, PA, USA, and RISC-Linz, Austria

Abstract

Scott, D.S., A type-theoretical alternative to ISWIM, CUCH, OWHY, Theoretical Computer Science 121 (1993) 411–440.

The paper (first written in 1969 and circulated privately) concerns the definition, axiomatization, and applications of the hereditarily monotone and continuous functionals generated from the integers and the Booleans (plus "undefined" elements). The system is formulated as a typed system of combinators (or as a typed λ-calculus) with a recursion operator (the least fixed-point operator), and its proof rules are contrasted to a certain extent with those of the untyped λ-calculus. For publication (1993), a new preface has been added, and many bibliographical references and comments in footnotes have been appended.

Preface (1993)

The main part of the text of this paper was written in England in October, 1969, mid-way through the term the author spent on leave from Princeton University visiting Professor Christopher Strachy and his *Programming Research Group* at Oxford University. The preparation of this paper for its long-delayed publication has been done while the author was on sabbatical leave from Carnegie Mellon University visiting Professor Bruno Buchberger at his *Research Institute for Symbolic Computation* at the Johannes Kepler University, Linz, Austria. The author is very much indebted not only to the universities mentioned for these various opportunities to take leave and to enjoy hospitality at the places visited, but also to Todd and Mary Wilson and Kim Wagner for typing the manuscript from a very old photocopy of a typescript, for writing and fixing the necessary TEX macros to typeset the new version as a report, and for helping assemble the bibliography.

Correspondence to: D.S. Scott, Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213-3890, USA. Email: dana.scott@cs.cmu.edu.

The author is also much indebted to the editors of this volume for the welcome suggestion that such "historical" papers might be published this way. This particular paper has, of course, an odd historical role: in it the author argues *against* the type-free calculi of Church and Curry, Kleene and Rosser, and their later uses by Böhm and Strachey. But then in November of 1969, after writing this report, the author himself saw that the method of monotone continuous functions (which grew out of traditional recursive function theory in discussing certain kinds of functionals) could be applied to posets other than just those generated from the integers (with bottom) by the very simple type constructors. In particular, it was recognized that there were posets whose continuous function spaces of self-maps could be identified with the given posets themselves. And so there actually were "traditional" models of λ-calculus that could be defined without first going through the proof theory of the formal system itself (and which could be related to many other mathematically meaningful structures and theories as well).

This revelation was both gratifying and embarrassing. After writing with tiresome sarcasm about the lack of meaning in the type-free λ-calculus introduced only formally, the author himself found an interesting kind of semantical interpretation for the "type-free" language. This total shift of gears is the reason the present paper was not published: the foundational program being advocated had apparently been completely outmoded by the discovery of the more general lattice-theoretic models. However, the *axiomatic* program laid out here had much to recommend it, and it was continued and extended in many directions by Milner, Plotkin, and many others, to whom the paper had been circulated privately – often at nth hand. Gunter in his book [9, p. 143] remarks, "The language PCF itself was introduced by Scott in what is probably the most well-known unpublished manuscript in Programming Language Theory". That exaggerates history somewhat, but the proof system proposed was in fact one of the main motivations for Milner to make automated proofs for such axiomatics (see [8, p. 153]). And this project in turn led directly to the definition, design and implementation of the programming language ML – a very important event in the history of computer languages, since ML has since prospered and taken on a role, importance and life of its own never dreamed of in the mid-1970s. Moreover, the completeness problem for the theory of the type system of this paper turned out to be far more delicate than was initially apparent (see the historical introduction to the Kahn–Plotkin paper in this volume).

On the other hand, the type-theoretical approach has not died out at all, because it has been taken over and absorbed into the applications of category theory to semantics and computation theory. The author is fond of saying that a category represents the "algebra of types", just as abstract rings give us the algebra of polynomials, originally understood to concern only integers or rationals. One can of course think only of particular type systems, but, for a full understanding, one really needs also to take into account the general theory of types, and especially *translations* or *interpretations* of one system in another. Category theory together with the notion of functor and natural transformation between functors has been proved over and

over again in the last half-century to be the appropriate way to go about these studies. The author himself does not always like or enjoy the discipline of category theory, which seems ofttimes to carry along very, very heavy machinery and odd terminology, but he long ago came to the conclusion that it is quite *unavoidable*. The extremely active current research in semantics also shows that it is an especially fruitful way to think. The book of Gunter [9] with its wide-ranging historical comments and references is adequate proof of this assertion.

The strange title of this paper ought perhaps to be explained. In 1966, Landin published an influential paper [14] which introduced a syntactical design style for programming languages, one of which he called ISWIM, standing for "If you See What I Mean". Also Böhm in 1966 published the paper [3] which named a language of combinators called CUCH, standing for "Curry–Church". There seemed to be a worrisome trend in funny acronyms starting here (of which perhaps the ultimate examplar is the well-known and very widely used editing/programming interface called GNU, recursively standing for "GNU is NOT Unix"). The author hoped to stop some proliferation by suggesting a return to the logically standard type-theoretical framework and thereby deter the creation of programming languages of doubtful foundation called (as a group) OWHY, standing for "Or What Have You." No one really understood the joke, and the effort was doomed to be of no avail. And history proved the author to be too conservative in any case.

In the body of the paper footnotes giving relevant comments and some references have been added for this publication. A brief afterthought has been added as a last section. Some comments in the original text have also been transposed to footnotes to help readability. Several editorial changes and corrections were incorporated, but the original text has essentially been preserved. The bibliography is to be found at the end of the paper.

0. Introduction (1969)

No matter how much wishful thinking we do, the theory of types is here to stay. There is *no other way* to make sense of the foundations of mathematics. Russell (with the help of Ramsey) had the right idea, and Curry and Quine are very lucky that their unmotivated formalistic systems are not inconsistent.[1] This is not to disparage formalistic work. In my view it is only through formalism that we can find a clear idea of the *scope* of mathematical knowledge. And I freely admit that one's research *may* be advanced by some purely formalistic play with symbols. My point is that formalism *without eventual interpretation* is in the end useless. Now, it may turn out that a system such as the λ-calculus will have an interpretation along standard lines (and I have spent more days than I care to remember trying to find one), but until it is produced

[1] The author still believes this statement.

I would like to argue that its purposes can just as well be fulfilled by a system involving types. Indeed, as far as *proofs* are concerned, the system with types seems to be much better.[2]

It is a pity that a system such as Zermelo–Fraenkel set theory is usually presented in a purely formal way, because the *conception* behind it is quite straightforwardly based on type theory. One has the concept of an arbitrary *subset* of a *given domain* and that the collection of *all subsets* of the given domain can form a *new* domain (of the next type!). Starting with a domain of individuals (possibly empty), this process of forming subsets is then iterated into the *transfinite*. Thus, each set has a type (or *rank*) given by the ordinal number of the stage at which it is first to be found in the iteration. One advantage of this method is that the types are built into the sets themselves and need not be made manifest in the formalism. (Computer people might say that the type checking in set theory is done at *runtime* rather than at *compile time*.) One disadvantage is that people tend to forget what is out of sight. But it *is* there, and one can make quite clear what is the type-theoretic background of set theory.[3]

For the purposes of understanding computation, however, set-theoretical formalism is not too helpful in any direct way. In the first place, too much of set theory concerns the *transfinite*, and ordinary computation has rather to do with finite processes. In the second place the axioms of set theory are meant to capture something essential of the idea of an *arbitrary* subset, while computation theory is more interested in the notion of an *algorithmically defined* subset (or function). Of course, one can define in set theory such notions as that of a general recursive function, but such definitions do not emphasize enough what is special about algorithms. Nor is it generally clear when a defined function *is* recursive. So what we want is a "restricted" system that is specially designed for algorithms.[4] What I shall present below is an independent system with its own axioms and rules; but, since I observe the canons of type theory, it can be (and indeed must be) read as a fragment of set theory so that its theorems can be recognized as *valid*. This is the main feature missing from the λ-calculus. Now I have only thought of this system in the last few days, so it may still be imperfect.[5] However, none of it is *wrong* (as will be seen from the simple character of the system), and it does seem to do in a much better way what I discussed

[2] This statement remarkably remains true! Even with the subsequent 20-year development of the theory of domains, the proof principles for recursively defined (or reflexive) domains are still being discovered. This can be well appreciated by reading the two very recent papers of Pitts [18, 19].

[3] Though set theory, and especially ZF set theory, underwent a truly vast development in the last 30 years, the common understanding of the type structure is probably not yet fully appreciated – to judge by the many arguments the author has had with category theorists.

[4] The subsequent development of intuitionistic ZF and the expansion of realizability interpretations has completely changed this position. Unfortunately, the axiomatics of "synthetic domain theory" have not been completely clarified so that a convenient foundation for computation theory and semantics can be given in set-theoretic terms (see [12, 25] and the references therein).

