

# Lambda-Calculus and Functional Programming

Jonathan P. Seldin  
 Department of Mathematics  
 Concordia University  
 Montréal, Québec, Canada  
 seldin@alcor.concordia.ca

The lambda-calculus is a formalism for representing functions.

By the second half of the nineteenth century, the concept of function as used in mathematics had reached the point at which the standard notation had become ambiguous.

For example, consider the operator  $P$  defined on real functions as follows:

$$P[f(x)] = \begin{cases} \frac{f(x) - f(0)}{x} & \text{for } x \neq 0 \\ f'(0) & \text{for } x = 0 \end{cases}$$

What is  $P[f(x + 1)]$ ? To see that this is ambiguous, let  $f(x) = x^2$ . Then if  $g(x) = f(x + 1)$ ,  $P[g(x)] = P[x^2 + 2x + 1] = x + 2$ . But if  $h(x) = P[f(x)] = x$ , then  $h(x + 1) = x + 1 = P[g(x)]$ . This ambiguity has actually led to an error in the published literature; see the discussion in (Curry and Feys 1958 pp. 81–82).

As the above explanation shows, it is possible to clarify the ambiguity by adding a paragraph of explanation or some other ad hoc device every time the notation is used. But this

was not sufficient for those who were constructing systems of symbolic logic for various reasons. The first of these was Gottlob Frege (1848-1925), who was trying to show that mathematics is all a part of logic and wanted a systematic notation for everything, including functions.

As a part of his formal system, he defined (Frege 1967) the graph (Werthverlauf, course-of-values) of a function  $f$ , which he denoted by  $\langle \dot{x} f(x) \rangle$ . Thus, the course of values of the squaring function would be denoted  $\langle \dot{x}(x^2) \rangle$  or  $\langle \dot{\alpha}(\alpha^2) \rangle$  (the Greek letters here being bound variables). It is extensional in the sense that  $\dot{x}(x^2 - 4x) = \dot{\alpha}(\alpha \cdot [\alpha - 4])$ , which follows since  $x^2 - 4x = x(x - 4)$ . In (Frege 1893), Frege defined the application of a graph  $\dot{x} f(x)$  to an argument  $\Delta$ , which he denoted  $\langle \Delta \cap \dot{x} f(x) \rangle$ . In general,  $\xi \cap \zeta$  is defined so that if  $\zeta$  is the graph of a function, then  $\xi \cap \zeta$  is the value of that function for the argument  $\xi$ , while if  $\zeta$  is not the graph of a function then  $\xi \cap \zeta$  is the graph of a propositional function which is always false. Note that  $\xi \cap \zeta$  is defined for all objects  $\xi$  and  $\zeta$  of the system. The principal theorem about this function is  $f(a) = a \cap \dot{x} f(x)$ . This function was extended to functions of more than one argument in (Frege 1893, §36) by the following trick: if  $F(x,y)$  is a function of two arguments, it is replaced by a function  $f$  of one argument such that  $f(a)$  is the function of  $y$  given by  $F(a,y)$ , and  $f(a)(b)$  is  $F(a,b)$ . Thus, for example, if  $F(x,y)$  is  $x - y$ , then  $f(2) = 2 - y$ , and  $f(2)(3) = 2 - 3 = -1$ .

The next formalism to deal with a notation for functions was that of Bertrand Russell (1872-1970), see (Russell 1908). It was later incorporated in his work with Alfred North Whitehead (1861-1947), (Whitehead and Russell 1910-1913). He used a notation isomorphic to Frege's, but only for propositional functions; other functions were treated as special cases of relations.

In the notation for propositional functions, Frege's « $\cap$ » was replaced by « $\epsilon$ », but the types restricted it so that it became essentially class (or set) membership.

In 1920, Moses Schönfinkel (1889–c. 1942) gave a seminar talk at Göttingen, later published as (Schönfinkel 1924), that carried this analysis of notations for functions a stage further. Schönfinkel's objective was to reduce the number of primitives in a system like that of (Whitehead and Russell 1910-1913) or like that developed by David Hilbert (1862-1943) in his lectures on the foundations of mathematics.

In his system, which he called a function calculus (Funktionenkalkül), there is only one operation that forms compound terms: it is the one Schönfinkel wrote « $fx$ », representing the application of a function  $f$  to an argument  $x$ ; it is defined for all terms. Schönfinkel adopted the trick of Frege given above for reducing functions of more than one argument to functions of one argument, although he seems to have developed it independently of Frege. In this notation, it is common to write « $fxyz$ » for « $((fx)y)z$ ».

The system has three basic constants,  $C$ ,  $S$ , and  $U$ . The constants  $C$  and  $S$  are characterized by transformations, so that  $Cxy = x$ . This means that for any term  $a$ ,  $Ca$  is the function which, for any  $x$ , has the value  $a$ ;  $Ca$  is thus the constant function whose value is  $a$ . The transformation rule for  $S$  is more complicated:  $Sxyz = (xz)(yz)$ . Other functions of a similar nature can be defined in terms of  $C$  and  $S$ , for example,  $SCCx = (Cx)(Cx) = x$ , so  $SCC$  is the identity function,  $I$ . Other functions which Schönfinkel defined in this way are a commuter  $T$  with the property that  $Tfxy = fyx$  and a compositor  $Z$  with the property that  $Zfgx = f(gx)$ . The constant  $U$  has a different character: it represents a logical operator that Schönfinkel wrote  $\langle fx \mid^X gx \rangle$  and interpreted to mean «for all  $x$ , not both  $fx$  and  $gx$ ».

Later in the 1920s, Haskell B. Curry (1900–1982) was led to essentially the same system. He had started reading (Whitehead and Russell 1910–1913) while he was still an undergraduate, and he noticed that in the very first chapter, which deals with propositional logic, there are two rules of inference; the first, *modus ponens*, which says that from  $p \supset q$  and  $p$  we may deduce  $q$ , is very simple in its formal structure, but the second rule, which allows the substitution of any formula for a propositional variable, is much more complicated. (The kind of complexity is that which would be noticed by somebody trying to write a computer program for this formalism.) In the mid-1920s, Curry decided to try to break down this rule of

substitution into simpler rules. In 1926, he realized that he could do this with essentially the same formalism as Schönfinkel had introduced; Curry called it combinatory logic. He did not discover Schönfinkel's paper until November 1927. Curry was later to become so completely identified with combinatory logic that the method of treating functions of more than one argument in terms of functions of one argument, which Frege used and which Curry took from Schönfinkel, has become known as currying.

