Comparing the Church and Turing Approaches: Two Prophetical Messages

Boris A. Trakhtenbrot

1. Introduction

The search for a precise mathematical characterization of what "algorithm" and "computable function" should mean resulted half a century ago in the discovery of three well-known equivalent approaches. Their chronological order is as follows: λ-definability (Church-Kleene, 1932–34), general recursiveness (Gödel-Herbrand, 1934) and Turing machines (1936).

The type-free λ-calculus was conceived by A. Church as a foundation for logic and mathematics, but this aim failed. In spite of this failure Church realized that a consistent part of this calculus is a paradigm for computation in the same way as predicate calculus is a paradigm for deduction. In 1934 he proclaimed his famous Thesis, which identifies the intuitive notion of computable function with the formal notion of λ-definable function, but there still was some lack of consensus about this Thesis. We learn from Davis *1982* that it was only after Turing's work that Gödel accepted Church's Thesis which had then become the Church-Turing Thesis. This is the way the miracle occurred: the essence of a process that can be carried out by purely mechanical means was understood and incarnated in precise mathematical definitions.

In retrospect it is not hard to understand why, unlike the previous approaches, only Turing's succeeded to convince once and for all that the genuine formulation had been achieved. The point is that Turing worked with the machine concept. Relying on this concept, he was able to give a direct analysis of computing processes and to provide clear arguments that all possible algorithms for computing functions can be embodied in such machines.

In modern jargon we can characterize Turing's approach as computer- (or hardware-) oriented; hence, also the expectation of significant practical consequences. Indeed, a Turing machine is like an actual computer except that it is error-free and has access to unlimited external memory in the form of a (potentially) infinite tape. Turing's theorem about the existence of a universal machine of this kind was a prophetical message which anticipated the era of universal digital computers; it exhibits in a rudimentary form the main ingredients of such computers. Turing's prophecy is in no essential way affected by the development of modern computing techniques which rely on more sophisticated technology and design principles. It is known that Turing himself went into computing at the National Physical Laboratory in 1945–48 and from 1948 on at the Computing Machine Laboratory in Manchester (England).

Let us also recall that, independently of Turing, E. Post also elaborated a similar analysis of processes which are performable in a purely mechanical way. However, Post's formulation is in terms of "combinatorial systems" and does not explicitly use the attractive machine paradigm.

Due to the logical interchangeability of hardware and software, one can develop programming languages based on Turing machines; indeed in the early period of computing some people developed such languages. This is an awkward programming style, but it should not be a hindrance in those situations in which one has to prove the existence of computable functions with specific properties because such proofs can be produced without having to explicitly write down the programs.

Now, returning to the earlier Church-Kleene and Gödel-Herbrand approaches, the idea occurs to characterize them (in contrast to the Turing-Post approach) as programming or software-oriented. With respect to Gödel-Herbrand recursiveness such a characterization is quite clear. Not only does recursiveness come closer to traditional mathematics, but as a matter of fact some versions of recursion and recursive schemes occur in most programming languages. Less evident is the relevance of λ-definability and in general of λ-calculus for programming languages. At first sight λ-definitions look very unusual and far from traditional mathematics and modern programming. S.C. Kleene (*1979*) remembers the rather chilly reception of audiences around 1933–35 to disquisitions on λ-definability. That is why after general recursiveness had appeared he had chosen to put his work into this format, which is more familiar to mathematicians:

> In retrospect, I now feel it was too bad I did not keep active in λ-definability as well. So I am glad that interest in λ-definability has revived, as illustrated by Dana Scott's 1963 communication. (Kleene *1979*).

Paradoxically, whatever the first impressions of λ-definitions may be, the objective truth is that the λ-calculus does implicitly incorporate some of the most important features of modern high-level programming languages. In this sense Church's approach is not just a companion to Turing's, but it also contains its own prophetical message, namely about future programming languages. Subsequent theoretical research and development of programming confirmed this message.

It took about ten years after the mathematical birth of Turing machines for the real computers to appear. The road from the λ-calculus to the study of existing programming languages and the design of new ones was considerably longer. In many respects this is astonishing, and the following are some preliminary suggestions as to why it is so.

First of all, the syntax of the λ-calculus is very simple. Therefore it is not immediately apparent how it can contain—even in an implicit and rudimentary form—features of high level programming languages, which usually rely on a very elaborate syntax.

In addition, there are the intricate semantical problems of the type-free λ-calculus. In order to produce λ-definitions for all the "computable" functions, Church was forced to put aside the semantically easier typed λ-calculus. In this way self-application became inevitable and hence also the nightmare of the set-theoretical paradoxes.

Thirdly, there is the dilemma between functional languages (and that is really what lambda-calculus may be pretending to be) and the languages with imperative features (like Fortran, Algol, Pascal, etc.) which actually dominated from the very beginning of the development of high-level programming languages.

Finally, let us mention one more restrictive factor: the λ-calculus is intrinsically sequential, whereas parallelism features are highly desirable in a modern language. (Sequentiality was also a primary restriction of Turing machines).

All these circumstances suggest that the realization of Church's vision and the rooting of the λ-calculus into the theory and practice of programming might not be free of evasiveness and obscurity. Before appropriate research was done by prominent computer scientists and logicians, it was impossibly to guess the full impact of Church's approach on programming.

Of course, by now these ideas and facts are well known to people who are engaged in this specific area. Hopefully, putting some of these ideas and facts in a certain order may be useful for a broader audience.

That is the main goal of this essay which is organized as follows: After having sketched in Section 2 the syntax of the λ-calculus, in Sections 3 and 4 the focus is on the languages LISP and ISWIM, which absorbed and promoted ideas of the λ-calculus. Landin's ISWIM was especially instrumental for standards of forthcoming functional programming languages such as Edinburgh ML and others; it paved the way to a consensus about what in Section 4 is called the Church-Landin Thesis.

Section 5 is about denotational semantics for λ-calculus and through it for programming languages in general.

The main goal of Sections 6 and 7 is to put into the right perspective the relevance of the λ-calculus for languages which allow such features (like assignments or parallelism) that to all appearances are alien to the spirit of the λ-calculus.

The style of exposition remains eclectic throughout; more or less rigorous definitions and claims alternate with informal speculations. I do not claim to have painted a complete picture. On the other hand numerous quotations from various sources have been included in order to keep the historical flavor and to reflect personal views of investigators in the area.

2. Lambda Calculus

λ-calculus amounts to λ-notations plus rules of manipulating them. The λ-notation introduced by Church is very natural and efficient; it is really a pity that it is not in general use. In daily life one has sometimes to guess whether an expression like x^y denotes a function or, alternatively, a number. Even when limited to functions one may still be confused about what function is intended. λ-notations avoid such ambiguity through the *abstraction* operation, that is a binding mechanism which explicitly points out that:

1. $\lambda x.x^y$ denotes the function, which for each x returns the value x^y;
2. $\lambda y.x^y$ denotes the exponential function.