[5] The report was written at one sitting over a very short period and never revised.

as the "algebraic theory of procedures" in my talk on the last day of the W.G. 2.2 meeting in Essex.[6]

1. Types

The first confusion we should avoid is that between *logical* types and what we might call *data* types. The former are what we invoke to study the latter. The *theory* of data types requires the logical types (and certain notions about objects of these types) for its formalization. For example, the idea of the set of all subsets of a set is a logical notion (or *mathematical* notion, if you prefer), because neither it, nor for that matter the "general" set, can ever present itself as an object of data – in any ordinary sense of that word. However, set-theoretical notions can be quite useful when, for instance, we wish to say that the set of all data (of a certain kind) has *no* proper subset with a particular property (say, of being closed under a specific operation). I like to imagine the data at the lowest logical type being structured by certain (fixed) relations and functions (objects of a higher logical type) and the theory of these allowing reference to (variable) objects of all the higher types – as in the example mentioned above.[7]

Obviously, for a good theory we want to be able to sort out data into different categories (types?), *and* (this is where the confusion begins) we want to study many *derived* types. As a simple example we could think of *persons* as forming a basic type and *organizations* as forming a derived type. By an *organization* I suppose we all understand a sort of *tree* of persons. It has a *head* (a person) and some *branches* which give the "chain of command". Well, there is no confusion here; where the problem lies is in this: the objects of the various logical types are richly structured. As we all know, the theory of trees (of persons, say) can easily be *simulated* or *modelled* by objects of higher logical types (oh, a "mathematical" tree is an ordered pair of a set together with a relation that partially orders the set – in a special way). Now is this a good thing? Sometimes yes, sometimes no. If we think of organizations as occurring in "nature", then we *do not* want to identify them with the mathematical models. It is like the distinction between abstract and concrete syntax: there is not always a good reason to consider an *expression* to be a *string* (*sequence* mathematically?) of *symbols*. Why? Because the string-oriented methods of breaking an "expression" into "parts" might not lead to the correct or useful notion of *part*. Similarly with *organizations*.

To put the point in another way: it seems best to allow ourselves the freedom of keeping our data types as *primitives*. Computers have taught us this. Think of the numbers. There is no unique *representation* of numbers, and we do not want to choose a particular one. All we really want to say is that they form *one* data type and that they

[6] The meeting was in the summer of 1969, and the method discussed there was from the long-unpublished de Bakker–Scott paper [6].

[7] And consider the subsequent development of "concrete domains" by many authors after the Kahn–Plotkin report.

have a certain structure. (I suppose they can be *added*, say, and can be tested in pairs for *equality* or *order*.) Indeed we may require several different types of numbers. In our theory about these numbers we very much want the results to be "machine independent". That is, the theorems ought to be valid for *all* representations – satisfying certain explicit structural conditions. (Equals added to equals remain equal?)

This attitude was somewhat obscured by Russell's approach to mathematics: he wanted to *reduce* mathematics to logic. How? By *defining* number. (The *number two is* the set of all two-element sets – be careful of type distinctions here!) The program is not quite successful. Why? Because one must still *postulate* the existence of an infinite set (a data type?) in order to have "enough" numbers. Set theory gets around this point by going to *transfinite types*! It is not exactly cheating, but it is not really satisfactory either. For *our* purposes it is much better to give a *theory* (theories) of number rather than definitions of number. And the same applies to the other data types. What we are going to do is to take what we need from Russell's *logic* without taking over his *philosophy* of mathematics.

Actually, it is fairer to say that we are stealing from Church's logic rather than Russell's, because my *notation* is closer to that of Church.[8] The reason is that for algorithms it is more natural to consider *functions* rather than *sets*. We can reduce the notion of function to that of set, but it is *not* convenient to do so. A much better plan is to treat sets (and relations) as special functions (truth-valued) as Church does.

To start with we have two "logical" types represented by the Greek letters ι and o in Church's notation. The first, ι, stands for the type of all *individuals*. I consider it to be the *largest data type*; all other data types are to be "included" in it – hence it is clearly a logical notion. (The exact way of treating the other data types will be discussed below.) The second, o, is the type of the *truth values* – a very logical notion. These are the two logical types of the lowest order. The higher types are represented as follows: If α and β are types, then so is $(\alpha \to \beta)$. What $(\alpha \to \beta)$ represents is the type of *functions from* objects of type α *to* those of type β. Again a logical construct.

To be more precise, we must distinguish between type *symbols* and the (sets of) *objects* of the corresponding type. Type symbols are strings of "ι", "o", "$\to$" "(", and ")". From my informal remarks in the last paragraph, anyone can write down the "grammar" for this (context-free) "language", and I shall not bother to do so. Aside from this collection of type symbols, we will also have a rich language of *expressions*. Each expression will have a (unique) type, and I shall write $X\colon \alpha$ to mean that the expression X is of type α. Again we must be careful to distinguish the expressions from what objects they denote. More on this later.

In the first place as expressions we shall allow, for each type α, an infinite list of *variables* denoted as follows by lower-case letters:

$$a_\alpha, b_\alpha, \ldots, x_\alpha, y_\alpha, z_\alpha, a'_\alpha, \ldots, z'_\alpha, a''_\alpha, \ldots$$

[8] This goes back to the well-known paper [4].

(I *suppose* I should do all this in abstract syntax because no one really cares what my variables look like. But I don't have time to fuss.) Those subscripted variables are in the *object language* of expressions. In my *metalanguage* I use *un*subscripted x, y, z, etc. to range over variables of any type. The unsubscripted capital letters range over compound expressions.

The compounds are made from the variables and *constants* by a certain rule. The constants are divided into two main classes: the *logical* and *nonlogical* constants. I choose not to discuss the latter at the moment – just remember to save room for them. And remember in general that any expression has its type – which must be *given* in the case of constants (and variables). The logical constants are infinite in number (so are the types); I list them along with their types (α, β, γ are arbitrary type symbols):

$$\Omega_\iota : \iota, \qquad \Omega_o : o, \qquad \top : o, \qquad \bot : o,$$

$$\supset_\alpha : (o \to (\alpha \to (\alpha \to \alpha))),$$

$$\mathbf{K}_{\alpha\beta} : (\alpha \to (\beta \to \alpha)),$$

$$\mathbf{S}_{\alpha\beta\gamma} : ((\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))),$$

$$\mathbf{Y}_\alpha : ((\alpha \to \alpha) \to \alpha).$$

One can probably guess what they mean, but I leave that discussion for the next section.[9]

As for the other expressions, suppose $X : \alpha$ and $F : (\alpha \to \beta)$; then we have

$$F(X) : \beta.$$

That is to say the standard function-value notation is the only way in which compounds can be made. Note that the type of X must *fit* the type of F for $F(X)$ to be well formed. Note, too, how complicated the types of our constants are. In particular, it is possible to form an expression of *any* type using only constants and no variables.

By a *formula*, we understand either an *atomic* formula of the form

$$X \leqslant Y,$$

where X and Y are expressions, or a *list*

$$\Phi_0, \Phi_1, \Phi_2, \ldots, \Phi_{n-1}$$

(possibly empty!) of atomic formulae Φ_i. We identify the one-termed list with the atomic formula.[10] (Since my natural syntax is string-like, note that in view of the fact that the concatenation of expressions is *never* an expression, we can *write* a list of atomic formulae *without* commas. Let us not worry about such small points, however.) If Ψ and Φ are two lists, then (in the metalanguage!)

$$\Phi \subseteq \Psi$$

[9] Kleene and many other logicians had used typed combinators, but the languages with the typed fixed-point combinator had not been considered all that much in 1969 except by Platek in his thesis [20].

[10] The lists of atomic formulae are really *conjunctions*, as is explained later.

means that every atomic formula in Φ also occurs in Ψ. (The symbol "$\leqslant$" is a symbol of the *object* language; similarly for "$\vdash$".)

By an *assertion*, we understand a string of the form

$$\Phi \vdash \Psi,$$

where Φ and Ψ are lists of atomic formulae. Intuitively lists of formulae are just *conjunctions* and $\vdash$ gives an *implication* between conjunctions – but this will all be clear when we find out in the next section the meanings of all our symbols. After that we will discuss *axioms* and *rules of inference* for generating the (or a good part of the) *valid* assertions in an "algebraic" way.

If Φ is a list and F is an expression of the same type as a variable x, then

$$\Phi[F/x]$$

denotes the result of *substituting* F for x throughout Φ.

We need a few abbreviations. If X and Y are expressions of the same type, then

$$X = Y$$

stands for the list

$$X \leqslant Y,\ Y \leqslant X.$$

If F is an expression, and if $F(X)(Y)(Z)$ is well formed, then we abbreviate this as

$$F(X, Y, Z).$$

Similarly, for more than three terms. (Note this abbreviation is a convention in the metalanguage and is not – at the moment – "sugaring" in the object language.) Also, we have only Ω_t and Ω_o at the lowest type. For higher types we define $\Omega_{(\alpha \to \beta)}$ to be the expression:

$$\mathbf{K}_{\beta\alpha}(\Omega_\beta) : (\alpha \to \beta).$$

The notation

$$\supset_\alpha (B)(X)(Y)$$

is not common. We usually write

$$(B \to X, Y),$$

but we also need $\supset_\alpha$ for general purposes.[11]

[11] Subsequently, the author's notation changed because it was confusing (he felt) to use either $\leqslant$ or $\subseteq$ for the information ordering within a domain. He thus adopted the "square" notation of $\sqsubseteq$ from lattice theory. This brought along $\sqcap$ and $\sqcup$ and took $\bot$ for the bottom element instead of Ω. It still seems to the author a better notation, but too many people prefer to write the easier $\leqslant$. However, in a system that might involve integers, sets (say as elements of power domains), and the information ordering, it seems cleaner to have different symbols for different notions.

2. Interpretation

The classical way of viewing the theory of types is to assign to each type α a *domain* D_α, where D_ι is a given domain of individuals, D_o is the domain of *two* truth values (denoted by $\top$ and $\bot$ for *true* and *false*), and each $D_{(\alpha\to\beta)}$ is the domain of *all* functions from D_α with values in D_β. This point of view is *not* convenient for our purposes. The reason is simple: classical type theory supposes *total* (everywhere defined) functions, while algorithms in general produce *partial* functions. We do not wish to reject a program if the function defined is partial – because as everyone knows it is not possible to predict which programs will "loop" and which will define total functions.