In a series of papers (Curry 1929; Curry 1930; Curry 1931; Curry 1932; Curry 1933; Curry 1934 Properties; and Curry 1934 Foundations), he developed an axiomatic theory based on this formalism up to the point of including set theory. In (Curry 1929), he pointed out that for any term  $X$  formed from the atomic constants and variables  $x_1, x_2, \dots, x_n$ , there is a term  $Y$  formed from the constants alone such that  $Yx_1 x_2 \dots x_n = X$ , and in (Curry 1930) he gave axioms to make this  $Y$  uniquely determined (in terms of the equality that the system axiomatized). In (Curry 1933), he defined the term  $[x_1, x_2, \dots, x_n]X$  to be this term  $Y$ . It follows by a simple substitution that for any terms  $M_1, M_2, \dots, M_n$ ,  $Y M_1 M_2 \dots M_n$  equals the result of substituting  $M_1, M_2, \dots, M_n$  for  $x_1, x_2, \dots, x_n$  respectively in  $X$ , and hence  $[x_1, x_2, \dots, x_n]X$  behaves like a multiple version of Frege's graph. The logical connectives and quantifiers were defined as atomic constants without special properties with respect to equality; in (Curry

1930) the constants are  $\Pi$  for the universal quantifier and  $P$  for implication, so that  $\langle (\forall x)A \rangle$  would be represented in the system by  $\langle \Pi([x]A) \rangle$  and  $\langle A \supset B \rangle$  by  $\langle PAB \rangle$ . The atomic constants Curry had discovered on his own were  $I$  (identity) with  $Ix = x$ ,  $C$  (commutor) with  $Cfxy = fyx$ ,  $B$  (compositor) with  $Bfgx = f(gx)$ , and  $W$  (diagnalizer) with  $Wfx = fxx$ . His discovery of Schönfinkel's paper gave him Schönfinkel's  $C$ , which he called  $\langle K \rangle$ , and also Schönfinkel's  $S$ , now written  $\langle S \rangle$ . In (Curry 1929), the atomic constants are  $K$  and  $S$ , but Curry did not feel that he understood the meaning of  $S$  as well as he understood that of the other atomic constants, and so the atomic constants for the rest of the papers in the series are  $B$ ,  $C$ ,  $K$ , and  $W$ . (Actually, Curry only started using special type for the combinators in (Curry and Feys 1958), but the usage has become standard so I will follow it here.)

Meanwhile, Alonzo Church (1903-present) wanted to construct a modification of Frege's system in which the use of free variables would be avoided. One of his goals was that "every combination of symbols belonging to our system, if it represents a proposition at all, shall represent a particular proposition, unambiguously, and without the addition of verbal explanations." (He also proposed to avoid the paradoxes of set theory and logic by restricting the law of excluded middle.) Church made Frege's treatment of functions basic to his system in the sense that terms were formed from the variables by two operations: (1) application, which is denoted  $\langle MN \rangle$  as

in the work of Schönfinkel and Curry, which represents the application of a function  $M$  to an argument  $N$  and which is defined for any two terms  $M$  and  $N$ , and (2) abstraction, which is denoted  $\langle \lambda x.M \rangle$  and is defined (in Church's original system) for any term  $M$  and for any variable  $x$  which occurs free in  $M$ . Church's rules of inference included one for changing bound variables and another for the calculation of the values of a function by a rule which is now called rule ( $\beta$ ):  $(\lambda x.M)N$  can be replaced by the result of substituting  $N$  for  $x$  in  $M$ . Like Curry, he represented the logical connectives and quantifiers by atomic constants. He published this system in (Church 1932–1933), but much of his motivation did not become clear until later papers, such as (Church 1951; Church 1951–1954).

Church's original notation was actually more complicated than this, in that he wrote  $\langle \{M\}(N) \rangle$  instead of  $\langle MN \rangle$ . The notation shown here is that of later work in the lambda-calculus, starting with (Church 1941).

During this period, Church had two graduate students, Steven Cole Kleene (1909–present) and John Barkley Rosser (1907–1989). Kleene studied arithmetic in Church's system, and wrote (Kleene 1934) and (Kleene 1935).

Kleene used a representation of the natural numbers suggested originally by Church; 1 is represented by  $\lambda xy.xy$ , 2 by  $\lambda xy.x(xy)$ , 3 by  $\lambda xy.x(x(xy))$ , etc. Under this representation, 0 would have to be  $\lambda xy.y$ , but in Church's original system this is not a well-formed term since  $x$  does not occur (free) in  $y$ .

In versions of the lambda-calculus used today, this is a well-formed term, and these terms, when used to represent the natural numbers (including 0) are not called the Church numerals.

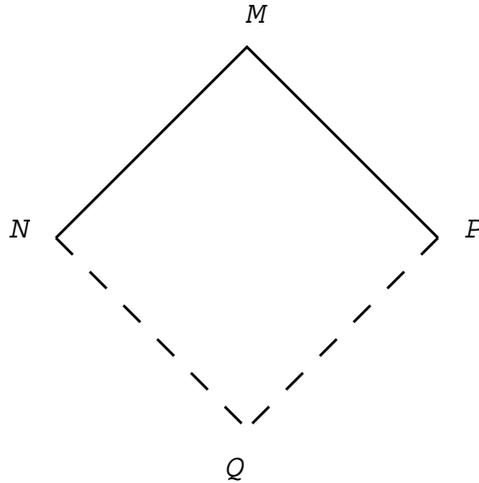
Rosser began work on a different version of combinatory logic, which led to his dissertation (Rosser 1935). In 1934, he and Kleene discovered something important about both the system of logic of Church and the combinatory logic of Curry: both systems are inconsistent in that they admit Richard's paradox (Kleene and Rosser 1935). Curry then began a thorough study of their paradox in detail; this led first to a new exposition of the paradox (Curry 1941 Paradox) and then to a new, simpler contradiction (Curry 1942 Inconsistency), which has since come to be known as Curry's paradox.