Hence, the respective derivative functions are $\lambda x.yx^{y-1}$ and $\lambda y.x^y lny$, whereas x^y denotes a number and it does not make any sense to ask about its derivative.

Naturally, one expects that $\lambda x.x^y$ should have the same meaning as $\lambda z.z.^y$, i.e., renaming of the bounded (dummy) variable x is allowed (unlike the renaming of the free variable y) so long as there is no "collision" with the free variable.

$\lambda x.E$ represents a function with formal parameter x and body E. According to more common programming language syntax, one would prefer to assign a name to this function—say f—via a declaration like

define f: function (x);
return E end.

The other fundamental notation is for *application*. Namely, (E_1, E_2)—a slight deviation from the more familiar $E_1(E_2)$—denotes the application of the function (operator) E_1 to the argument (operand) E_2.

Formally, the expressions (terms) of the (pure) λ-calculus are introduced by induction; at the same time one defines the set Free(E) of free variable occurrences in E and bindings in E.

(0) A variable x is a term; x is its only free occurrence.

(I) *Application.* If E, F are λ-terms, so is $(E\,F)$ and the free occurrence and the bindings are those in E and in F.

(II) *Abstraction.* If E is a λ-term and x a variable, then $\lambda x.E$ is a term. Free $(\lambda x.E)$ = Free (E) − {free occurrences of x in E}. The bindings in $(\lambda x.E)$ contain all those of E and in addition the free occurrences of x in E are bound by the abstractor λx.

To make λ-terms more readable, save brackets as suggested by the examples: Instead of $((E_1E_2)E_3)$ write $E_1E_2E_3$ (applications associate to the left). Instead of $\lambda x_1.(\lambda x_2.(\lambda x_3.E))$ write $\lambda x_1x_2x_3.E$ (abstractors associate to the right).

Note that only single-argument (monadic) functions are formalized in this way. This is not an essential restriction because there is a straightforward method (currying) which reduces the use of a function f of several arguments to the use of a related monadic function $\tilde{f}$. For example, suppose we wish the binary function f to apply to x and y and to produce x^y. Then the monadic $\tilde{f}$, applied to x alone, produces the monadic exponential function $\lambda y.x^y$, whose value is just x^y when applied to y. In his *1984* paper J.B. Rosser comments in connection with this:

> This is the way computers function. A program in a computer is a function of a single argument. People who have not considered the matter carefully may think, when they write a subroutine to add two numbers, that they have produced a program that is a function of two arguments. But what happens when the program begins to run, to produce the sum $A+B$? First A is brought from memory. Suppose that at that instant the computer is completely halted. What remains in the computer is a program to be applied to any B that might be forthcoming, to produce the sum of the given A and the forthcoming B. It is a function of one argument, depending on the given A, to be applied to any B, to produce the sum $A + B$.

λ-notations are consistent with the use of higher-order functions which allow as arguments and/or return as values other functions. As a matter of fact, this underlies the idea of "currying". Using more programming jargon, one might say that in the λ-calculus programs themselves may be dealt with as data. Such a flexibility is a direct consequence from the lack of any type constraints. The price one has to pay for this is that self-applicable functions are allowed as well. For example, in the expression

$$\phi = def\ \lambda x.f(xx) \tag{1}$$

x is applied to itself.

Many programming languages share this feature of the type-free λ-calculus and allow procedures which can take themselves as arguments. But it is well known that self-application leads to contradictions, as is evident from the following definition:

$$P(f) = def\ if\ f(f) \neq 0\ then\ 0\ else\ 1. \tag{2}$$

According to (2) $P(f) \neq f(f)$ for all functions f, hence applying P to itself one gets $P(P) \neq P(P)$.

Clearly, this points to serious semantical problems we have to confront when trying to interpret λ-terms as definitions of functions in the set-theoretical sense (mappings from sets to sets). We shall return to this topic in Section 5 but meanwhile let us recall the operational semantics of the λ-calculus. It relies on three operations which intuitively are expected to preserve the meaning of terms:

1. *α-reduction* viz, renaming of bounded variables, as explained above. This corresponds to the *static scope* discipline in programming terminology.
2. *β-reduction* viz evaluation by substitution (or *call by name*):

$$(\lambda x.M)N \text{ reduces to } M[x := N]$$

in which $M[x := N]$ means the result of replacing each free occurrence of x in M by N (after appropriate renaming of bound variables in order to avoid collision of free and bound variables).

3. *β-expansion*—the operation inverse to β-reduction.

A red B means that A may be transformed to B by 0 or many operations of these sorts.

If no β-reductions are possible on B either immediately or after some α-reductions, B is said to be in normal form and then if A *red* B, B is a normal form of A.

Consider the term ϕ as in (1). $\phi\phi$ reduces via one β-reduction to $f(\phi\phi)$; in this sense $\phi\phi$ is a fixed point of the function f. The term $Y =_{\mathrm{def}} \lambda f.\phi\phi$ being applied to arbitrary F produces a fixed point of F, i.e.,

$$YF \ \mathit{red}\ F(YF). \tag{3}$$

Y is the so-called paradoxical combinator (fixed point combinator). The λ-calculus mimics the computation of a program by reductions. Church identified the positive integers $1, 2, 3, \ldots$ with λ-terms (in normal form) $\hat{1}, \hat{2}, \hat{3}, \ldots$ defined as follows:

$$\lambda fx.fx,\ \lambda fx.f(fx),\ \lambda fx.f(f(fx)), \ldots$$

Now, given a closed (i.e., without free variables) λ-formula F it expresses (λ-defines) the partial function f such that, for each positive integer n, $f(n) = m$ or $f(n)$ is undefined, according to whether $F\hat{n}$ *red* $\hat{m}$ or not. This is a correct definition since for a given n there can be at most one m such that $F\hat{n}$ *red* $\hat{m}$ (consequence of the Church-Rosser theorem). On the other hand, starting from $F\hat{n}$ the reduction process may be performed in a determinate and calculable way. In this sense a λ-definable function is "effectively calculable".

Finally, in 1936 Church published his definite proposal, which due to Kleene is well known as

Church's Thesis: *The λ-definable functions are* all *the effectively calculable functions.*

Note: Due to the relation between recursion and fixed points, the use of self-application is very useful in producing λ-definitions for functions. In order to mimic recursion Turing suggested the use of the fixed point combinator to provide λ-definitions that are less complicated than those originally elaborated by Kleene.

For readability we use mixed expressions in the sequel with many argument functions and other familiar mathematical notations, though in all cases we could keep the standard formal syntax of λ-terms.

Let us now return to the semantical difficulties caused by self-application. A simple way to avoid them would be to use the typed λ-calculus which does not allow self-application at all. In fact we shall consider a family of languages parametrized by Ω—the set of primitive types (e.g., integers, booleans, etc.). Moreover, we shall also parametrize with respect to C—the set of constants of the language, abandoning in this way the commitment to consider "pure" λ-calculus. The formal definitions are as follows:

Type expressions (or simply-types):
Each element of Ω (each primitive type) is a type.
If α, β are types, so is $\alpha \rightarrow \beta$.
Now, $C = \underset{\alpha}{U} C^{\alpha}$ where C^{α} is the set of constants of type α. For each type α an infinite set X^{α} of variables of type α is considered as well; $X = \underset{\alpha}{U} X^{\alpha}$.