The solution to this problem of total versus partial functions is to make a "mathematical model" for the theory of partial functions using ordinary total functions. The idea is not at all original to the author (he has taken it from more "standard" versions of recursive function theory – in particular, from the thesis of Platek [20]). Other authors in recursive function theory discussed monotone and "hereditarily consistent" functionals, notably Kleene, Rogers, Putnam and Davis, but there may be a few points of originality. In one direction, the axiomatization of the next section – especially the induction rule – is original as far as the author knows.

What we do is to adjoin a "fictitious" element Ω_ι to the domain D_ι and an element Ω_o to D_o. We call Ω_ι the "undefined" individual and Ω_o the "undefined" truth value. However, we need to distinguish the new elements from the old. To do this we create a relation $\leqslant$ on D_ι and $\leqslant$ on D_o (same symbol – different relations) meaning, roughly, "is less or equally defined as". Thus, a reasonable assumption is that

$$\Omega_\iota \leqslant x$$

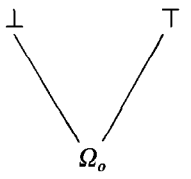
for all $x \in D_\iota$; but that

$$x \leqslant y$$

implies

$$x = y$$

for $x, y \in D_\iota$ with $x \neq \Omega_\iota$.[12] We make this assumption about $\leqslant$ on D_o, but for the moment not about D_ι. Thus, a "picture" of D_o could be



[12] [from the original text] I am sorry that I must use the same symbols in the metalanguage as some of those in the object language. It is a sad fact that there are just *too few* symbols. If I were more careful with quotes I would say that the *symbol* "$\leqslant$" is being given the interpretation of denoting the *relation* $\leqslant$. Hopefully, the reader can take the required care in his own thought.

with the slanting lines indicating $\leqslant$. We are therefore involved with a *three-valued* "*logic*" with the new value Ω_o "in between" $\top$ and $\bot$ but placed "a little lower down". (Look at the picture!) We shall see presently how good a three-valued logic we have.[13] (Similar pictures could be given for the "reasonable" view of D_ι.)

The upshot of all this is that D_ι and D_o are *partially ordered* by $\leqslant$. What about $D_{(\iota\to\iota)}$ for example? Well, what *is* $D_{(\iota\to\iota)}$? In the classical version we took *all* functions; not so here. We want only the *monotonic* functions (as we shall find later – *at most* these). By *monotonic* we understand a function $f: D_\iota \to D_\iota$ (this is the usual mathematical notation) where $x \leqslant y$ implies $f(x) \leqslant f(y)$. In words this means: the more you *define* an argument, the more you define its *value* under a "computable" function. The same idea can be applied to a *variable* f in the combination $f(x)$. That is to say, if we *define*

$$f \leqslant g$$

to mean

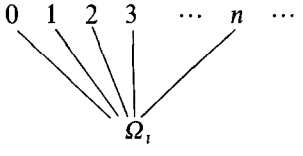
$$f(x) \leqslant g(x)$$

for all $x \in D_\iota$, then the combination $f(x)$ is monotonic in *both* f and x, and the set $D_{(\iota\to\iota)}$ is partially ordered. Note that there is a natural "smallest" element among the elements of $D_{(\iota\to\iota)}$, namely the function $\Omega_{(\iota\to\iota)}$, such that

$$\Omega_{(\iota\to\iota)}(x) = \Omega_\iota$$

for all $x \in D_\iota$. Note too that once $D_{(\iota\to\iota)}$ has its own $\leqslant$, we can then define $D_{(\iota\to\iota)\to\iota}$ and $D_{((\iota\to\iota)\to(\iota\to\iota))}$, etc., by the *same* plan of taking only the monotonic functions. Indeed we can now define D_α for every type symbol α.

The plan just described of using monotonic functions is almost correct, but not quite. An example will make it all clear. Let $D_\iota = \{0, 1, 2, \ldots\} \cup \{\Omega_\iota\}$, the domain of ordinary integers *plus* Ω_ι. In pictures:



Now the monotonic functions f are almost like ordinary functions except we allow

$$f(n) = \Omega_\iota$$

for certain arguments n if we so desire. If we read this equation as: f is *undefined* at n, then we agree that f is *defined* only for a *subset* of the integers. Conversely, if f_0 is defined on a subset S of the integers, with integer values, then we can extend f_0 to

[13] Kleene introduced such a three-valued logic in [13], and it has been discussed by many, many authors.

a *monotonic* function f by defining

$$f(n)=\begin{cases} f_0(n) & \text{if } n\in S, \\ \Omega_\iota & \text{otherwise.} \end{cases}$$

This requires $f(\Omega_\iota)=\Omega_\iota$. Such functions we call *strict*. Not all functions need be strict; we allow the *constant* functions, say

$$g(x)=0, \quad \text{all } x\in D_\iota,$$

where $g(\Omega_\iota)\neq\Omega_\iota$. The need for the distinction between strict and nonstrict functions will become clear later.[14]

We have not yet seen the difficulty with monotonic functions, however. The $D_{(\iota\to\iota)}$ just described is fine. It is only when we come to $D_{((\iota\to\iota)\to\iota)}$ that there is a question. An element $h\in D_{((\iota\to\iota)\to\iota)}$ is a *functional*. The equation

$$h(f)=n,$$

where $f\in D_{(\iota\to\iota)}$ and $n\in D_\iota$ means that h "computes" the value n from the argument f. But f is a *function*; an infinite object (e.g. it may have infinitely many function values). What does it mean to "compute" with an "infinite" argument? In this case it means most simply that $h(f)$ is determined by asking of f (maybe by some algorithmic process) finitely many *questions* – that is to say, *values*:

$$f(m_0), f(m_1), \ldots, f(m_{k-1}).$$

That is, as a functional h is *continuous* in some sense. Now fortunately we do not need at this point to involve ourselves in topology to any great extent. We can use our partial ordering $\leqslant$ to pin down what we need most.

Note that even though D_ι is a very trivial partially ordered set, $D_{(\iota\to\iota)}$ is *not*. The partial ordering on $D_{(\iota\to\iota)}$ is quite complex (at least as bad as the Boolean algebra of all sets of integers – if not worse). In particular, we can form in $D_{(\iota\to\iota)}$ many infinite chains of functions:

$$f_0<f_1<f_2<\cdots<f_n<\cdots$$

For example, let $f(x)=x$ for all $x\in D_\iota$ and define

$$f_n(x)=\begin{cases} x & \text{for } x=0,1,2,\ldots,n, \\ \Omega_\iota & \text{otherwise.} \end{cases}$$

Then the f_n form a chain in the above sense and each $f_n<f$. Actually it is easy to see that f is the *least upper bound* of the f_n in the sense of the partial ordering $\leqslant$ on $D_{(\iota\to\iota)}$.

[14] But of course it did not become clear until very much later that there are many different categories of domains, and that sometimes it is necessary to work only with strict functions and to have functors that "lift" continuous functions to strict functions.

We can write

$$f=\bigvee_{n=0}^{\infty} f_n.$$

In fact, $D_{(\iota\to\iota)}$ has the property that *every* chain has a *lub*. What does this have to do with continuity? Well, it is easy to show in an intuitive way that

$$h\left(\bigvee_{n=0}^{\infty} f_n\right)=\bigvee_{n=0}^{\infty} h(f_n)$$

holds for *every* chain of f_n if h is continuous. What we are going to do is to take the above equation as the *abstract* definition of continuity. (Note that the lub operation trivially works in the ground domain D_ι.)

To be a bit more precise: suppose D_α and D_β have been defined and partially ordered by $\leqslant$ relations in such a way that lubs of chains always exist. *Then* for $D_{(\alpha\to\beta)}$ we allow only those functions $h: D_\alpha\to D_\beta$ that are at the same time monotonic *and* continuous. This space $D_{(\alpha\to\beta)}$ has a natural partial ordering, and (as should be proved by the reader) lubs of chains always exist in $D_{(\alpha\to\beta)}$. (Hint:

$$\left(\bigvee_{n=0}^{\infty} h_n\right)(x)=\bigvee_{n=0}^{\infty} h_n(x).)$$

Note that with this convention the application operation $f(x)$ is always monotonic and continuous in each of f and x.

We have now defined the domains D_α for all α, which means that all we have done so far is to specify the *ranges of our variables* x_α. (Note that D_ι can be *any* "abstract" set with a $\leqslant$ satisfying the *lub* condition. One should not always restrict attention to the integers.) The next step is to specify the meanings of the constants. Now we obviously want the *symbols* "Ω_ι", "Ω_o", "$\top$", "$\bot$" to stand for the *values* Ω_ι, Ω_o, $\top$, $\bot$. The other constants are functions and must be so defined. Thus,

$$\supset_\alpha(p, x, y)=\begin{cases} x & \text{if } p=\top,\\ y & \text{if } p=\bot,\\ \Omega_\alpha & \text{if } p=\Omega_o,\end{cases}$$

for all $p\in D_o$, $x, y\in D_\alpha$, where $\Omega_\alpha\in D_\alpha$ has already been defined. Next

$$\mathbf{K}_{\alpha\beta}(x, y)=x$$

for all $x\in D_\alpha$ and $y\in D_\beta$. Then

$$\mathbf{S}_{\alpha\beta\gamma}(f, g, x)=f(x, g(x))$$

for all $f\in D_{(\alpha\to(\beta\to\gamma))}$, $g\in D_{(\alpha\to\beta)}$, and $x\in D_\alpha$. The definition of $\mathbf{Y}$ is more difficult.