Curry's paradox can be stated fairly easily using the fixed-point operator (which Curry originally called the paradoxical combinator). This is the term  $Y$ , which can be defined to be  $\lambda x. (\lambda y. x(yy))(\lambda y. x(yy))$  and has the property that  $YF$  converts to  $F(YF)$ , thus making  $YF$  a fixed point of  $F$ .

Clearly, if  $N$  is negation, then  $YN$  is the fixed point of negation and converts to  $N(YN)$ , which is itself paradoxical. But Curry's paradox does not require negation; implication is enough. Suppose that in a system of combinatory logic or  $\lambda$ -calculus with an atomic constant  $P$  denoting implication, there is added a predicate «provability», with the following properties: 1) the scheme  $P(PX(PXY))(PXY)$ , which is a well known tautology that is also a theorem of intuitionistic proposi-

tional logic, is provable for any two terms  $X$  and  $Y$ , 2) the rule of modus ponens (from  $PXY$  and  $X$  to conclude  $Y$ ) is satisfied, and 3) any term convertible to a provable term is provable; then the system is inconsistent in the sense that any term  $Y$  is provable. For let  $Y$  be any term and let  $X$  be  $Y(\lambda x.Px(PxY))$ . Then  $X$  converts to  $PX(PXY)$ . It follows that the term  $P(PX(PXY))(PXY)$ , which is provable by hypothesis, converts to  $PX(PXY)$ , which is therefore provable, and since this converts to  $X$ , it, too, is provable. Hence, by modus ponens,  $PXY$  is provable, and, by modus ponens again, so is  $Y$ .

On the other hand, Church had good reasons to believe that the part of the systems dealing with the representation of functions was consistent. The corresponding part of combinatory logic had been proved consistent in (Curry 1930); it was only the system obtained by adding the additional postulates of (Curry 1934 Properties) that was inconsistent. (Kleene and/or Rosser told Curry that it was (Curry 1934 Properties) that gave them what they needed for their proof of inconsistency.) Church therefore had reason to believe that the corresponding part of his system was also consistent. Thus, he separated from the rest of the system of (Church 1932-1933) the part dealing with representing functions and calculating their value, and he called it the «lambda-calculus», and he and Rosser proved it consistent in (Church and Rosser 1936) by a result known as the Church-Rosser Theorem.



The Church-Rosser Theorem applies to more general systems than lambda-calculus. It says that if there is any rewriting procedure that makes it possible to transform a term  $M$  into terms  $N$  and  $P$ , then there is a term  $Q$  into which both  $N$  and  $P$  can be transformed by the same rewriting rules. When the rewriting rules involve replacements of parts of formulas, this implies that the order in which the parts are replaced does not make a difference. In the lambda-calculus, the rewriting rules are changes of bound variables and the rule ( $\beta$ ), and define a relation called reduction. The third rule of the lambda-calculus, which is the reverse of rule ( $\beta$ ), allows the calculations of the values of functions to be reversed, and is called expansion. When one term is transformed into another by reduction and expansion steps, the terms are said to be convertible. A corollary of the Church-Rosser Theorem, which is often taken as another form of the theorem itself, says that if two terms are convertible, then there is a

term to which both reduce. Furthermore, for those lambda-terms which have normal forms (i.e., which reduce to terms which cannot be further reduced), the Church-Rosser Theorem implies that normal forms are unique up to changes of bound variables. The Church-Rosser Theorem and its corollaries have become important in term rewriting theory, where a system of rewriting rules is not considered coherent unless it satisfies the Church-Rosser property.

At about the same time, Church began to realize that, in an important sense, the lambda-calculus is complete. As a result of his work on representing arithmetic in Church's original system, Kleene had discovered a great many functions that can be represented by lambda-terms. Since the lambda-term representing a function will calculate its values using a mechanical procedure (reduction using the rule ( $\beta$ )), any function which can be represented by a lambda-term is effectively calculable (mechanically calculable). As a result of Kleene's results and some similar results of Rosser, Church conjectured that any effectively calculable function of natural numbers can be represented by a lambda-term. This conjecture was the origin of Church's Thesis, and was published in (Church 1936 Unsolvability), where Church used the lambda-calculus to show that there are unsolvable problems of elementary number theory. The more usual form of Church's thesis, that all effectively computable functions are (partial) recursive, is a consequence of a proof in (Kleene 1936  $\lambda$ -definability) that a

function is recursive if and only if it can be represented by a lambda-term. The definition of a (partial) recursive function is due originally to Jacques Herbrand (1908–1931) and Kurt Gödel (1906–1978), but it was Kleene who really developed recursive function theory, beginning with (Kleene 1936 General). Nevertheless, as Rosser recalls in (Rosser 1984), Gödel did not believe Church's thesis until after Alan Turing (1912–1954) defined an abstract computer (now called a Turing machine) in (Turing 1936) and proved in (Turing 1937) that a numerical function can be computed by a Turing machine if and only if it is (partial) recursive.

What (Church 1936 Unsolvable) actually proved is that there are classes of problems of the form  $P(n)$  for each natural number  $n$  such that there is no general algorithm which, given  $n$ , will produce a solution to  $P(n)$ . The particular problems which Church proved unsolvable are 1) to determine for a given lambda-term whether it has a normal form, and 2) to determine for two given lambda-terms whether they are convertible. These problems become problems of elementary number theory when the terms are coded by natural numbers. Church called this coding «Gödel numbering» after the numbering introduced in (Gödel 1931), but today we can just as well think of the numerical coding that is automatically involved when anything is represented in a computer. Church used the results of (Church 1936 Unsolvable) to prove (Church 1936 Note; Church 1936 Correction) that there is no algorithm that will

decide whether a given formula of the first-order predicate calculus is a theorem.

The lambda-calculus was also used (Church and Kleene 1937) to represent some of the transfinite ordinal numbers. (Kleene 1938) separated representations of the transfinite ordinal numbers from the lambda-calculus. These two papers separated the transfinite ordinal numbers from set theory, where they had first appeared. This was important for logic at the time because Gerhard Gentzen (1909–1945) proved (Gentzen 1936) the consistency of first order arithmetic using transfinite induction on certain ordinal numbers (the ordinal numbers up to  $\varepsilon_0$ ). Separating these transfinite ordinal numbers from set theory was important in understanding just what this proof involved. This work of Church and Kleene was the beginning of the subject of ordinal diagrams, which is important in proof theory.