The languages $L(\Omega, C)$ consists of typed terms; $L^{\alpha}(\Omega, C)$ is the set of terms of type α. Below we omit the parameters Ω, C supposing them fixed:

Basis: $C^{\alpha} \subset L^{\alpha}, X^{\alpha} \subset L^{\alpha}$

Application: $u \in L^{\alpha \rightarrow \beta}, v \in L^{\alpha}$ implies $(u\,v) \in L^{\beta}$

Abstraction: $x \in X^{\alpha}, u \in L^{\beta}$ implies $\lambda x.u \in L^{\alpha \rightarrow \beta}$

Binding and reductions are as in the untyped case. Clearly, $L(\Omega, C)$ is a proper part of the untyped language with constant form C. In particular no self-application may occur in a typed term.

3. LISP

LISP was designed by J. McCarthy in the late fifties–early sixties. It is the second (after FORTRAN) oldest programming language which is still widely used, especially for work on artificial intelligence. The design of LISP relies on several fresh ideas which became quite popular, e.g., computing with symbolic expressions (rather than numbers) and their appropriate representation by list structures. However, for our consideration, those features that are inherited from the lambda calculus, are more relevant.

1. The most striking feature is the explicit use of the LAMBDA operator for naming functions:

> To use functions as arguments, one needs a notation for functions, and it seemed natural to use the λ-notation of Church *1941*. I didn't understand

> the rest of the book, so I wasn't tempted to try to implement his more general mechanism for defining functions. Church used higher order functionals instead of using conditional expressions. Conditional expressions are much more readily implemented on computers. (Quotation from McCarthy in Wexelblat *1981*, p. 173–197)

2. LISP promoted the idea of treating basic operations (*car*, *cdr*,... and ultimately even conditionals) as functions. Hence, they could be composed and dealt with as constants of the λ-calculus. Some slight deviations from this standard were recognized in the sequel. So, as D. Park pointed out,

> The LABEL notation invented by N. Rochester was logically unnecessary since the result could be achieved using only LAMBDA—by a construction analogous to Church's (paradoxical) Y-operator, albeit in a more complicated way. (Wexelblat *1981*).

3. LISP also inherited "currying" from Church's Lambda Calculus. McCarthy recognized procedures as functions of one argument; it is possible to apply one procedure to another and to return a third one as a result.

4. The representation of LISP programs as LISP data is in full accordance with the type free lambda calculus. McCarthy's universal function EVALUATE strongly relies on this idea.

5. The clear vision that the functional style of programming is the most appropriate way to assure referential transparency. At first McCarthy considered it important to express programs as applicative expressions built up from variables and constants using functions, i.e., to pursue a functional applicative style of programs in which side effects are avoided (Pure LISP).

Let us now mention two essential points in which LISP deviated from the λ-calculus ideal.

The first one concerns the introduction of "dirty" features such as assignments and goto's. Considering that tricks with side effects may be a source of computational efficiency, in the sequel McCarthy gave up the Pure LISP. (Much later, for similar reasons, other languages (for example, ML) that are generally recognized as functional languages resorted to some of these "dirty" features.)

But the most striking deviation from the λ-calculus legacy is the dynamic scope discipline of LISP as opposed to the static scope of the λ-calculus. Thus, (up to specific LISP notations) the expression

$$\lambda x.(\lambda p.((\lambda x.px)0)(\lambda u.(x+2))1$$

would be evaluated to 2 in LISP whereas its value according to the static scope is 3.

That this was not a conscious decision, but rather an unfortunate oversight of McCarthy, is testified to in his survey (Wexelblat *1981*). It is a very instructive story, and it is worth recalling that

> James R. Slagle programmed ... LISP function definition and complained when it did not work right ... In modern terminology, lexical (static) scoping was wanted but dynamic scoping was obtained ...
>
> Firstly, ... I regarded this as a bug and expressed confidence that it would soon be fixed. Unfortunately, the devices invented to fix it were not able to manage definitely with the problem ...

After all, modernized and sanitized versions of LISP such as SASL (Turner) and LIPSKIT (Henderson) abandoned this unfortunate dynamic scope and restored the original scoping of the λ-calculus.

4. *ISWIM*

In 1966 Peter Landin published his famous paper (Landin *1966*) under the intriguing title, "The Next 700 Programming Languages", where he introduced the language ISWIM (If you See What I Mean). In comparison with the λ-calculus the major innovation is in an additional binding mechanism through the *let* and *letrec* constructs. These constructs significantly improve the binding by abstraction permitting the statement of declarations (definitions) in a convenient way. At the same time ISWIM was also high order: hence functions which have other functions as arguments could be declared. We limit ourselves here to a few examples which will hopefully illustrate this point:

$$let\ (x = y + 1)\ in\ (x^2 - x + 1) \tag{1}$$

may be considered as a notational version ("syntactical sugar"—according to Landin) of the standard λ-expression

$$(\lambda x.x^2 - x + 1)(y + 1). \tag{2}$$

What is gained here is only a more articulate and more suggestive exhibition of the relevant subexpressions: the definition $x = y + 1$ with "definiendum" x and "definiens" $y + 1$, and the "main part" (i.e., the user of the definition) $x^2 - x + 1$.

On the other hand, *letrec* introduces recursive definitions and provides a more substantial improvement: note that

$$\textit{letrec } f = \lambda n.(\textit{if } \mathrm{equal}(n, 0) \textit{ then } 1 \textit{ else } n \cdot f(n-1)) \textit{ in } f(3) \tag{3}$$

is quite different from

$$\textit{let } f = \lambda n.(\textit{if } \mathrm{equal}(n, 0) \textit{ then } 1 \textit{ else } n \cdot f(n-1)) \textit{ in } f(3). \tag{4}$$

In (3) the defined f binds the occurrences of f in both the definiens and the main part $f(3)$; hence the intended value of the whole ISWIM expression is $3! = 6$

In (4) unlike (3) only the occurrence of f in $f(3)$ is bound, whereas the occurrence in $f(n-1)$ is free and hence refers to a global function supplied by the environment (say, by an external declaration).

Usually, instead of (3) and (4) ISWIM would use the sugared forms, say, (3′) below instead of (3):

$$\textit{letrec } f(n) = \textit{ if } \mathrm{equal}(n, 0) \textit{ then } \ldots \tag{3′}$$

The *letrec* construct was conceived by P. Landin as syntactical sugar for specific λ-terms which use the (paradoxical) fixed-point operator Y. For example (3) might be desugared to (the nonrecursive declaration)

$$\textit{let } f = Y(\lambda f.\lambda n(\textit{if } \mathrm{equal}(n, 0) \ldots)) \textit{ in } f(3)$$

which might be further desugared, like (1) to (2).