Already we should be forced to *prove* something in order to see that the above definitions "make sense". In particular, we need to know that the functions $\supset_\alpha, \mathbf{K}_{\alpha\beta}, \mathbf{S}_{\alpha\beta\gamma}$ do belong to the correct domains. This means that we must show (1)

that they are well-defined functions, and (2) that they are monotonic and continuous. Actually, this all follows from a "composition theorem" for monotonic and continuous functions; we give a representative example of an instance of this theorem.

Theorem 2.1. *Suppose functions g, h, k are monotonic and continuous, and suppose f is defined by the equation*

$$f(x, y) = g(h(x, y), k(x, y)).$$

Then, f is monotonic and continuous (in each of its variables).

The monotonic part of the theorem is obvious. To prove the continuity we must calculate

$$\begin{aligned} f\left(\bigvee_{n=0}^{\infty} x_n, y\right) &= g\left(h\left(\bigvee_{n=0}^{\infty} x_n, y\right), k\left(\bigvee_{n=0}^{\infty} x_n, y\right)\right) \\ &= g\left(h\left(\bigvee_{l=0}^{\infty} x_l, y\right), k\left(\bigvee_{m=0}^{\infty} x_m, y\right)\right) \\ &= g\left(\bigvee_{l=0}^{\infty} h(x_l, y), \bigvee_{m=0}^{\infty} k(x_m, y)\right) \\ &= \bigvee_{l=0}^{\infty} \bigvee_{m=0}^{\infty} g(h(x_l, y), k(x_m, y)). \end{aligned}$$

Now we note that in a partially ordered set a lub of lubs is just the lub over the double index set (in this case over the pairs (l, m).) Furthermore, to calculate a lub it is sufficient to find the lub of a *cofinal* subset. In our case the elements $g(h(x_n, y), k(x_n, y))$ are cofinal, because we assume the x_n form a *chain* ($x_n \leqslant x_{n+1}$), and if $n = \max(l, m)$, then

$$g(h(x_l, y), k(x_m, y)) \leqslant g(h(x_n, y), k(x_n, y))$$

by monotonicity. Hence,

$$\begin{aligned} f\left(\bigvee_{n=0}^{\infty} x_n, y\right) &= \bigvee_{n=0}^{\infty} g(h(x_n, y), k(x_n, y)) \\ &= \bigvee_{n=0}^{\infty} f(x_n, y). \end{aligned}$$

Thus, $f(x, y)$ is continuous in x.

It is clear that the proof just given applies for any number of variables. Note too that the variables are independent and may belong to *different* domains. Thus, in our example we could have had: $x \in D_\alpha$, $y \in D_\beta$, $h \in D_{(\alpha \to (\beta \to \gamma))}$, $k \in D_{(\alpha \to (\beta \to \delta))}$, and $g \in D_{(\gamma \to (\delta \to \varepsilon))}$ producing $f \in D_{(\alpha \to (\beta \to \varepsilon))}$. The reader should note in particular that our functions are all "Curried" (as the saying goes), so that $f(x, y)$ really means $f(x)(y)$. Thus, $f(x) \in D_{(\beta \to \varepsilon)}$, and one should take care to remark that with x *fixed*, $f(x)(y)$ is

continuous in y; whence $f(x) \in D_{(\beta \to \varepsilon)}$ *is* correct. Then, with x *variable*, $f(x)$ is continuous in x, which gives the result.

It is also clear, as remarked before, that *application* $f(x)$ is monotonic and continuous in f and in x. Therefore, any *combination* defines a good function. This remark, for example, justifies the definition of $\mathbf{S}_{\alpha\beta\gamma}$:

$$\mathbf{S}_{\alpha\beta\gamma}(f, g, x) = f(x)(g(x)),$$

in Curried form, and shows why

$$\mathbf{S}_{\alpha\beta\gamma} \in ((\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))).$$

We have followed Curry (and others) in giving names only to "combinators" $\mathbf{K}_{\alpha\beta}$ and $\mathbf{S}_{\alpha\beta\gamma}$, because all others can be *defined* in terms of these. For example:

$$\begin{aligned} \mathbf{I}_\alpha(x) &= x \quad \text{(the outright definition)} \\ &= \mathbf{K}_{\alpha(\alpha\to\alpha)}(x)(\mathbf{K}_{\alpha\alpha}(x)) \\ &= \mathbf{S}_{\beta\gamma\alpha}(\mathbf{K}_{\alpha(\alpha\to\alpha)})(\mathbf{K}_{\alpha\alpha})(x). \end{aligned}$$

Hence,

$$\mathbf{I}_\alpha = \mathbf{S}_{\beta\gamma\alpha}(\mathbf{K}_{\alpha(\alpha\to\alpha)})(\mathbf{K}_{\alpha\alpha}), \quad \text{(the indirect definition)},$$

where $\beta = (\alpha \to ((\alpha \to \alpha) \to \alpha))$ and $\gamma = (\alpha \to (\alpha \to \alpha))$. Another example:

$$\begin{aligned} \mathbf{C}(x)(f) &= f(x) \\ &= \mathbf{I}(f)(\mathbf{K}(x)(f)) \\ &= \mathbf{S}(\mathbf{I})(\mathbf{K}(x))(f) \\ &= \mathbf{K}(\mathbf{S}(\mathbf{I}))(x)(\mathbf{K}(x))(f) \\ &= \mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{I})))(\mathbf{K})(x)(f), \end{aligned}$$

whence

$$\mathbf{C} = \mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{I})))(\mathbf{K}),$$

where we have left it to the reader to fill in the type subscripts. Obviously, this "economy" has only "theoretical" importance. The proper thing to do is to introduce a λ-operator and write

$$\mathbf{I}_\alpha = \lambda x_\alpha [x_\alpha],$$

$$\mathbf{C}_{\alpha\beta} = \lambda f_{(\alpha\to\beta)} \lambda x_\alpha [f_{(\alpha\to\beta)}(x_\alpha)].$$

We shall not do so at the moment because we do not want to formulate all the rules about free and bound variables. Our axioms are simpler if we keep to the "algebraic" theory that has only free variables.

So $\mathbf{K}_{\alpha\beta}$ and $\mathbf{S}_{\alpha\beta\gamma}$ are the so-called *combinators*, but what is $\supset_\alpha$? Answer: $\supset_\alpha$ is the McCarthy operator for forming the *conditional expression.* Thus, instead of

$\supset_\alpha(p)(x)(y)$ we may write (informally) the more usual $(p \to x, y)$. Its properties are well known. What might not be so well known is its use at higher types. Thus, consider the definition

$$\mathbf{P}_\alpha(x, y)(p) = \supset_\alpha(p)(x, y),$$

where $x, y \in D_\alpha$. What is $\mathbf{P}_\alpha(x, y) \in D_{(o \to \alpha)}$? Answer: it is an *ordered pair*. Indeed let us write informally

$$\langle x, y \rangle = \mathbf{P}_\alpha(x, y);$$

then we see

$$\langle x, y \rangle(\top) = x,$$

$$\langle x, y \rangle(\bot) = y,$$

which shows why $\langle x, y \rangle$ is a pair. Note, however, that this is a *logical construction* and should *not* be confused with a *data type* for pairs of individuals. (By the way, one should check that $\supset_\alpha$ really is monotonic and continuous.)

Finally, we must interpret $\mathbf{Y}_\alpha$. We use the notation for Curry's "paradoxical" combinator, *but* we cannot use Curry's definition. Why? Simple: my view is that the type-free theory *makes no sense whatsoever*. This remark applies to the theory *as a whole* and does not prevent me from gaining inspiration from *parts* of the theory.[15] The inspiration we need about $\mathbf{Y}_\alpha$ is the so-called *fixed-point property*: if $f \in D_{(\alpha \to \alpha)}$, then we want

$$\mathbf{Y}_\alpha(f) = f(\mathbf{Y}_\alpha(f)).$$

Two questions: how do we know f has a fixed point; and if it does, which one shall we choose for $\mathbf{Y}_\alpha(f)$? Answer: we are lucky in being able to choose the *least* fixed point – "least" in the sense of $\leqslant$ on $D_{(\alpha \to \alpha)}$. It is constructed by iteration as follows: For $x \in D_\alpha$, let

$$f^n(x) = \underbrace{f(f(\ldots f(x)\ldots))}_{n \text{ times}},$$

where $f^0(x) = x$. Then note

$$\Omega_\alpha \leqslant f(\Omega_\alpha);$$

[15] [from the original text] The same could have been said about infinitesimals in the calculus – except that nowadays they have a reasonable interpretation [in nonstandard analysis]. [comment 1993] Of course, once models for the type-free theory had been defined, the "paradoxical" definition could be used. Surprisingly, it turned out that these definitions, although giving fixed points, did not always have simple properties. See the discussion and references in Barendregt [1].

and so

$$f(\Omega_\alpha) \leqslant f^2(\Omega_\alpha);$$

and by induction

$$f^n(\Omega_\alpha) \leqslant f^{n+1}(\Omega_\alpha),$$

because f is *monotonic*. We can thus define

$$\mathbf{Y}_\alpha(f) = \bigvee_{n=0}^{\infty} f^n(\Omega_\alpha),$$

and then calculate by continuity:

$$f(\mathbf{Y}_\alpha(f)) = \bigvee_{n=0}^{\infty} f^{n+1}(\Omega_\alpha) = \mathbf{Y}_\alpha(f).$$

This proves that f has a fixed point. Now suppose a is any other. Then

$$\Omega_\alpha \leqslant a,$$

and so

$$f(\Omega_\alpha) \leqslant f(a) = a,$$

whence

$$f^n(\Omega_\alpha) \leqslant a.$$

Thus,

$$\mathbf{Y}_\alpha(f) = \bigvee_{n=0}^{\infty} f^n(\Omega_\alpha) \leqslant a.$$

This proves that $\mathbf{Y}_\alpha(f)$ is the *least* fixed point.