These results of Church and his associates are all presented in (Church 1941). Also found there is a variant of the lambda-calculus which is closer to Curry's combinatory logic in that the combinator  $K$  can be defined in it. Church called this « $\lambda$ - $K$ -conversion», and it has now come to be called «the  $\lambda K$ -calculus»; Church's original system is now called the « $\lambda I$ -calculus».

The  $\lambda K$ -calculus is formed by defining  $\lambda x.M$  to be well formed even if  $x$  does not have a free occurrence in  $M$ .

Meanwhile, Curry looked again at the foundations of his version of combinatory logic, and produced a revision of these foundations and a new consistency proof of the underlying system (Curry 1941 Revision; Curry 1941 Consistency). Here consistency was proved using the Church-Rosser Theorem, and completeness referred not to the representability of partial recursive functions but to the provability of an equation between any two combinatory terms which represent convertible lambda-terms.

During this same period, Rosser was working on the exact relationship between lambda-calculus and combinatory logic. This led to his paper (Rosser 1942), in which he defined abstraction in combinatory logic by a simple induction on the length of the term. (This definition also appeared in (Church 1941).) When Curry saw this and reformulated it in terms of the basic combinators he was using, he finally understood the role of Schönfinkel's **S**. (Rosser used a set of atomic combinators different from any used by Curry, and did not use **S** in his definition.) Although Curry saw this in the spring of 1942, it did not appear until (Curry 1949).

The definition is as follows:  $[x]x$  is **I**;  $[x]a$  is **Ka**, where  $a$  is a variable distinct from  $x$  or is an atomic constant; and  $[x]MN$  is **S**( $[x]M$ )( $[x]N$ ). This definition, or a variant of it, has become the standard way of defining abstraction in combinatory logic.

This work by Rosser and Curry clarified the connection between lambda-calculus and combinatory logic to such an extent that the two have been considered equivalent ever since. Aside from the differences in the basic rules of term formation, the main differences concern the properties of the reduction and conversion (equality) relations.

The natural reduction for combinatory logic, called weak reduction, does not allow replacements within the scope of an abstraction operator; this is because  $[x]M$  is an abbreviation for a term in which  $M$  and its subterms do not occur. On the other hand, in the natural reduction for lambda-calculus, replacements inside the scope of an abstraction are extremely natural. Thus, what is now called  $\lambda\beta$ -reduction satisfies a scheme which weak reduction does not, namely  $(\xi)$ : if  $M$  reduces to  $N$ , then  $\lambda x.M$  reduces to  $\lambda x.N$ . (The same is true if «reduces» is replaced by «converts».) This gives us two kinds of reduction and two kinds of conversion.

There is a third kind of each of reduction and conversion: Curry's original papers all included postulates to satisfy a principle of extensionality  $(\zeta)$ : if  $Mx$  converts to  $Nx$ , where  $x$  does not occur free in  $M$  or  $N$ , then  $M$  converts to  $N$ . It turns out that for conversion (but not reduction),  $(\zeta)$  is equivalent to the conjunction of  $(\xi)$  and  $(\eta)$ : if  $x$  does not occur free in  $M$ , then  $\lambda x.Mx$  converts to  $M$ . (Note that this says that everything converts to an abstraction.) If  $(\eta)$  for

reduction is added to the postulates for  $\lambda\beta$ -reduction, the result is usually called  $\lambda\beta\eta$ -reduction or  $\lambda\eta$ -reduction.

Equality relations in combinatory logic equivalent to  $\lambda\beta$ -conversion and  $\lambda\beta\eta$ -conversion are relatively easy to define. But reduction relations in combinatory logic corresponding to the corresponding  $\lambda$ -reductions are not so easy. Curry defined a combinatory reduction, called strong reduction in (Curry and Feys 1958 § 6F) which is equivalent to  $\lambda\beta\eta$ -reduction. However, a completely satisfactory combinatory reduction equivalent to  $\lambda\beta$ -reduction has still not been found. Curry was working on this with J. Roger Hindley (1939–present) and Jonathan P. Seldin (1942–present) at the time of his death in 1982 (see (Curry, Hindley, and Seldin 1984)), and Mohamed Mezghiche (1953–present) in (Mezghiche 1984) gives a proposal for this kind of reduction, and in (Mezghiche 1989) he gives a partial characterization of terms in normal form, but for none of the candidates for a combinatory beta-reduction do we yet have a complete characterization of terms in normal form.

These reactions to the Kleene-Rosser paradox led to ideas that are now considered fundamental in logic and computer science, but they did not advance the original objectives of Church and Curry in designing their original systems. When Church later came back to the elucidation of Frege's ideas in (Church 1951), he used a version of lambda-calculus with types that he had originally introduced in (Church 1940). The types of this system differ from the types of (Russell 1908; White-

head and Russell 1910-1913) in that the types are not natural numbers but are more like the data types of modern computer languages. Type theory based on this kind of typed lambda-calculus has since been studied by a number of people, including Leon Henkin (1921-present), who proved its completeness (Henkin 1950) in connection with the development of his proof of completeness of the first-order predicate calculus; see also (Henkin 1963). But the most important work on this type theory was done by Peter B. Andrews (1937-present), who has used it as the basis of a system of automatic theorem-proving TPS; see (Andrews 1965; Andrews 1986), including the references given there, for the type theory itself and (Andrews et al. 1988) for TPS.

Church's system has two atomic types,  $0$  (the type of propositions) and  $\iota$  (the type of individuals). Other atomic types have been used in recent variants of typed lambda-calculus, and these atomic types are now thought of as the basic types of most modern programming languages (such as Integer, Real, Boolean, String, etc.) The compound types are the type of functions from  $\alpha$  to  $\beta$ , which is now usually denoted  $\langle\alpha \rightarrow \beta\rangle$ , but which Church denoted  $\langle(\beta\alpha)\rangle$ . Russell's «first-order (propositional) functions» correspond to terms of type  $\iota \rightarrow 0$ , his «second-order (propositional) functions» correspond to terms of type  $(\iota \rightarrow 0) \rightarrow 0$ , etc. The types limit the formation of terms, since to apply  $M$  to  $N$ , the type of  $N$  must be  $\alpha$  and the type of  $M$  must be  $\alpha \rightarrow \beta$ ; then  $MN$  has type  $\beta$ . If  $x$  is

a variable of type  $\alpha$  and  $M$  is a term of type  $\beta$ , then  $\lambda x.M$  has type  $\alpha \rightarrow \beta$ .