In general, ISWIM allows arbitrary mutual declarations and nested declarations as illustrated in (5):

$$\begin{aligned}&\mathit{letrec}\ (f(n) = (\mathit{if}\,\mathrm{equal}(n,0)\ \mathit{then}\ 1\ \mathit{else}\ f(n-1)+g(n-1))\ \mathit{and}\ g(n) = 2n+3)\\ &\mathit{in}\quad (\mathit{let}\ f(n) = (\mathit{if}\,\mathrm{equal}(n,0)\ \mathit{then}\ 1\ \mathit{else}\ n\cdot f(n-1))\ \mathit{in}\ f(3))\end{aligned} \tag{5}$$

Consider the defined f in the mutual recursive declaration for f and g; clearly by this declaration it is specified as the function $\lambda n.(n+1)^2$. On the other hand it binds the occurrence of f in $n \cdot f(n-1)$; hence the value of the whole expression will be $3 \cdot (3+1-1)^2 = 27$.

The *let* and *letrec* constructs reflect the down-top style of definitions and their use, and this is the common style in programming. In more common programming notations instead of, say

$$\mathit{let}\ f(n) = n + y\ \mathit{in}\ f(5) \tag{6}$$

one might use something like

$$\begin{aligned}&\mathit{define}\ f : \mathit{function}\ (n);\\ &\quad \mathit{return}\ n + y\\ &\mathit{end};\\ &\quad f(5).\end{aligned} \tag{6'}$$

In fact, ISWIM also allows as synonymous for *let* and *letrec* the *where* and *whererec* formats which exhibit a top-down approach. For example, in the simple case of (1) the "where" format looks like:

$$\begin{aligned}&x^2 - x + 1\\ &\mathit{where}\\ &x = y + 1\end{aligned} \tag{1'}$$

Often, when the λ-formulas of LISP programs are cumbersome and deeply nested, the use of the "*let*" and "*letrec*" constructs provide more readable notations which come closer to the conventional mathematical style of formulating and referring to definitions.

In addition to the reduction rules of the λ-calculus, the operational (reduction) semantics of ISWIM also includes specific rules for manipulation with declarations. For example, in the case of a block with a simple (nonmutual) declaration, expansions of calls are performed according to the

Expansion Rule:

letrec $x = N$ in M reduces to *letrec* $x = N$ in $M[x := N]$

Note that for the nonrecursive *let*, this rule in fact yields a slightly improved sugaring of the β-reduction of the λ-calculus (compare (1) and (2) above).

We considered above the untyped ISWIM. In order to get its typed version ISWIM (Ω, C) we have only to extend the syntax for $L(\lambda, C)$ with declaration clauses. For example, using I^α as a shorthand for $\mathrm{ISWIM}^\alpha(\Omega, C)$ we have:

Block with mutual recursive declarations:
$x_1 \in X^{\alpha_1}, \ldots, x_k \in X^{\alpha_k}$
$u_1 \in I^{\alpha_1}, \ldots, u_k \in I^{\alpha_k},\ v \in i^\beta$ imply

$$\mathit{letrec}\ (x_1 = u_1\ \mathit{and}\ \ldots x_k = u_k)\ \mathit{in}\ v \in I^\beta.$$

ISWIM was originally a purely functional language, and here we consider it only as such, although Landin later added an imperative feature—the "program point". ISWIM (unlike LISP) obeys the static scoping of the λ-calculus with respect to both binding mechanisms, hence renaming bounded variables will in no way affect the meaning of an ISWIM program.

One other deviation from the λ-calculus is that ISWIM evaluates "by value", whereas the λ-calculus evaluates "by name". For example:

$$(\lambda x.6)(5/0) \tag{7}$$

returns the value 6, whereas *let* $x = 5/0$ *in* 6 requires 5/0 to be evaluated first and hence is not defined. This is not a dangerous deviation for two reasons:

1. Call by name and call by value cannot produce two different values (as the alternative scoping rules may do). The worst that can happen (see the example (7)) is that evaluation by name produces a result, whereas call by value is not defined.
2. Call by value can be modeled through call by name plus appropriate primitive constants.

Later, the ISWIM innovations became widely used in all programming languages which developed the λ-calculus paradigm.

We mention here only two such developments: The first one is Edinburgh ML (Gordon et al. *1979*), with a rich type structure and many facilities which in particular facilitate interactive proofs of programs (hence the initials ML for Meta Language).

The second one is LUCID (Wadge and Ashcroft *1985*), conceived by its authors as a Data Flow Language, with the intention of a parallel implementation. In fact LUCID may be desugared to ISWIM with an appropriate set of constants. (The authors deal mainly with untyped ISWIM).

But beyond the important technical aspects of his contribution, Landin was the first to fully grasp and promulgate what we would like to call

The Church-Landin Thesis: *Programming Languages are λ-calculus sweetened with specific sugar.*

Remarkably, from the very beginning Landin had in mind not only functional languages but also imperative ones such as Algol (Landin *1965*). On one hand, he defined the so-called SECD machine, on which λ-formulas may be manipulated in the same way as common imperative programs in real computers. On the other hand—and that was the heart of the thesis—he had the clear vision of the λ-calculus underlying both functional and imperative languages.

5. *Semantics*

It is not the purpose here to advocate the importance of having formal (mathematical) means for the specification of the semantics of programming languages; clearly, relaxing the requirement of formality may lead to inconsistencies (Knuth *1967*). The need of precise formal syntax was recognized much earlier, and FORTRAN and later ALGOL already established high standards of syntax specification. The problems with semantics were and still are considerably more difficult.

In Sections 3 and 4 semantics of λ-calculus and of ISWIM was specified by the computational behavior of symbolic manipulation (say—reductions of the λ-calculus). This sort of specifying semantics is called *operational,* and it is quite different from the *denotational* one, which is generally accepted in mathematics and logic. In contrast to operational semantics, the denotational meaning of a piece of syntax is an object belonging to a (hopefully) well-understood semantical domain. Moreover, the meaning of an expression is defined by induction through the meanings of its subexpression. Hence the denotational approach provides a referentially transparent semantics which obeys the *compositionality principle* of the kind usual in mathematics and logic.

Scott and Strachey (*1974*) took on the challenge of elaborating a denotational semantics (they used the term "mathematical semantics") for programming languages. Scott promoted the fundamental idea that models of the λ-calculus might be an appropriate tool for providing programming languages with formal semantics. Moreover, it was clear that the type-free λ-calculus had to be dealt with. As in general for other algebraic calculi, it is not difficult to consider a superficial model by taking appropriate equivalent classes of syntactical object. For the λ-calculus, it would roughly amount to factorizing the class of all λ-terms with respect to convertibility; but such a model would be of little help. The famous solution presented by Scott in 1969 was the construction of λ-models based on *continuous lattices*. After that Scott developed the theory of *domains*, as the basis for mathematical semantics of programming languages.