We must still prove that $\mathbf{Y}_\alpha(f)$ is monotonic and continuous in f.[16] The monotonicity is very easy, but the continuity requires some computation. Suppose $f_n \leqslant f_{n+1}$ is a chain of functions in $D_{(\alpha \to \alpha)}$. Then

$$\left(\bigvee_{n=0}^{\infty} f_n \right)(x) = \bigvee_{n=0}^{\infty} f_n(x).$$

Let

$$a = \bigvee_{n=0}^{\infty} \mathbf{Y}_\alpha(f_n);$$

[16] [from the original text] Sorry about that! When one says what one means, one must demonstrate the correctness of one's definitions.

we must show

$$a = \mathbf{Y}_\alpha\left(\bigvee_{n=0}^{\infty} f_n\right).$$

It is obvious from monotonicity that

$$a \leqslant \mathbf{Y}_\alpha\left(\bigvee_{n=0}^{\infty} f_n\right).$$

To complete the proof, compute:

$$\begin{aligned}
\left(\bigvee_{n=0}^{\infty} f_n\right)(a) &= \bigvee_{n=0}^{\infty} f_n(a) \\
&= \bigvee_{n=0}^{\infty} f_n\left(\bigvee_{m=0}^{\infty} \mathbf{Y}_\alpha(f_m)\right) \\
&= \bigvee_{n=0}^{\infty}\bigvee_{m=0}^{\infty} f_n(\mathbf{Y}_\alpha(f_m)) \\
&= \bigvee_{n=0}^{\infty}\bigvee_{m=0}^{\infty} f_{\max(n,m)}(\mathbf{Y}_\alpha(f_{\max(n,m)})) \\
&= \bigvee_{n=0}^{\infty} f_n(\mathbf{Y}_\alpha(f_n)) \\
&= \bigvee_{n=0}^{\infty} \mathbf{Y}_\alpha(f_n) \\
&= a.
\end{aligned}$$

Hence, a is a fixed point, and so $\mathbf{Y}(\bigvee_{n=0}^{\infty} f_n) \leqslant a$. Thus, we have proved

$$\mathbf{Y}_\alpha \in D_{((\alpha\to\alpha)\to\alpha)}.$$

Having now interpreted all of our constants, we can define the important notion of *validity*. Suppose X and Y are two expressions of the same type. In general, they contain *variables*; hence they do not denote, as they stand, anything in particular. But, if we *assign values* in the appropriate D_α to each of the variables, then all of the symbols in the expressions become meaningful and X and Y have values. Thus, *under the assignment to the variables*, the atomic formula $X \leqslant Y$ is either *true* or *false*. Now consider an assertion $\Phi \vdash \Psi$. It is said to be *valid* if under *every* assignment of values to variables that makes *all* the atomic formulae of the list Φ true, *all* the atomic formulae of Ψ are true also. That is, Φ "implies" Ψ with the variables being universally quantified.

Strictly speaking we have only defined validity with respect to the *given* domain D_ι. We are more interested at the moment in those assertions that are *universally valid* for *all* choices of D_ι. We shall see many examples of valid assertions in the next section.

Notice, however, how different our method is compared to the λ-calculus. In the latter theory, validity of equations (interconvertibility) is defined in a purely formal manner. Here we have *defined* validity "semantically" and must *discover* the formal properties of this notion. Of course in our metalanguage we are taking "on faith" the existence of the various higher-type continuous functions in the D_α that we have been defining. But this is *normal* mathematics. The validity of conversions in λ-calculus has *no such mathematical foundation.*[17]

3. Axiomatization

In the first place there are some very general properties of $\vdash$ that would be the same for any similar theory. We give "axioms" (quite self-evidently valid assertions) and "rules of inference" (simple deduction methods that clearly preserve validity).

(INCLUSION) $\quad \Phi \vdash \Psi, \quad$ where $\Psi \subseteq \Phi$.

(CONJUNCTION) $\quad \dfrac{\Phi \vdash \Psi \quad \Phi' \vdash \Psi'}{\Phi, \Phi' \vdash \Psi, \Psi'}$.

(CUT) (OR SYLLOGISM) $\quad \dfrac{\Phi \vdash \Psi \quad \Psi \vdash \Theta}{\Phi \vdash \Theta}$.

(SUBSTITUTION) $\quad \dfrac{\Phi \vdash \Psi}{\Phi[X/x] \vdash \Psi[X/x]}$,

where X and x are of the same type.

Clearly, the rule of substitution can be generalized to simultaneous substitution for several variables. However, this more general rule can be proved as a derived rule of inference.

Next the relation $\leqslant$ enjoys several useful properties:

(REFLEXIVITY) $\quad \vdash x \leqslant x$.

(TRANSITIVITY) $\quad x \leqslant y,\ y \leqslant z \vdash x \leqslant z$.

(MONOTONICITY) $\quad x \leqslant y,\ f \leqslant g \vdash f(x) \leqslant g(y)$.

(EXTENSIONALITY) $\quad \dfrac{\Phi \vdash f(x) \leqslant g(x)}{\Phi \vdash f \leqslant g}$, where x is *not* in Φ.

Note especially that we have stated these principles *without type subscripts.* This is a very convenient trick available in the metalanguage. The point is that, say, $\vdash x \leqslant x$ is valid for variables x of *all* types. Similarly for the axiom of transitivity with the understanding that x, y, z are all of the *same* type – otherwise the formulae would not

[17] [from the original text] At least to date, but I despair of ever seeing an adequate justification.

all be well formed. In the axiom of monotonicity and the rule of extensionality, x and y must be of the same type, say α, and f and g must be of a type $(\alpha \to \beta)$ for the assertions to make sense.

It is interesting to ask why the rule of extensionality is correct. Well, suppose $\Phi \vdash f(x) \leqslant g(x)$ is valid. Since x is a variable being assumed not to occur in any formula of Φ, it is a "free" variable in $f(x) \leqslant g(x)$. "Free" in the strongest sense that we are free to give it any value. Now consider $\Phi \vdash f \leqslant g$. Give values to the variables in Φ (and to f and g) to make all the formulae of Φ true. By assumption $f(x) \leqslant g(x)$ is true for all values of x (of the correct type!). *Hence*, by *definition* of $\leqslant$ for functions, $f \leqslant g$ is true. Thus, we have shown that from the validity of $\Phi \vdash f(x) \leqslant g(x)$, the validity of $\Phi \vdash f \leqslant g$ follows.

By the way, remember that $X = Y$ is short for $X \leqslant Y$, $Y \leqslant X$. Note that $=$ is indeed an equality relation because we can now *prove*, as theorems from the axioms and rules we already have, that

$$\vdash X = X,$$

$$X = Y,\ Y = Z \vdash X = Z,$$

$$X = Y \vdash Y = X,$$

$$\Phi[X/x],\ X = Y \vdash \Phi[Y/x].$$

The last one relies heavily on monotonicity.

Inasmuch as we have assumed no "non-logical" constants, there is in general very little to say about individuals of type ι except for the "undefined" individual Ω_ι:

(MINIMALITY$_\iota$) $\vdash \Omega_\iota \leqslant x_\iota$.

Later we shall be able to prove this about all the types and all the Ω_α – which are defined and not primitive. If, and only in the case of individuals (and truth values), we wanted to have the "reasonable" view of Ω_ι and the other individuals, we *might* want to assume:[18]

(DISCRETENESS$_\iota$) $\Phi,\ \Phi[\Omega_\iota/x_\iota],\ y_\iota \leqslant x_\iota \vdash \Phi[y_\iota/x_\iota]$.

This principle means that the only $y_\iota \leqslant x_\iota$ are Ω_ι and x_ι itself. This is definitely *not* correct for higher types. In the case of truth values we shall not have to assume it, but shall prove it from the properties of the conditional expression.

The first, most trivial, property of truth values is[19]

(MINIMALITY$_o$) $\vdash \Omega_o \leqslant x_o$.

[18] The principle was stated incorrectly in the original paper. The revision was suggested by Todd Wilson.

[19] Note here, as for type ι, we can use the *particular* variable x_o for emphasis of the type.

Then we have for $\supset_\alpha$:[20]

(CONDITIONALITY) $\vdash \supset_\alpha(\top, x, y)=x,$

$\vdash \supset_\alpha(\bot, x, y)=y,$

$\vdash \supset_\alpha(\Omega_o, x, y)=\Omega_\alpha,$ where $x, y:\alpha.$

Finally, we must express the idea that $\top$, $\bot$ are the only allowed truth values:

(EXHAUSTION)

$$\frac{\Phi[\top/x_o]\vdash\Psi[\top/x_o] \quad \Phi[\bot/x_o]\vdash\Psi[\bot/x_o] \quad \Phi[\Omega_o/x_o]\vdash\Psi[\Omega_o/x_o]}{\Phi\vdash\Psi}.$$

This principle allows us to argue by cases. For example, we can prove all the well-known laws of the conditional expression by simple (but lengthy!!) arguments by cases. Thus, they are not needed as axioms, but they *must* be proved as "lemmas" very early on.