One of the most important results concerning typed lambda-calculus is that every typed term has a normal form (i.e., it reduces to a term which cannot be further reduced). This was originally proved by Alan Turing, who was a graduate student at Princeton during 1936–38 and heard Church lecture on the type theory in 1937–38, but Turing never published this proof, and it was only published by Robin Gandy (1919–present) in (Gandy 1980). (The first published proof is due to Curry (Curry and Feys 1958 Theorem 9F9); many other proofs have appeared since.)

A stronger version of this result, known as the strong normalization theorem, says that every reduction sequence beginning with such a term terminates in a term in normal form; the standard method now used for proving it was introduced by W. W. Tait (1929–present) in (Tait 1967). For an exposition of this proof, see (Hindley and Seldin 1986 Appendix 2). This strong normalization theorem is now considered necessary for any variant of typed lambda-calculus.

Another important property of Church's type theory is that the typed lambda-terms can be interpreted in a set-theoretic universe, where the types are interpreted as sets of terms and  $\alpha \rightarrow \beta$  is interpreted as the set of all types whose domain is  $\alpha$  and whose range is a subset of  $\beta$ .

This kind of interpretation is impossible for pure lambda terms, since in a set-theoretic universe not all terms  $M$  and  $N$  can be interpreted in such a way that  $M$  is interpreted as a function and  $N$  as an object in its domain. The restriction on application terms with types guarantees that this kind of interpretation is always possible.

Meanwhile, Curry had been working on an idea that turned out to be very similar (although that was not to be clear for two decades); he called it the «theory of functionality», and it was about categories that turned out to be similar to Church's types. But Curry started earlier, presenting the idea to the American Mathematical Society in 1930 and publishing it in (Curry 1934 Functionality) and (Curry 1936).

Curry's idea differed from Church's type theory in that instead of having the types assigned by the rules of formation, they are assigned by the deductive rules of the system. There is a term  $F$  with the property that  $F\alpha\beta X$  means that  $X$  is a function from  $\alpha$  to  $\beta$ , so that  $F\alpha\beta$  corresponds to Church's type  $\alpha \rightarrow \beta$ . Curry originally defined  $F$  in such a way that  $F\alpha\beta X$  converts to  $(\forall x)(\alpha x \supset \beta(Xx))$ ; hence,  $F\alpha\beta X$  actually says something more general than that  $X$  is a function from  $\alpha$  to  $\beta$ ; it says that  $X$  is a function whose domain includes  $\alpha$  and that under  $X$  the image of  $\alpha$  is included in  $\beta$ . In the theory of functionality, variables are not assigned fixed types; rather, types are assigned to variables by assumptions. So whereas in Church's system, if  $x$  is a variable of type  $\beta$ , the term  $\lambda x.x$

has type  $\beta \rightarrow \beta$  for this particular type  $\beta$ , in Curry's theory of functionality,  $I$  has type  $F\alpha\alpha$  for any type  $\alpha$ .

Even before Kleene and Rosser discovered their inconsistency, Curry had planned to use the theory of functionality to help avoid paradoxes. After his study of their paradox, Curry developed a plan for trying to find "partially typed systems" (systems that are untyped in their basic formation rules but use the theory of functionality as an important part of their deductive rules) that could be proved consistent and would serve the purpose of his original system. He called this kind of system illative combinatory logic. To prove systems consistent, he planned to use the techniques of (Gentzen 1934). He discussed these plans in (Curry 1942 Combinatory; Curry 1942 Advances).

The plans were to look at systems based on three different sets of logical primitives. The first was to use as a primitive the term  $F$  of the theory of functionality. The second was to use a term  $E$  representing restricted generality;  $EXY$  has the logical properties of  $(\forall x)(Xx \supset Yx)$ . The third was to use primitives for the universal quantifier and implication, the theory of universal generality. Curry thought at the time that these three systems came in increasing strength. Curry did not get back to any of this until the 1950s, when he had already started working on (Curry and Feys 1958).

He had convinced Robert Feys (1889–1961) to work with him on this project because there was to be an emphasis on exposition, and Feys was very good at writing exposition.

He discovered that the most general possible form of the theory of functionality was inconsistent (Curry 1955).

This is the form of the theory in which any term can occur as a type.

It was not long before he found some versions of the theory which are consistent (Curry 1956). These results are all reported in (Curry and Feys 1958, Chapters 9–10).

The most important of these versions is the one called basic functionality in (Curry and Feys 1958, Chapter 9). Its characteristic is that terms can only occur as types if they are formed the way the types of Church's type theory are formed, from atomic types by the construction of  $F\alpha\beta$  from  $\alpha$  and  $\beta$ .

Curry was still thinking of the statement that term  $X$  has type  $\alpha$  as  $\alpha X$ , the application of  $\alpha$  to  $X$ . The way types are formed in the basic theory prevents any reductions in  $\alpha X$  that do not occur in  $X$ , but that is an accident of the basic theory, and in (Curry and Feys 1958, Chapter 10), there is an inference from  $K\beta X$  (saying that  $X$  has type  $K\beta$ ) to  $\beta$ .

In the mid-1960s, Curry had the idea of restricting the use of conversion rules in the theory of functionality so that conversions would have to take place either entirely within the type or entirely within the term. This made the theory of functionality less a foundational system for parts of mathe-

matics and more like the rules of term formation in Church's type theory, and in fact this version of the theory of functionality is now called type assignment.

Type assignment for the lambda-calculus is defined as a system of natural deduction in the sense of (Gentzen 1934), where the assumptions assign types to variables. There are two rules: one for introducing the arrow type  $(\Gamma, x : \alpha \vdash M : \beta \Rightarrow \Gamma \vdash (\lambda x.M) : \alpha \rightarrow \beta$ , provided that  $x$  does not occur free in  $\Gamma$ ) and one for eliminating it  $(M : \alpha \rightarrow \beta, N : \alpha \vdash MN : \beta)$ . In combinatory logic, the rule for introducing the arrow type is replaced by an axiom scheme assigning types to the atomic combinators:  $I : \alpha \rightarrow \alpha$ ,  $K : \alpha \rightarrow (\beta \rightarrow \alpha)$ , and  $S : (\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$ .