For the typed λ-calculus and typed ISWIM, where no self-application may occur, models for denotational semantics are easy.

To define the semantics of $L(\Omega, C)$ we need a *type-frame* $D(\Omega, C)$, that is a family of sets D^α called domains, one for each type α such that:

1. $D^{\beta\to\gamma}$ consists of *some total* functions from D^β into D^γ.
2. The constants from C^α are interpreted by some elements from D^α.

An environment for a type-frame $D(\Omega, C)$ is a mapping $e : X \to D$ which respects types. $e[d_1/x_1, \ldots, d_k/x_k]$ is the environment that differs from e only on the variables $x_1, \ldots, x_n$ for which the values $d_1, \ldots, d_n$ are assumed. Env denotes the set of all environments.

Establishing a semantics for L amounts to specifying a "nice" mapping

$$[\![\ \ ]\!] : L \times \mathrm{Env} \to D$$

which respects types; $[\![u]\!]e$ is the meaning of the term u under the environment e.

We expect that these meanings are consistent with the operational semantics; in particular reductions should preserve the meanings of terms. This is indeed the case if we assume that D is a *full frame*, that is for each β, γ the domain $D^{\beta\to\gamma}$ consists of *all total* functions from D^β into D^γ. It is very easy to prove:

Theorem: *Assume D is a full frame. Then there exists a (unique) mapping $[\![\ \ ]\!]$ which correctly interprets the constants from C and such that:*

0) for $x \in X^\alpha$ $[\![x]\!]e = e(x)$
1) *Application:* $[\![uv]\!]e = [\![u]\!]e\,([\![v]\!]e)$
2) *Abstraction:* $\forall d \in D^\alpha\ \forall x \in X^\alpha\ [\![\lambda x.u]\!]e\ (d) = [\![u]\!]\ e[d/x]$

In fact, in addition to the full frame, there may be other frames for which the theorems holds as well. All such frames are called λ-models of $L(\Omega, C)$.

Call λ-terms u, v denotationally equivalent (notation $u \approx v$) iff in each model:

$$[\![u]\!]e = [\![v]\!]e \text{ whatever the environment } e \text{ may be} \qquad (*)$$

The claims we formulate below point to fundamental properties of denotational semantics in general; their analogues hold for other languages as well:

Claim 1 (Replacement Rule): *Assume that* $u \approx v$ *and that* w_2 *is the result of literally replacing (without renaming of bound variables) a subterm* u *of* w_1 *by* v. *Then* $w_1 \approx w_2$.

Claim 2 (Consistency of the denotational semantics with the operational one): *Assume the terms* u, v *are convertible (i.e., one can reduce* u *to* v *using* α*-reductions,* β*-reductions and expansions); then they are denotationally equivalent.*

Denotational semantics for ISWIM (Ω, C) is handled very similarly to, though a bit more complicated than, $L(\Omega, C)$. As in the case of $L(\Omega, C)$ one needs a type frame $D(\Omega, C)$, and again $D^{\beta\to\sigma}$ consists of some *total* functions from D^β into D^σ.

The main point is that one has to assure in each domain $D^{(\alpha\to\alpha)\to\alpha}$ the existence of a fixed-point operator A_α such that for all $f \in D^{\alpha\to\alpha}$ both (Yf) and $f(Yf)$ are the same element in D^α. It is also necessary that the fixed-point operators be chosen consistently with the structure of the frame. A well-known approach is to use *continuous models*; this amounts to considering frames whose domains are *complete partial orders* (*cpo's*) with continuous functions from D^α into D^β as elements of $D^{\alpha\to\beta}$. We won't go into detail here on this subject, but the following point should be emphasized. Since (according to the definition of a cpo) in each domain D^α there exists the specific element $\perp_\alpha$ - the undefined element of type α - considering only *total* functions is not a restrictive requirement (roughly—mimic a partial function by a total function which sometimes returns $\perp$).

Two ISWIM expressions E_1, E_2 are said to be equivalent (notation: $E_1 \approx E_2$) if they have the same meaning in all continuous models and for each environment.

As expected, the analogue of Claim 1 holds for ISWIM as well. Consistency with operational semantics is manifested by a lot of important equivalences. Here are some illustrations:

Expansion
$(letrec\ x = N\ in\ M) \approx (letrec\ x = N\ in\ M[x := N])$

Denesting :
$let\ x_1 = (let\ x_2 = u_2\ in\ u_1)\ in\ v \approx$
$let\ (x_1 = u_1\ and\ x_2 = u_2)\ in\ v$
assuming x_2 is not free in v

Explicit parametrization:
$(let\ f = u\ in\ v) \approx let\ h\,x = u[f := hx]\ in\ v[f := hx]$.

There are a lot of other important equivalences, concerning expansions of recursive calls, transformations of simultaneous declarations into iterated ones. We shall return to this topic later in Section 6.

Unlike the typed languages where things are relatively easy, for the type-free λ-calculus, the crux is to give a consistent meaning of self-application. Scott developed a rich theory of models of self-applicable functions for the semantics of type free λ-calculus as well as other programming languages. This theory relies on a novel understanding of the notion of function and application of a function to an argument because ordinary mathematical functions are not self-applicable.

Scott's deep and elegant theory was too overloaded with subtle mathematical details to be accessible to the broad programming community. That is why he returned to this topic over and over again, looking for a more conventional approach.

Even prominent experts in programming used to complain on the cumbersome machinery of denotational specifications. In his *1978* Backus estimated the state of affairs as follows:

> Why did the good mathematics of the Scott-Strachey approach not work? ... It results in a bewildering collection of productions, domains, functions and equations that is slightly more helpful ... than the reference manual of the language.

And here is a quotation from Scott *1982*:

> When I remember how much headaches I have caused to people in Computer Science who have tried to figure out the mathematical details of the theory

> of domains I have to cringe ...
>
> The difficulty in the presentation of the subject is in justifying the level of abstraction in comparison with the payoff; often the effort needed for understanding ... does not seem worth the trouble—especially if the notions are unfamiliar or excessively general ...
>
> It takes some time to learn the notations and terminology and to become comfortable with them, to gain sufficient intuition ...

However, it may be that, for the first time since Church started his programming exercises on the λ-calculus, λ-programs obtained a precise and robust model-theoretic semantics. Moreover, from now on mathematical semantics of constructs in diverse programming languages might be specified via translations into appropriate forms of λ-calculus.

6. *Imperative Features*

Two prominent features—assignments and goto's—are directly inherited from the von Neumann computer architecture. Both are completely alien to the spirit of λ-calculus and are nicknamed "dirty features" by the adepts of pure functional programming. The goto's were long ago recognized as a troublesome control mechanism by the pioneers of "structured programming" (Dijkstra: goto's are harmful). Assignments are the main vehicle through which the computer memory (or store) is altered by program execution. Due to them (even without goto's) one can count to achieve tricky side effects and computational efficiency. Imperative languages were intensively developed beginning with the early FORTRAN. Notably John Backus, the designer of FORTRAN, later joined the critics of the "dirty features"; in his Turing lecture (*1978*) he called for the liberation of programming from the von Neumann style and outlined a new functional language.