The axioms for the combinators come directly from their definitions:

(**K**-CONVERSION) $\vdash \mathbf{K}_{\alpha\beta}(x, y)=x,$ where $x:\alpha, y:\beta.$

(**S**-CONVERSION) $\vdash \mathbf{S}_{\alpha\beta\gamma}(f, g, x)=f(x, g(x)),$

where $f:(\alpha\to(\beta\to\gamma))$, $g:(\alpha\to\beta)$, and $x:\alpha$.

Sometimes people give other complicated equations for the combinators. This seems to be a desire to avoid using the rule of extensionality (say, in the work of Rosser). This would seem to be important only in the situation where one wanted to do away with variables altogether. We are not using *bound* variables here, but there seems to be no reason not to have free variables. Hence, all those equations may be proved by extensionality – just as we prove propositional truths by using the rule of exhaustion of cases.

Finally, we must isolate the basic facts about the fixed-point operator:

(STATIONARINESS) $\vdash f(\mathbf{Y}_\alpha(f))\leqslant\mathbf{Y}_\alpha(f),$ where $f:(\alpha\to\alpha).$

This is slightly weaker than stating an equation – but the other direction can be proved by using the very important rule:

$$\text{(INDUCTION)}\ \frac{\Phi\vdash\Psi[\Omega_\alpha/x] \quad \Phi, \Psi\vdash\Psi[f(x)/x]}{\Phi\vdash\Psi[\mathbf{Y}_\alpha(f)/x]},$$

where the type restrictions are these: $x:\alpha$ and $f:(\alpha\to\alpha)$, *and* where x must *not* occur in Φ. (We need x as a "free" variable as in the rule of extensionality.)

[20] [from the original text] Note that we could drop all the type subscripts, and the reader could very safely put them back in. It would be interesting to formalize rules for the metalanguage for doing this automatically. [comment 1993] This was done together with a type of polymorphism by Milner in ML to excellent effect.

As a first example of the use of induction we prove the fixed-point *equation*: thus,

$$\Omega_\alpha \leqslant f(\Omega_\alpha)$$

is obviously provable. Further,

$$x \leqslant f(x) \vdash f(x) \leqslant f(f(x))$$

is provable by monotonicity. Therefore, by induction

$$\mathbf{Y}_\alpha(f) \leqslant f(\mathbf{Y}_\alpha(f)).$$

As a second example, we show that the fixed point is *minimal*: thus,

$$f(a) \leqslant a \vdash \Omega_\alpha \leqslant a$$

is obvious. Further,

$$f(a) \leqslant a, x \leqslant a \vdash f(x) \leqslant a$$

is easy to establish by monotonicity and transitivity. Therefore, by induction

$$f(a) \leqslant a \vdash \mathbf{Y}_\alpha(f) \leqslant a.$$

Although he does not see how to prove it at the moment, the author is reasonably certain that these two instances of induction, though useful, are not the whole story. That is, it is not enough only to assume them as axioms; the full rule is really needed. For example, let us define (as a suitable combination of **K**'s and **S**'s) the composition operator **B** where for $f:(\beta \to \gamma)$, $g:(\alpha \to \beta)$ and $x:\alpha$ we have as provable:

$$\vdash \mathbf{B}_{\alpha\beta\gamma}(f, g, x) = f(g(x)).$$

Now a mildly interesting theorem about fixed points is the following:

$$\mathbf{B}(f, g) = \mathbf{B}(g, f), \mathbf{B}(f, h) = \mathbf{B}(h, f), g(\Omega) \leqslant h(\Omega) \vdash g(\mathbf{Y}(f)) \leqslant h(\mathbf{Y}(f)),$$

where type subscripts must be written in so that it all makes sense. The easy proof by induction is clear. The author does not see how to prove this in any other way.[21]

Having seen how useful induction is, one may well ask: why is it valid? We shall argue its correctness by a method that brings out some other useful points of the theory. Note in particular this property of the *pairs* we defined in the last section:

$$x \leqslant x', y \leqslant y' \dashv\vdash \mathbf{P}(x, y) \leqslant \mathbf{P}(x', y'),$$

where in general $\Phi \dashv\vdash \Psi$ is short for the *two* assertions $\Phi \vdash \Psi$ and $\Psi \vdash \Phi$. Hence, we see that any list Φ of formulae is such that there exists a (long) "paired" formula $X \leqslant Y$, where

$$\Phi \dashv\vdash X \leqslant Y$$

[21] In fact, there have been many subsequent studies of induction principles. See for example [18, 19], but also compare the proofs in [5].

is provable. Next we recall the property of combinators whereby, if X is any expression and x is any variable, then there exists an expression F such that F *does not* contain x and

$$\vdash F(x) = X$$

is provable.[22] By these remarks it becomes clear that without loss of generality the Ψ of the induction rule can be taken to be of the form $g(x) \leqslant h(x)$, where g and h are variables and their "definitions" $g = G$, $h = H$ can be pushed into the Φ. Thus, the rule has the simpler looking form

$$\frac{\Phi \vdash g(\Omega) \leqslant h(\Omega) \quad \Phi, g(x) \leqslant h(x) \vdash g(f(x)) \leqslant h(f(x))}{\Phi \vdash g(\mathbf{Y}(f)) \leqslant h(\mathbf{Y}(f))},$$

where still x may not occur in Φ.

The validity of the conclusion of the rule is now almost self-evident: assume values are given to the variables (except for x) so that Φ is true. Then by the first assumption

$$g(\Omega) \leqslant h(\Omega)$$

is true. Then by the second assumption where x is valued as Ω, we have

$$g(f(\Omega)) \leqslant h(f(\Omega)).$$

Applying the second (inductive) assumption repeatedly, we find

$$g(f^n(\Omega)) \leqslant h(f^n(\Omega))$$

to be true. Now passing to the limit and using the continuity of g and h, we have

$$g(\mathbf{Y}(f)) \leqslant h(\mathbf{Y}(f))$$

true as desired.

It is this rule of induction that I consider the main advantage of my system. As far as I know there is nothing like it for the λ-calculus.[23] Of course, the rule *is* very much like McCarthy's principle of *recursion–induction* (see [15]). In fact, I view my rule as being a more general version of McCarthy's principle, which, after we gain more experience with it, will be even easier to apply. Note that the principle as I state it is "abstract" and does not require any "knowledge" of the *integers* in its formulation. Of course, I used ordinary integer-indexed iteration to *justify* the rule – but that argument was in the metalanguage. The "formal" connection with integer induction and recursion will be discussed in the next section.

[22] [from the original text] The F can effectively be found from the X and can be called $\lambda x[X]$ – this is a short-hand in the *metalanguage*.

[23] [from the original text] Morris in his thesis [17] proves a minimal fixed-point property for the λ-calculus. The proof, however, is *very complicated* – like the Church–Rosser theorem – and it is not quite clear exactly what it gives one for proofs about λ-expressions.

4. Completeness

We have presented a language (Section 1); we have interpreted it semantically (Section 2); we have written down many self-evident "axioms" and validity-preserving rules (Section 3); now it is time to ask: How much progress have we made? *If* we could only show that the axioms were *complete* in the sense of allowing us to derive *every* valid assertion by formal proofs based on the axioms and using the rules, *then* we could be very satisfied by our progress. Such completeness is impossible, however. The argument is a standard one of showing that the class (of Gödel numbers) of valid assertions is *not recursively enumerable.* (The class of theorems *is* recursively enumerable.) Standard as the argument is, it may be instructive to see how it can be expressed in our language. We will, by the way, assume the discreteness rule throughout.

What we shall do is to introduce four *nonlogical constants* $0:\iota$, $Z:(\iota\to o)$, $+1:(\iota\to\iota)$, and $-1:(\iota\to\iota)$ of the types indicated. We shall then write down five simple equations involving these constants, in a list called $\mathscr{A}$, such that any interpretation making $\mathscr{A}$ true must be *isomorphic* to the system of *integer arithmetic.* That is to say the system where 0 is zero, Z is the monadic predicate of being equal to zero, and where $+1$ and -1 are the successor and predecessor functions. It will thus be "difficult" to enumerate valid assertions of the form[24]

$$\mathscr{A}\vdash\Phi.$$

Now that we are moving a little away from "theory" and a little closer to "practice", we must begin to "soften" our notation. Thus, the $(\to,)$-notation is better than the $\supset$-notation. We shall also use the λ-notation, but keep in mind that $\lambda x[X]$ is a metalinguistic abbreviation for an expression F not involving the variable x such that $\vdash F(x)=X$ is provable – which expression is not important.[25] We shall also write $x+1$ (with *no* parentheses) instead of $+1(x)$ and similarly for $x-1$. We could even introduce the many excellent features of format used in Landin's ISWIM – but remember: *our* variables are forever typed![26]

The first four equations of $\mathscr{A}$ are very elementary:

$$Z(\Omega_\iota)=\Omega_o,$$

$$Z(0)=\top,$$

$$0-1=\Omega_\iota,$$

$$\lambda x_\iota[x_\iota+1-1]=\lambda x_\iota[x_\iota],$$

$$\mathbf{Y}_{(\iota\to\iota)}(\lambda f_{(\iota\to\iota)}[\lambda x_\iota[(Z(x_\iota)\to 0, f_{(\iota\to\iota)}(x_\iota-1)+1)]])=\lambda x_\iota[x_\iota].$$

[24] Because, as is argued below, there is no recursive enumeration of all true equations between primitive recursive functions.