Although all partial recursive functions can be represented in the untyped lambda-calculus, not all such functions can be represented by terms to which types are assigned. This is not only because any numerical function represented by a typed term must be a total function; there are total recursive functions which cannot be represented by a term with a type. Which functions can be represented by typed terms depends on exactly how the numbers and functions are represented and what types they have.

If the Church numerals are used and no special arithmetic types are introduced, then the functions which can be represented are given by Helmut Schwichtenberg (1942-present) in (Schwichtenberg 1975-6). In this case, each of the Church nu-

merals has each type of the form  $(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$ . On the other hand, numerals can be represented by new constants  $0$  and  $\sigma$  (for zero and successor), provided that a primitive recursion operator or an iterator (a mapping from these new constants to the corresponding terms used in the Church numerals) is also introduced as a new constant, and an atomic type  $N$  for the natural numbers can also be introduced; in this case, we can represent more numerical functions, and, in fact, what we have is the system  $T$  of functionals of finite type introduced by Gödel for interpreting arithmetic in (Gödel 1958). For an exposition of this interpretation see (Hindley and Seldin 1986 Chapter 18).

In the 1950s, Curry noticed that if the terms are removed and the arrow interpreted as implication, then the axiom schemes for the basic combinators become well-known schemes for the implication fragment of intuitionistic propositional logic (and in fact form a complete set of axiom schemes for this logic) and the rule for eliminating the arrow types becomes the rule of modus ponens; see (Curry and Feys 1958 §9E). At the end of the 1960s, a number of people realized independently that it is possible to think of the types as formulas and of the terms as proofs, and, in doing this, extended the idea to other connectives and quantifiers : W. A. Howard (1926–present), N.G. de Bruijn (1918–present), H. Läuchli (1933–present), and Dana S. Scott (1932–present); for references see (Hindley and Seldin 1986 p. 194). Joachim

Lambek (1922–present) had a similar idea about the same time, but he emphasized the relation to category theory and was concerned with equivalence classes of proofs regarded as being, in some sense, the same proof; see (Lambek 1968; Lambek 1969; Lambek 1972; Lambek 1974). Of the papers that introduced this idea of formulas-as-types, (Howard 1980) has become so well known that the idea is often called the «Curry-Howard isomorphism», even though this does some injustice to the others. This idea has turned out to be very fruitful in proof theory.

After the publication of (Curry and Feys 1958), Curry turned his attention to the theory of restricted generality; see (Curry 1960; Curry 1961). Further work done with Jonathan P. Seldin (Seldin 1968, Chapters 4–5; Curry et al. 1972 Chapters **15–16**) obtained consistency proofs for systems strong enough to interpret first-order logic, and indicated that for these systems there is not that much difference between restricted generality and universal generality. An extension in which it is possible to mention propositions but not quantify over them is (Curry 1973); this system has interesting models, which Peter Aczel (1941–present) calls Frege structures; see (Aczel 1980; Scott 1975). Extensions to higher order logic have been made by Martin W. Bunder (1942–present), who has been interested in interpreting set theory in this kind of system and who has based all of his work on the theory of restricted generality; see (Bunder 1969; Bunder 1983 Predi-

cate; Bunder 1983 One; Bunder 1983 Set; Bunder 1983 Weak; Bunder 1986/7). In (Bunder 1984), he showed that category theory (which cannot be easily interpreted in set theory) can be interpreted in a system of restricted generality.

Frederic B. Fitch (1908–1987) also worked on systems of logic based on combinatory logic. But Fitch's systems, although provably consistent, have never attracted much attention, and they are not very strong. See (Fitch 1936; Fitch 1963; Fitch 1974; Fitch 1980 Consistent; Fitch 1980 Extension). Another logician who worked on type-free systems of logic was W. Ackermann (1896–1962), but his systems, although provably consistent, never caught on any more than Fitch's systems did; see (Ackermann 1950; Ackermann 1952–1953).

Still another approach to logic based on lambda-calculus is due to Solomon Feferman (1928–present), who has combinators which are not defined for all arguments. He has systems that can easily be shown to be conservative extensions of systems known to be consistent and are hence adequate for standard theories. See (Hindley and Seldin 1986 pp. 295ff) and the references given there.

In Feferman's systems,  $Ka$  (and hence  $Kax$ ) and  $Sab$  are defined whenever  $a$  and  $b$  are, but  $Sabx$  is not always defined even when  $a$  and  $b$  are. See the discussion of uniformly reflective structures below.

Additional work on lambda-calculus and logic has been done by Adrian Rezus (1949–present); see (Rezus 1981).

For some logicians, the interpreting of logic and set theory in combinatory logic or lambda-calculus was much less important than interpreting the type-free lambda-calculus in set theory. For these logicians, such an interpretation is necessary for the lambda-calculus to be more than an empty formalism. This was not a consideration for most logicians active in the 1930s, which was before most model theory had been developed, but by the 1950s it had become a major issue, especially in the United States. This is probably due to the influence of Alfred Tarski (1901-1983), the founder of model theory, who travelled from Poland to the U.S.A. to attend a conference scheduled to begin on 1 September 1939 and was unable to return to Poland after World War II broke out. As a result, he settled in the U.S.A. and had many students there at a time when a lot of government money was becoming available for scientific research. Thus, by the 1960s, the lack of models for the lambda-calculus had become a major obstacle to its acceptance among many logicians.

It is, of course, easy to create trivial models of the lambda-calculus: the term models, which are equivalence classes of terms under conversion. But these trivial models are not satisfying as an indication of the meaning of the lambda-calculus to logicians who need a model for such a meaning. At first, the problem of finding a non-trivial model for the lambda-calculus must have seemed hopeless. The fact that every term in the lambda-calculus can be applied to any

other term means that the domain of any model,  $D$ , would have to be isomorphic to the set of all functions from  $D$  into itself, but Georg Cantor (1845–1918) had already proved that impossible. But at the end of the 1960s, Dana S. Scott realized that it is not necessary for a model of the lambda-calculus that  $D$  be isomorphic to the set of all functions from  $D$  into itself; it is enough that  $D$  be isomorphic to a suitably rich subset of the set of functions from  $D$  to itself. Scott's idea was to use the set of continuous functions from  $D$  to itself for a set  $D$  with a suitable topology. Curry identifies the set of functions from the set of ordered pairs of elements of  $D$  into  $D$  with the set of functions from  $D$  to itself; this suggests using a class of topological spaces that form a cartesian closed category. Scott used the category of continuous lattices with an induced topology, and the model that results is usually called the  $D_\infty$  model. For references, see (Hindley and Seldin 1986 p. 154). He and Christopher Strachey (1916–1975) developed this idea into the subject of denotational semantics; see (Stoy 1977) and the works cited there.