Instead of commenting on the controversy between functional and imperative programming we shall confine ourselves here to the following claim: λ-calculus is not solely the core of functional languages but it is also of exceptional relevance to imperative languages as well.

In order to give evidence to this claim we focus here on fully typed algol-like languages (Pascal comes close enough to them). These languages allow an extremely broad repertoire of program constructions: assignments, conditionals, command sequencing, recursion, high-order procedures with value parameters (call by value), variable parameters (call by reference) and procedure parameters. No restrictions are assumed on the nesting of blocks and of mutually recursive procedure declarations. Nevertheless, what we are going to explain is that despite the multifariousness of these features, the "true» syntax of Algol-like languages is nothing but ISWIM(Ω,C) with appropriately chosen parameters Ω, C.

Concretely, we consider an illustrative toy language PROG, whose syntax reflects familiar programming languages except for one essential point: an explicit distinction is made between the type *loc* of memory locations and the type *val* of storable values. (In common programming terminology one refers implicitly to this distinction via "left" and "right" values of expressions.)

For example in the assignment

$$x := cont(y) + a$$

x is of type *loc*; on the other hand in the value expression $cont(y) + a$, y is of type *loc*, a is of type *val*, and *cont* is intended to perform the explicit dereferencing from locations to their value contents. Note also the difference between the Boolean expressions

$$x = y \qquad cont(x) = cont(y);$$

the first one expresses sharing of locations, whereas the second one expresses equality of values.

Below, after having sketched some features of PROG we proceed to the description of a translation *Tr* from PROG into ISWIM. This translation is the first step in a two-step process of formalizing the semantics of programs π in PROG. The second one amounts to assign meanings to $Tr(\pi)$ in the standard ISWIM way. Programs simply inherit their semantics directly from the ISWIM terms in which they translate. Moreover, the abstract syntax, viz. parse tree of $Tr(\pi)$ is actually *identical* to that of π. The translation serves mainly to make explicit the binding conventions and the implicit type coercions of PROG. Having in mind these circumstances we argue that IWSIM (Ω, C) provides the *genuine* syntax of PROG—a worthy alternative for the ALGOL-jargon which came down through history.

Types of PROG. As primitive types consider

$$\Omega = \mathrm{def}\{loc, val, bool, stat\}$$

where *stat* is intended to be the type of statements and parameterless procedures.

Blocks and bindings in PROG. Procedures of all higher finite types derived from Ω may be declared, passed as parameters, and returned as values.

Procedure identifiers are bound in PROG via declarations occurring at the head of a procedure block, e.g.,

$$\begin{array}{ll} \mathit{begin} & \\ & \mathit{proc}\, P_1(\mathit{formal1}, \mathit{formal2}) \Longleftarrow \mathit{Body1}, \\ & \mathit{proc}\, P_2(\mathit{formal}) \Longleftarrow \mathit{Body2} \qquad \mathit{end}; \\ & \mathit{Blockbody} \\ \mathit{end} & \end{array} \tag{*}$$

Blocks with variable declaration have the format

$$\mathit{begin} \quad \mathit{var}\, x; \mathit{St} \quad \mathit{end}$$

where x is of type *loc* and St is a statement, i.e., a phrase of type *stat*.

The syntax preserving translations Tr. We choose the parameters Ω, C of ISWIM as follows:

1. Ω – coincides with the set of primitive types in PROG.
2. C consists of two parts:
   Σ – the signature of PROG, i.e., the set of function symbols and predicate symbols it uses (say, *plus, equal, less*$\cdots$)
   $\Delta = \{\mathit{cont}, \mathit{seq}, \mathit{var}, \mathit{assign}, \cdots\}$ contains symbols that correspond to the program constructors.

Procedure blocks are translated, using *letrec*. For example the block $(*)$ above is translated into

$$\begin{array}{l} \mathit{letrec}\ P_1(\mathit{formal1}, \mathit{formal2}) Tr(\mathit{Body1})\ \mathit{and} \\ \qquad P_2(\mathit{formal}) = Tr(\mathit{Body2}) \quad \mathit{in} \quad Tr(\mathit{Blockbody}) \end{array}$$

By translating procedure declarations in this way it follows from the definition of ISWIM that static scope applies.

Blocks with variable declarations are handled with λ-abstraction and with constant *var* of type $(\mathit{loc} \to \mathit{stat}) \to \mathit{stat}$:

$$Tr(\mathit{begin}\ \mathit{var}\ x;\ St) = \mathit{var}(\lambda x.Tr(St)).$$

Hence, the binding effect of *var* x in the block is reflected in the binding effect of λx on $Tr(x)$.

Other expressions and statements are translated directly by introducing suitable constants from Δ, but no binding operators.

Examples: Consider the value expression $cont(y)+a$. The intended meaning of $cont(y)$ is not a value, but a function from stores to values. (In Algol jargon such functions are called *thunks.*) Below we use $valexp$ as a notation for its type (a shorthand for $(loc \to val) \to val$). But then we expect also the coercion to $valexp$ for the type of a. Hence

$Tr(cont(y) + a) = plus(cont(y), Ka)$
with $cont$ of type $loc \to valexp$
K of type $val \to valexp$
$plus$ of type $(valexp, valexp) \to valexp$.

The translation of assignments is straightforward, e.g.,

$$Tr(x := cont(y) + a) = assign(x, Tr(cont(y) + a))$$

with $assign$ of type $(loc, valexp \to stat)$.

In this way the syntax preserving translation Tr may be accomplished and hence most of the semantics of PROG. All that remains is the appropriate choice of domains and checking that the constants in Δ are continuous functions and also that they are consistent with the underlying intuition.

All this is almost a routine exercise with the significant exception of var. Here is where one of the crucial points of the whole enterprise concerning the connection of local variables with global procedures and the semantics of local storage (Trakhtenbrot et al. *1984*) is hidden.

What are the advantages of presenting PROG in the ISWIM format?

An immediate benefit is that it reveals the fundamental difference between the binding mechanisms in blocks with variable declarations and in blocks with procedure declarations. This is made explicit by their contrasting translations into ISWIM; the first are translated using λ-abstraction and the constant var, whereas the second are translated using the *letrec* binding mechanism.

A principal consequence of the syntax preserving translation is that the basic properties of the procedural mechanism in programs can be recognized as direct consequences of more elementary and well-understood properties of ISWIM. Thus many familiar and relatively simple equivalent transformations of ISWIM correspond to significant transformations of programs. The soundness of these transformations is therefore independent of even the meaning of the primitive constructs of the programming language (i.e., ultimately—of the interpretation of the constants in Δ). This observation brings us to the clear distinction between two levels of abstraction for algol-like programs: on one hand there are the *program schemes* with uninterpreted signature Σ but still relying on the meaning of program constructs; they are just the objects studied in classical *comparative schematology.* On the other hand there are the λ-schemes (following the terminology of Damm and Fehr) which abstract from both Σ and Δ.