[25] [from the original text] I *could* make it precise, if it were necessary.

[26] [from the original text] Remember too that we have no "program points" nor assignment statements – just a "pure" functional calculus.

The last is harder to appreciate – especially with the subscripts – but all will be made clear. (As a first step, drop the subscripts.) The first equation means that the truth value for Ω's being zero is "undefined". The second equation means the zero *is* zero; hence, $0 \neq \Omega$ because $\top \neq \Omega$. We have not used $\neq$ as part of our "official" notation before and we will *not* make it official now. What it means to say that $\top \neq \Omega$ is that the assertion

$$\top = \Omega \vdash \Phi,$$

where Φ is *arbitrary*, is provable (and hence valid). In other words, a *false* equation implies anything. The third equation means that the predecessor of zero is "undefined". The fourth equation means that

$$x+1-1=x$$

is true for all values of $x \in D_\iota$ – and everyone knows what *that* means.

The last equation involves a recursive definition. Indeed, let $N:(\iota \to \iota)$ stand for the left-hand side of the equation. Then by "definition",

$$\vdash N(x) = (Z(x) \to 0,\ N(x-1)+1).$$

What the fifth *equation* is telling us is that

$$N(x) = x$$

must hold for all x. That is telling us a lot! For one thing we will know that

$$(Z(x) \to 0,\ x-1+1) = x$$

holds for all x. From our other equations we know that $x+1 \neq 0$ for all $x \neq \Omega$ (because $0-1=\Omega \neq x = x+1-1$). We also know that $0 \neq 0+1 \neq 0+1+1$, etc. (Hence $\Omega+1=\Omega$ by discreteness, but that is not too important.) What is new from the last equation is that if $x \neq 0$ and $x \neq \Omega$, then $Z(x) = \bot$ and $x-1+1=x$. (Hence $x-1 \neq \Omega$ for $x \neq 0$, $x \neq \Omega$.) In particular, we see $Z(x) = \top$ iff $x=0$. In other words, zero, successor, and predecessor have all the elementary relations they should. But even more than this is contained in the fifth equation, and that more is *mathematical induction.*

To be more precise we shall derive this rule:

$$\frac{\mathscr{A}, \Phi \vdash \Psi[\Omega/x] \quad \mathscr{A}, \Phi \vdash \Psi[0/x] \quad \mathscr{A}, \Phi, \Psi \vdash \Psi[x+1/x]}{\mathscr{A}, \Phi \vdash \Psi},$$

where x is not in Φ. To make the derivation even more transparent, we may assume without loss of generality that Ψ has the form

$$g(x) \leqslant h(x),$$

where $x:\iota$ and $g, h:(\iota \to \alpha)$ for a suitable α. The problem then is to show that the hypotheses imply

$$\mathscr{A}, \Phi \ \vdash\ g \leqslant h.$$

In view of what we have assumed in $\mathscr{A}$, it will be sufficient to prove

$$\mathscr{A}, \Phi \vdash \lambda x[g(N(x))] \leqslant \lambda x[h(N(x))].$$

Now look at the *definition* of N. Note first that we can prove (by assumption)

$$\mathscr{A}, \Phi \vdash \lambda x[g(\Omega(x))] \leqslant \lambda x[h(\Omega(x))].$$

The desired conclusion will follow by our general rule of induction if only we can prove

$$\mathscr{A}, \Phi, \lambda x[g(f(x))] \leqslant \lambda x[h(f(x))]$$

$$\lambda x[g((Z(x) \to 0, f(x-1)+1))] \leqslant \lambda x[h((Z(x) \to 0, f(x-1)+1))].$$

We can now drop the λx's in the conclusion, and we can argue by cases $Z(x) = \Omega$, $Z(x) = \top$, $Z(x) = \bot$. Each case is easily done in view of our three assumptions. Thus, the proof is complete.

That shows the *deductive* power of $\mathscr{A}$. The *semantic* import should now be clear: any interpretation making $\mathscr{A}$ true must be isomorphic to the integers. Now using the $0, Z, +1, -1$ notation it is clear that we can write down **Y**-definitions of any primitive recursive function (of any number of arguments). Consider two such definitions F and G. Clearly, now

$$\mathscr{A} \vdash F = G$$

is valid (in all interpretations) if the functions defined by F and G in the *standard* integers are equal. It is well known that one cannot enumerate the pairs of equal primitive recursive functions; hence the class of valid assertions of our calculus cannot be recursively enumerable.

Is this incompleteness result a cause for despair? I think not. The system is really very strong. Much stronger than what logicians call *primitive recursive arithmetic* because of the use of the higher types. It is more in the line of a higher-type theory of recursive functionals proposed by Gödel and studied by him and others. Our system is more usable that Gödel's (I won't say "stronger" because I do not know at the moment) in view of our use of partial functions. In particular, we can define not only primitive recursive but also general recursive functions – on the basis of $\mathscr{A}$.

It is enough to see how to define Kleene's μ-operator. The trick is a common one: suppose f is a given function and we want to find the least integer x such that $f(x) = 0$ and $f(y) \neq \Omega$ for smaller y. We define by recursion the function g such that

$$g(x) = (Z(f(x)) \to x, g(x+1)),$$

and then the desired integer is $g(0)$. More formally

$$\mu x[f(x) = 0] = \mathbf{Y}(\lambda g[\lambda x[(Z(f(x)) \to x, g(x+1))]])(0).$$

It would be interesting to have a clearer idea of what the various definable partial functions of *higher type* really are – maybe the recursive functionals people already know.[27] One particularly important question is this: suppose we fix the standard arithmetic interpretation of $0, Z, +1, -1$. Suppose we consider a *definition* of a function $f:(\iota \to \iota)$. We have just shown that every general recursive function may be defined, but conversely? Is every definable function general recursive? The answer may be yes, but the higher types make it complicated to see.[28] A possible outline of an argument might be as follows: use an abstract machine to calculate f by symbolic manipulation of its definition. In fact, we might even be able to use (almost directly) Landin's λ-calculus machine. Our axioms for $\supset, \mathbf{K}, \mathbf{S}, \mathbf{Y}$ (and for $0, Z, +1, -1$) tell us exactly how to make conversions. If the expression for f is F, we just try reducing

$$F(\underbrace{0+1+1 \ldots +1}_{n \text{ times}})$$

until we "finally" get

$$\underbrace{0+1+1 \cdots +1}_{f(n) \text{ times}}.$$

Clearly, if the reduction rules always work to produce an answer when $f(n)$ is defined, then f is indeed general recursive because the rules are effective.

Note that the rules can never give a *wrong* answer. The big question is whether, when we *expect* an answer semantically, we can always *find* it formally. If we could prove that, it would be a kind of completeness of our calculus – completeness for numerical equations – and would be the most we could hope for. It would also be – theoretically at least – a very important result. One wonders what the situation would be on domains other than the well-worn integers. It strikes the author as significant that this question cannot even be asked of the λ-calculus because the λ-calculus has no meaning. One has to take the reduction rules "on faith" and has no "standards" to which they must match.

A third question about completeness concerns the power of expression of our language: can we define everything we want? The answer is not all that clear because it is difficult to see just what we do want. Certainly we want what we have at

[27] They did. See, for example, the study of Hyland [11] and the brief remarks in [7].

[28] [from the original text] The proof might be contained in the thesis of Platek [20], but that is a very difficult work to read, to say the least. [comment 1993] The document is still difficult for the author to read, but the proof required is more simply based on the idea of "effectively given domains". This means that since the domains used here are all algebraic cpos with effectively enumerable bases of finite elements, the general recursiveness of all definable functions on the integers can be proved semantically by structural induction on the size of the definition. See [22, 10, 23] for various expositions.

the moment, but consider the simple truth function $\vee : (o \to (o \to o))$ given the table:

$\vee$	$\top$	Ω	$\bot$
$\top$	$\top$	$\top$	$\top$
Ω	$\top$	Ω	Ω
$\bot$	$\top$	Ω	$\bot$

Axiomatically, $\vee$ is determined by the following equations:

$$\vdash \vee(p, q) = \vee(q, p),$$

$$\vdash \vee(p, \top) = \top,$$

$$\vdash \vee(p, \bot) = p.$$

Then all the other facts about the table follow by monotonicity. (Note that this $\vee$ is indeed monotonic.) Now the question is: do we want it? Is there any reason to exclude *any* monotonic truth function.

By the way, it is an interesting exercise to prove that *all* monotonic truth functions of any number of arguments can be defined in terms of $\vee$ and $\neg$, where $\neg : (o \to o)$ is defined in the usual way:

$$\neg(p) = (p \to \bot, \top).$$

The main trouble with $\vee$ is that it is *symmetric*. Thus, consider the function definition:

$$h(x) = ((Z(f(x)) \vee Z(g(x)) \to x, x+1),$$

where I inadvertently wrote $(p \vee q)$ for $\vee(p, q)$. To evaluate h in terms of the given functions f and g, we must calculate f and g *in parallel*. Thus, for a particular x, we start to calculate both $f(x)$ and $g(x)$. If one of them gives the answer 0, we stop and are sure that $h(x) = x$. If both give answers other than 0, we know $h(x) = x + 1$. Otherwise, $h(x) = \Omega$. Well, that is a kind of algorithm, but it has a flavor different from the usual bread-and-butter calculations that proceed one step at a time. Do we enjoy this new flavor enough to call it computable? Some people would say yes, but I wonder. It seems harmless, but maybe we should think about it more.[29]

[29] [from the original text] Platek also discussed *and rejected* the idea in his thesis in his study of Kleene's work. [comment 1993] Plotkin settled all the questions asked here when the symmetric $\vee$ is allowed in [21]. As the discussion in the preface to the Kahn–Plotkin paper shows, extensive research on "sequential functions" has still not come to complete satisfactory conclusions.