Scott's model was studied mainly by Martin Hyland (1949–present) and Christopher P. Wadsworth (1946–present). Their work led to new syntactical insights. Among others who have studied this model are Luis E. Sanchis (1926–present), Mario Coppo (1947–present), Mariangiola Dezani-Ciancaglini (1946–present), Simonetta Ronchi della Rocca (1946–present),

C. P. K. Koymans (1957–present), M. Zacchi (1947–present), Gordon Plotkin (1946–present), M. B. Smyth (dates?), and Henri Volken (1945–present); for references, see (Hindley and Seldin 1986 p. 154; Barendregt 1984 Chapter 5 and Part V).

Since Scott's first model, others have been introduced. These include the model  $D_A$  due independently to Gordon Plotkin and Erwin Engeler (1930–present), studied by Guisepppe Longo (1947–present), the model  $P\omega$  due to Gordon Plotkin and Dana S. Scott, the model  $T^\omega$  due to Gordon Plotkin and studied by Henk Barendregt (1947–present) and Guisepppe Longo, the Böhm tree model, due to Henk Barendregt (Barendregt 1984 §18.3), the filter models, due to Henk Barendregt, Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betty Venneri (1951–present), the hypergraph model, due to Luis E. Sanchis, and the information systems model due to Dana S. Scott. Different kinds of models can be constructed by changing the set theory, as was done by Michael von Rimscha (1952–present). Still another alternative is to abandon the requirement that application always be defined; such partial models are the uniformly reflective structures of H. R. Strong (1942–present) and E. G. Wagner (1931–present), which were intended to be axiomatizations of abstract recursion theory, and which, again, show the relationship between lambda-calculus and the theory of effective computability. For a list of these models with their properties and references, see (Hindley and Seldin 1986 §12F). See also (Barendregt 1984 Chapter 5 and Part V).

Finally, it is also natural to model lambda-calculus in category theory instead of set theory; work on this has been done by J. Lambek, Dana S. Scott, and C. P. J. Koymans (among other people); see (Lambek and Scott 1986) and the references given there.

As more and more models appeared, the question arose of what it means to be a model of combinatory logic or the lambda-calculus. Finding a suitable definition turned out to be more complicated than first anticipated, and the definition of a model of the lambda-calculus is different (and more complicated) than that of a model of combinatory logic.

Among the people who have worked on this are Albert R. Meyer (1941–present), J. Roger Hindley, Giuseppe Longo, Henk Barendregt, C. P. J. Koymans, and Kim Bruce (1948–present). See (Meyer 1982; Hindley and Seldin 1986 p. 128) and the references given in the latter.

Models for typed lambda-terms or type assignment are much easier to construct, since, as pointed out above, the usual ideas from set theory are sufficient; the basic types are assigned sets, and the compound type  $\alpha \rightarrow \beta$  is assigned to the set of all functions from the set assigned to  $\alpha$  to the set assigned to  $\beta$ . (Hindley 1983 Completeness) proves the completeness of type assignment to lambda terms in terms of this very natural semantics, and (Hindley 1983 Curry's) proves completeness with respect to a semantics based on models of the lambda-calculus.

We have already seen (in the discussion of Church's thesis above) that the lambda-calculus is a kind of model for effectively computable functions. For this reason if for no other, it would seem natural to expect that lambda-calculus would have become connected with electronic computers. There is a connection, but much of its importance has only been seen in the last couple of decades. Since then, however, the connection to computers has come to overshadow all other aspects of lambda-calculus.

It is true that Curry, who took a leave of absence from The Pennsylvania State University to do applied mathematics for the U. S. Government during 1942–1946, became a member of the ENIAC team in late 1945 and was, for a period in 1946, the Acting Chief of the Computing Laboratory. As a result of this experience, he wrote a number of papers on computing, including one (Curry 1954) which proposed using combinators to combine programs into larger ones. But this does not seem to have had much influence on later developments in computers.

In 1960, John McCarthy (1927–present), who was working on LISP (McCarthy 1960), a language originally designed for symbolic computation using list representation (McCarthy 1981), needed a notation for functions and borrowed Church's lambda notation. But this was not really a use of the lambda-calculus; in fact, the theoretical basis for LISP, to the extent that it has one, is Kleene's theory of recursive functions

(McCarthy 1963). Nevertheless, LISP was the first functional programming language.

The idea behind functional languages is that instead of giving instructions about storing and modifying values, as is done in traditional, imperative languages, functional languages are based on the evaluation of expressions.

For example, to program the factorial function, whose value for  $n$  is  $n(n - 1)(n - 2)\dots 1$ , it is necessary to give the commands

```

INTEGER FUNCTION FAC(X)

INTEGER N

FAC = 1

FOR I = 1 TO N FAC = FAC*I.
```

(Here the first two lines declare the types of the variables, the third line initializes the value of the function, the last line does the work of the computation, and  $N$  is the number for which we wish to evaluate  $FAC$ . In some languages a separate program must be written for each value of  $N$ .) In a functional language, on the other hand, a complete definition using recursion can be as easy as the following:

```

fac n → if zero n then 1 else n * (fac (n - 1)) fi
```

Here the function zero takes an integer as argument and returns true if the integer is 0 and false otherwise, and if a then  $b$  else  $c$  fi evaluates  $b$  if the boolean value of  $a$  is true and evaluates  $c$  if the value of  $a$  is false.

In an imperative language, there are often side effects; a command may have an unintended effect on another part of the program (by changing a value stored in a particular location), and avoiding these side effects is one of the problems involved in writing a good program. On the other hand, in a pure functional language, in which evaluation of expressions is all there is, there are no side effects. Most functional languages are not pure, but make some compromise with the principles of pure functional languages for convenience of either use or implementation, and these languages do have some side-effects.