7. *Parallelism Features*

The λ-calculus is generally recognized as a language which specifies sequential (or serial) computations. In modern programming language design this feature had been deemed too restrictive; as a remedy various facilities of parallelism were suggested. Remarkably, the λ-calculus shares this drawback with the Turing approach to computability. To preserve its universality the generalized Turing algorithm needs to be given some kind of parallelism. Shepherdson (this volume) deals with this issue; the following two illuminating remarks are quoted from there:

> [1] Certainly what Turing actually concentrated on in his paper—written before the existence of electric computers—was human calculation. And ... Turing assumed that the computation was serial, proceeding as a sequence of elementary steps, whereas a machine may do several steps simultaneously.
>
> [2] Clearly, if one wants to include all conceivable machines one must allow parallel operation. In the case *usually considered* [my italics—B.T.] of computation over total structures, i.e., ones whose functions and relations are defined for all arguments, parallel procedures are no more powerful than serial ones, for one oan obviously serialize a parallel procedure by subdividing the time scale. This is no longer true if there are partial functions, e.g., the function f defined by
>
> $$\begin{aligned} f(x) &= x, \text{ if } f_1(x) \text{ is defined or } f_2(x) \text{ is defined} \\ &= \text{undefined otherwise,} \end{aligned}$$
>
> obviously cannot be computed by any serial procedure because it might choose the wrong one of f_1, f_2 to evaluate first. And there is no bound on the number of processing units which may need to be simultaneously active as is shown by the example
>
> $f(x) = x$ if any of $f_1(x)$ or $f_1(f_2(x))$ or $f_1(f_2(f_2(x)))$ or ... ad inf, is defined.

The remark about serializing parallel procedures points out a possible source of confusion arising when one tries to formalize the notion of sequential function and to contrast it with the notion of parallel function.

This distinction makes sense when the domain and the range of the function are appropriately structured and do not reduce to (what Shepherdson refers to as being) a "total structure".

Church and von Neumann, independently of each other, formulated and investigated a model of growing automata with cells working concurrently and in a synchronized way. But the computations performed by Church-Neumann automata can be serialized just for "total structure" reasons. Therefore this model does not compute any parallel functions, despite the explicit exhibition of an unbounded number of processing units which are simultaneously active.

On the other hand, recall that in semantical models of ISWIM whose domains have a cpo structure there occur de facto partial mappings, albeit disguised as total functions. Hence, one can expect the distinction between sequential and parallel functions to make sense. A typical example is the parallel OR, which unlike the sequential one, returns the value *true* when one of its arguments is true, even if the other one is $\perp$ (undefined). OR behaves essentially as the function $f(x)$ in Shepherdson's example. Similarly one can contrast the parallel and the sequential versions of the conditional "if...then...else".

Precise definitions of the notion "sequential function" (and by negation of the notion "parallel function") for different well-structured domains were formulated and investigated in Kahn and Plotkin *1978*, Plotkin *1977*, and Sazonov *1976*. In general these definitions are consistent with each other and support the hope that indeed they capture the essence of the phenomenon.

Now returning to the λ-calculus as a programming language, the questions arise: (i) in what precise sense is it sequential (serial)? and (ii) what might be the ways to enrich it with parallelism?

As to the first question, one of the possible explanations is the following: Assume that in a model of ISWIM (Ω, C) all constants from C are interpreted as sequential functions; then each function definable in ISWIM (Ω, C) is also sequential. In other words this phenomenon may be explained as follows: application (as a binary mapping) and fixed points operators (as unary mappings) are sequential functions.

One way of enriching ISWIM with parallelism might be to impose on some of the constants in C specific interpretations requiring them to be parallel functions (say, parallel conditionals). This approach is reminiscent of the one used earlier in Section 6 to enrich ISWIM with imperative features; it was realized by Plotkin *1977* and Sazonov *1976*, where the comparative power at parallel functions (constructs) was also analyzed.

As an alternative attractive approach it would be nice to incorporate concurrent computations directly into the calculus. Milner *1984* and Hoare *1985* created special calculi for concurrent computation, conducted by communication among independent agents: CCS (Milner) and CSP (Hoare). This trend is developing very successfully, both in theory and in applications to programming and communication. Unfortunately, its relationship to the λ-calculus and to the notion of parallel function as explained above is not clear.

In his *1984* Milner testifies:

> Sequential computation has a well-established (model) theory, due to the λ-calculus, which existed long before any notion of implementing a programming language. The λ-calculus was (and is) a paradigm for evaluation, in the same way that the predicate calculus is a paradigm for deduction. More recently and largely due to Dana Scott, the model theory of the λ-calculus has grown and has been harmonized with the evaluation theory.
>
> CCS is an attempt to provide an analogous paradigm for concurrent computation, conducted by communication among independent agents. It arose after several unsuccessful attempts by the author to find a satisfactory generalization of the λ-calculus, to admit concurrent computation.
>
> The relationship between a calculus for communication and the lambda calculus is far from clear.
>
> The notion of higher-order function which fit so well with the λ-calculus, seems to find no obvious generalization in the setting of concurrent communicating systems.

8. Concluding Remarks

The previous sections hopefully gave evidence to the claim that in many essential aspects high-level programming languages are inherited from the λ-calculus. This claim was referred to as the Church-Landin Thesis.

Dealing with lambda definitions for numerical functions, Church pursued the sole object of giving a precise characterization of computability This was before the existence of computers and programming languages Certainly, it would be an anachronism to discern in it direct and explicit indications on the future development of high-level programming languages All this was realized only later, especially due to Landin and Scott's investigations.

Let us shortly recall the main features of the λ-calculus and its ISWIM extension, considered as functional programming languages:

1. Abstraction as a fundamental binding mechanism, to which other bindings may be adequately reduced.
2. Declaration mechanism via the *letrec* construct (ISWIM)
3. Static scoping rule for bindings; hence bound variables play the role of

place holders and may be renamed.

4. Call by name (by substitution) as an evaluation rule. Call by value can be mimicked.
5. Clear distinction between the flexible type-free language and the more restrictive, but more amenable, typed language.
6. Currying, as a way to deal only with one-argument operators.

And, in addition to these syntactical features,

7. An illuminating denotational semantics, which serves as a robust guide in reasoning about programs.

On a high schematological level even imperative and/or parallel programming languages behave as the λ-calculus and may inherit most of the features listed above.

Less clear is the relation between the λ-calculus and the calculi for communications that were developed to support parallel programming.

But actually the post-ISWIM development of the λ-calculus was and still is impetuous, and so is its impact on programming language design. This comes to light basically through the elaboration of more developed type structure and type discipline—an area which is beyond the scope of this essay. Reynolds *1985* and Burstall and Lampson *1984* give a good idea of the activities in the area. The following illustrations are borrowed from these papers:

(i) Burstall and Lampson designed PEBBLE—a core language to support the writing of large programs in a modular way, which takes advantage of type checking. In order to achieve the goal of "programming in large" designers usually invent various features and sometimes make ad hoc decisions. As to PEBBLE, "it provides a precise model for those features, being a functional language based upon the λ-calculus with a peculiar type structure in which types are values. It is addressed to the problems of data types, abstract data types and modules."