5. Conclusions

It seems to me that the idea of a monotonic and continuous function is a very natural one for anyone thinking about computability.[30] What I have tried here is to give a logical *calculus* (or even: *algebra*) for the notion using type theory. The point is that the types are natural – the higher-type functionals are useful – and they have the advantage of possessing a semantical interpretation. It is important to remember that I consider the higher-type functionals as *logical* notions to be kept separated from the *data types* (more on this below). I think I have given enough detail here to demonstrate that this "algebra of computation" works very smoothly and naturally – though I admit that fully formal proofs would be very lengthy. (Should we think of automating any proof procedure?[31]) It is not very surprising that there is a nice algebra since all we really need are the conditional expression and the possibility of explicit definition (**S** and **K**) and of recursive definition (**Y**). (I ask again: do we want $\vee$? any others??) There is the question of computational completeness mentioned in Section 4, however, and this should be given more thought.

Now what about other data types? My present view is that all the data should be kept in type ι. In Section 4, I showed how one might structure the objects of type ι as the integers with the aid of $0, Z, +1, -1$. Numbers are only *one* type of data. We could imagine D_ι as being divided into *many disjoint* parts, each part with its own structure and with axioms for the structure given in the same style as in $\mathscr{A}$. For example, one part might be LISTS (regarded as data rather than as logical constants), and we would need structure such as *NIL*, *cons*, *car*, *cdr*. The axioms would be very similar to those in $\mathscr{A}$. The advantage of the "axiomatic" approach over the "definitional" method would be in the freedom we would allow ourselves in representing the data, say, in a machine. We would only have to check that the data structure as implemented satisfied the axioms. (Question: what to do about "overflow", that is, the finite character of most representations?) Now Hoare (to mention only one person) has already started writing down axioms for data structures, and it would seem that the present theory offers a rigorous framework for this activity. The idea requires much more study, however.

Finally, we must agree that the study of λ-calculus cannot replace the study of programming languages. It is true that the logical notation allows us to express many

[30] [from the original text] The recursive function people have been considering monotonic functions for a long time, and Bekić recently came across the idea again in his study of automata theory [2]. Park and Florentin also used the notion in discussions of the Floyd–Manna proof theory for programs. It is mentioned, for example, in the Eilenberg–Wright automata theory via categories. Indeed, it is a "folk" notion. The author became interested in it when trying to find the best induction principle for de Bakker's algebraic approach to program equivalence – and he is indebted to de Bakker for many discussions on trying to develop a useful algebra. (See the later work in [5].) Of course, he knew about Platek's work, but Platek does not "seem" to discuss continuity.

[31] Milner and coworkers did in his LCF!

computations and that the system could be given the look of a programming language, but we have *not* built into the theory logical notions corresponding to the full glory of the *assignment statement* and to the idea of jumps and *goto*'s. Landin has tried to do this with ISWIM, but the personal view of the author is that the result is not quite successful. Landin *does* have a clean, regular, and powerful language, but in a certain sense it is just another language: evaluation still must be done on a machine. Now maybe Landin's evaluations are easier to follow than some other methods, but somehow I do not feel that he has given a "logical" explication of the notion of assignment.[32] My current idea is that we should take up Strachey's plan of giving an "axiomatic" discussion of the store and its transformations using the theory of L and R values. The locations (addresses) and the stores would be treated *as new data types.* Why? Because they have machine representations.[33] Well, these ideas are still in a state of flux, but the author hopes that the distinction between logical and data types can help us sort out the features of a rather murky landscape.

6. Afterthought (1993)

The historical overview presented here by the revisions to the paper has been very brief and very, very selective. A long and tiring literature search would be required to write a really satisfactory discussion of all major developments and influences. There are many names that should have been mentioned. Twenty-five years is not such a long time, but the enormous number of conference proceedings and journal literature produced in theoretical areas of computer science and programming-language semantics over that period make a bibliographer's task quite daunting. Also, each month brings several new papers from many very active centers of research. So, I doubt whether a good history can ever be written in my lifetime. But the author is not trying to excuse himself for not doing better!

Rereading this paper after such an interval was a surprise for me, however. Despite some doubtful rhetoric, the paper still makes reasonable sense, has clear definitions, and suggests possible trends or problems that were indeed taken up with positive results. Moreover, not all the questions raised have been settled. The paper, therefore, served a function, and it is still worthwhile to stop to consider why.

[32] [from original text] I hope Landin will reply to this criticism, because it seems to me to be a basic "philosophical" point that should be cleared up. [comment 1993] The author rather doubts that he did. However, work by Milner, Plotkin, and many, many others have refined, contrasted and related "operational semantics" to "denotational semantics".

[33] Indeed, the ideas were much developed, as shown by the books of Stoy [24] and Milne-Strachey [16]. Nevertheless, a really satisfactory mathematical theory of *store* and *assignment* is still missing.

References

[1] H.P. Barendregt, *The Lambda Calculus – Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, Vol. 103 (North-Holland, Amsterdam, Revised edition, 1984).

[2] H. Bekić, Definable operations in general algebras, and the theory of automata and flowcharts, Tech. Report, IBM Laboratory, Vienna, 1969.

[3] C. Böhm, The CUCH as a formal and description language, in: T.B. Steel, Jr., ed., *Formal Language Description Languages for Computer Programming, IFIP Working Conference on Formal Language Description Languages*, Vienna 1964 (North-Holland, Amsterdam, 1966) 179–197.

[4] A. Church, A formulation of the simple theory of types, *J. Symbolic Logic* **5** (1940) 56–68.

[5] J.W. de Bakker, *Mathematical Theory of Program Correctness* (Prentice-Hall, Englewood Cliffs, NJ, 1980).

[6] J.W. de Bakker and D. S. Scott, A theory of programs, IBM Seminar, Vienna, August 1969, in: J.W. Klop et al., eds., *J. W. de Bakker, 25 Jaar Semantiek: Liber Amicorum* (CWI, Amsterdam, 1989) 1–30.

[7] R.O. Gandy and J.M.E. Hyland, Computable and recursively countable functions of higher type, in: R.O. Gandy and J.M.E. Hyland, eds., *Logic Colloquium 76* (North-Holland, Amsterdam, New York, Oxford, 1977) 407–438.

[8] M.J. Gordon, A.J. Milner and C.P. Wadsworth, *Edinburgh LCF*, Lecture Notes in Computer Science, Vol. 78 (Springer, Berlin, Heidelberg, New York, 1979).

[9] C.A. Gunter, *Semantics of Programming Languages – Structures and Techniques* (MIT Press, Cambridge, MA, 1992).

[10] C.A. Gunter and D.S. Scott, Semantic domains, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science* (Elsevier, Amsterdam, 1990) 633–674.

[11] J.M.E. Hyland, Recursion theory on the countable functionals, Ph.D. Thesis, Univ. of Oxford, 1975.

[12] J.M.E. Hyland, First steps in synthetic domain theory, in: A. Carboni, M.C. Pedicchio and G. Rosolini, eds., *Category Theory, Proc. Como 1990*, Lecture Notes in Mathematics, Vol. 1488 (Springer, Berlin, 1990).

[13] S.C. Kleene, *Introduction to Metamathematics*, The University Series in Higher Mathematics (van Nostrand, New York, 1952).

[14] P.J. Landin, The next 700 programming languages, *Comm. ACM* **9** (1966) 157–164.

[15] J. McCarthy, A basis for a mathematical theory of computation, in: P. Braffort and D. Hirschberg, eds., *Computer Programming and Formal Systems* (North-Holland, Amsterdam, 1963) 33–70.

[16] R. Milne and C. Strachey, *A Theory of Programming Language Semantics* (Chapman and Hall, London, 1976).

[17] J.H. Morris, Lambda calculus models of programming languages, Ph.D. Thesis, MIT, Cambridge, MA, 1968.

[18] A.M. Pitts, A co-induction principle for recursively defined domains, *Theoret. Comput. Sci.* 1994, to appear; available as University of Cambridge Computer Laboratory Tech. Report No. 252, April 1992.

[19] A.M. Pitts, Relational properties of recursively defined domains, in: *Proc. 8th Ann. IEEE Symp. of Logics in Computer Science*, Montreal 1993 (1993).

[20] R.A. Platek, Foundations of recursion theory, Ph.D. Thesis, Stanford Univ., 1966.

[21] G.D. Plotkin, LCF considered as a programming language, *Theoret. Comput. Sci.* **5** (1977) 223–257.

[22] D.S. Scott, Lectures on a mathematical theory of computation, in: M. Broy and G. Schmidt, eds., *Theoretical Foundations of Programming Methodology* (Reidel, Dordrecht, 1982) 145–292.

[23] M.B. Smyth, Topology, in: S. Abramsky, D.M. Gabbay and T.S.E. Maibaum, eds., *Handbook of Logic in Computer Science*, Vol. 1 (Clarendon Press, Oxford, 1992) 641–761.

[24] J.E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory* (MIT Press, Cambridge, MA, 1977).

[25] P. Taylor, The fixed point property in synthetic domain theory, in: *Proc. 6th Ann. IEEE Symp. of Logics in Computer Science*, Amsterdam 1991 (1991) 152–161.