(ii) Reynolds *1984* discusses, among other things, the polymorphic λ-calculus which was defined independently by Girard and Reynolds. Unlike the common "first-order λ-calculus" the polymorphic one (called also second-order λ-calculus) allows the definition of polymorphic functions by abstraction on type variables. For example, in the second order lambda-term

$$\Lambda\alpha.\lambda f^{\alpha\to\alpha}.\ \lambda x^{\alpha}.f(fx) \qquad (*)$$

the abstraction $\Lambda\ \alpha$ is on the type variable α, whereas $\lambda f^{\alpha\to\alpha}$ and λx^{α} abstract on "common" variables f and x of types $\alpha \to \alpha$ and α respectively.

The term (*) specifies the polymorphic "doubling" function that can be applied to any type (say *integer*) to obtain a doubling function for that type.

Certain forms of polymorphism are allowed in some widely used programming languages (e.g., ADA). But the polymorphic λ-calculus suggests a novel and powerful programming style with the following peculiarities:

1. all programs terminate;
2. a wide spectrum of data (integers, booleans, lists, ...) are represented as polymorphic functions.

In fact (2) is inspired by the way in which Church encoded natural numbers, truth values,... in his λ-definitions. For example the term (*) is nothing but a polymorphic variant of Church's untyped code for the number two.

And again we are faced with a wonderful anticipation: a seemingly ad hoc technicality in Church's λ-definitions is reincarnated after half a century in a novel approach to data representation.

References

Backus, J.

1978 Can programming be liberated from the von Neumann style? *Comm. ACM* **21** (1978) 613–641.

Barendregt, H.

1984 *The Lambda Calculus: Its Syntax and Semantics,* revised edition. Studies in Logic 103. Amsterdam: North-Holland Publ. Co. (1984).

Böhm, C.

1966 The CUCH as a formal and description language for computer programming. In: *Formal Language Description for Computer Programming,* ed. Steel. Amsterdam: North-Holland Publ. Co. (1966).

Burstall, R., and B. Lampson

1984 A kernel language for modules and abstract data types. In: *International Symposium on Semantics of Data Types,* Lecture Notes on Computer Science 173. Berlin: Springer-Verlag (1984).

Church, A.

1941 *The Calculi of Lambda-Conversion.* Princeton, NJ: Princeton University Press (1941).

Damm W., and E. Fehr

1980 A schematological approach to the procedure concept of Algol-like languages. In: *Proc. 5-iem Colloque Sur Les Arbres,* Lille, pp. 130–134 (1980).

Davis, M.

1982 Why Gödel didn't have Church's Thesis. *Inf. Contr.* **54** (1982) 3–24.

Fortune, S., D. Leivant, and M. O'Donnell

1983 The expressiveness of simple and second-order type structure. *J. ACM* **30/1** (1983) 1451–185.

Gordon, M., R. Milner, and C. Wadsworth

1979 *Edinburgh LCF*, Lecture Notes on Computer Science 78. Berlin: Springer-Verlag (1979).

Halpern, Y., A. Meyer, and B. Trakhtenbrot

1984 From Denotational to Operational and Axiomatic semantics for Algol-like Languages, Lecture Notes in Computer Science 164, pp. 474–500. Berlin: Springer-Verlag (1984).

Hoare, C.A.

1985 *Communicating Sequential Processes.* London: Prentice Hall (1985).

Kahn, G., and G. Plotkin

1978 Structures de donnees concretes. IRIA-LABORIA Report 336 (1978).

Kleene, S.C.

1979 Origins of recursive function theory. *Ann. Hist. Comp.* **3** (1979) 52–67; see also Conf. Rec. 20th Amm. IEE Symp. on FOCS, 1979, pp. 371-382.

Knuth, D.

1967 The remaining trouble spots in ALGOL-60. *Comm. ACM* **10/10** (1967).

Landin P.

1965 A correspondence between Algol-60 and Church's Lambda-Notation. *Comm. ACM* **8** (1965) 89–101 and 158–165.

1966 The next 700 programming languages. *Comm. ACM* **9** (1966) 157–166.

Meyer, A.

1982 What is a model of the Lambda calculus? *Inf. Contr.* **52** (1982) 87–122.

Milner, R.

1984 Lectures on a calculus for communicating systems. In: *Seminar on Concurrency CMU, Pittsburg.* Lecture Notes on Computer Science 197, pp. 197–220. Berlin: Springer-Verlag (1984).

Morris, J.

1968 *Lambda Calculus Models of Programming Languages.* Dissertation MIT (1968).

Plotkin, G.

1977 LCF considered as a programming language. *Theor. Comp.* **5** (1977) 223–257.

Reynolds, J.C.

1981 The essence of Algol. In: *Proceedings of the International Symposium on Algorithmic Languages,* eds. de Bakker and van Vliet. Amsterdam: North-Holland Publ. Co. (1981).

1985 Three approaches to type structure. In: *TAPSOFT Advanced Seminar on the Role of Semantics in Software Development.* Lecture Notes in Computer Science. Berlin: Springer-Verlag (1985).

Rosser, J.B.

1984 Highlights of the history of the lambda calculus. *Ann. Hist. C.* **6/4** (1984) 337–349.

Sazonov, V.

1976 Expressibility of functions in D. Scott's LCF language. *Alg. Log.* **15** (1976) 308–330 (in Russian).

Scott D.

1972 Continuous lattices. In: *Topics in Algebraic Geometry and Logic,* Lecture Notes in Mathematics 274, pp. 97–136. Berlin: Springer-Verlag.

1982 *Domains for Denotational Semantics,* Lecture Notes in Computer Science 140, ICSLP. Berlin: Springer-Verlag (1982).

Scott, D., and C. Strachey

1971 Towards a mathematical semantics of computer Languages. In: *Proceedings of a Symposium on Computer and Automata, New York* (1971).

Shepherdson, J.C.

1988 Mechanisms for computing over arbitrary structures. This volume.

Trakhtenbrot, B.

1976 Recursive program schemes and computable functionals. In: *MFCS, Proceedings 1976,* Lecture Notes in Computer Science 45, pp. 137–151 Berlin: Springer-Verlag (1976).

Trakhtenbrot, B., Y. Halpern, and A. Meyer

1984 *From Denotational to Operational and Axiomatic Semantics for Algol-like Languages,* Lecture Notes in Computer Science 164, pp. 474–500. Berlin Springer-Verlag (1984).

Wadge, W. and Ashcroft, E.

1985 *LUCID, The Data-flow Programming Language.* London: Academic Pres (1985).

Wexelblat, R. (ed.)

1981 *History of Programming Languages,* pp. 173–197. New York: Academic Press (1981).