Clearly, the lambda-calculus is a paradigm of a pure functional language.

The first publications to suggest the importance of this use lambda-calculus in programming were by Corrado Böhm (1923–present) and Peter Landin (dates?); see (Böhm 1966; Landin 1964; Landin 1965). About the same time, Christopher Strachey proposed the use of lambda-calculus for semantics (Strachey 1966).

In the evaluation of expressions, it is often necessary to find an instance of a general pattern rather than a particular expression.

In the functional definition of the factorial function given above, it was necessary to determine whether a natural number is zero. But in a function on lambda terms implementing  $\beta$ -re-

duction, for example, it is necessary to determine whether or not an expression has the form  $(\lambda x.M)N$ , which means that the test here is for a pattern.

This is known as pattern-matching, and is a feature of almost all new functional programming languages. It was introduced independently by R. Burstall (1934–present) and David Turner (1946–present); see (Burstall 1969; Turner 1976).

John Backus (1924–present) emphasized the limitations of imperative languages and raised the level of awareness of functional programming in (Backus 1978), a paper which had a major impact.

Up until this point, there was interest in the idea of functional languages, but efficient implementations were a major problem. This began to change about 1980, when larger computers capable of serious symbolic calculation began to become available and affordable. But David Turner also helped things along with a new idea for more efficiency (Turner 1979).

Turner's idea concerned the way terms are represented in the computer. The natural way is to represent them as trees. But this can involve considerable duplication, for example in the reduction of  $SXYZ$  to  $XZ(YZ)$ , the tree representing  $Z$  is duplicated. Even more duplication may be required in conversions involving the fixed-point operator, converting  $YF$  to  $F(YF)$ . Turner proposed replacing these duplicated parts of the trees

with pointers to a single location, so that parts of the tree are not duplicated but stored only once.

This is now the standard method of implementing functional languages.

Another question that arises in the implementation of functional languages is the order in which evaluation steps take place. The Church-Rosser Theorem (see above) tells us that it makes no difference in theory in which order replacements are made in reducing terms, but in practice the order in which replacements are made can make a big difference in efficiency. Furthermore, an order which is efficient in one context may not be efficient in another. For this reason, Landin proposed an order he called «lazy evaluation» (Landin 1965), which has become one of the standard strategies of evaluation.

In lazy evaluation, functions are evaluated symbolically before the values of their arguments are evaluated (unless the values of the arguments are needed for the symbolic evaluation of the function). In the pure lambda-calculus, lazy evaluation means always carrying out the replacement as far left as possible. Thus, lazy evaluation reduces  $KxU$  to  $x$  immediately regardless of what happens to  $U$ ; in fact,  $U$  may not have a normal form.

But lazy evaluation conflicts with strictness. Strictness means that if any argument of a function has no value, then the function has no value.

Thus,  $Kx$  is not strict in ordinary lambda calculus, as the previous example shows. Alan Mycroft (1956–present) has introduced (Mycroft 1981) «strictness analysis», which enables a compiler to combine these two approaches in an efficient manner.

A background to this work on efficient evaluation of functional languages is the work done on various strategies of lambda-reduction. Since the 1960s, work in this area has been done by Gérard Berry (1948–present), Jean-Jacques Lévy (1947–present), J. W. Klop (1945–present), and Jan Bergstra (1951–present), and related work on reduction has been done by J. R. Hindley and Gerd Mitschke (1941–present); see (Barendregt 1984 Part Chapter 3 and III) and the references given there.

For a general discussion of modern implementation techniques, see (Peyton Jones 1987).

Just as the untyped lambda-calculus is a paradigm for a pure functional language, so typed lambda-calculus or lambda-calculus with type assignment is a paradigm for the typing discipline in programming languages in general. This means that questions arising in connection with types in general can be treated in terms of type assignment. This matter was first taken up in computer science by the functional programming community, and, in particular, by Robin Milner (1934–present) in (Milner 1978).

This paper deals with the problem of a program that is essentially the same over any of several types but which, in the older imperative languages must be rewritten for each separate type. For example, a sort routine may be written with essentially the same code except for the types for integers, booleans, and strings. It is clearly desirable to have a method of writing a piece of code that can accept the specific type as an argument. Milner developed his ideas in terms of type assignment to lambda-terms. It is based on a result due originally to Curry (Curry 1969) and Hindley (Hindley 1969) known as the principal type-scheme theorem, which says that (assuming that the typing assumptions are sufficiently well-behaved) every term has a principal type-scheme, which is a type-scheme such that every other type-scheme which can be proved for the given term is obtained by a substitution of types for type variables. This use of type schemes allows a kind of generality over all types, which is known as polymorphism.

When type assignment is considered in connection with functional languages it is often necessary to adopt new primitive types and terms for constructions which can be defined in terms of ordinary lambda-terms. One of these constructions is the cartesian product type, whose terms are ordered pairs. Pairs and their projection functions have been represented in pure lambda-calculus since the 1930s (Curry, Hindley, and Seldin 1972 § 13A3), but they do not always have the most de-

sirable properties with respect to types. In addition, there is a property, the surjective property of pairing, which is desirable in connection with category theory and which says, in effect, that every term is a pair; formally it says that  $\langle \text{fst } X, \text{snd } X \rangle$  converts to  $X$ , where fst and snd are the left and right projections respectively. In pure lambda-calculus, no representation of pairs and projections functions satisfies the surjective pairing property, and (Klop 1980) showed that adding pairs and projection functions with the surjective property causes the Church-Rosser Theorem to fail. Garrel Pottinger (1944–present) showed (Pottinger 1981) that for typed terms, surjective-pairing is compatible with the Church-Rosser theorem. More recently, Klop and Roel C. de Vrijer (1949–present) have shown (Klop and de Vrijer 1989) that certain special cases of the surjective property are consistent with the Church-Rosser theorem in the untyped lambda-calculus; one of the cases is that in which  $X$  is required to be a term in normal form.

Many of the newer functional languages are typed.

A number of functional languages have been introduced over the years; the most important (and most easily used) date from about 1980 or later. (Landin 1964) introduced the SECD machine, and Landin went on to introduce the functional language ISWIM (Landin 1966). David Turner created three pure functional languages: SASL (Turner 1976), KRC (Turner 1981), and Miranda (Turner 1985). A variant of LISP called SCHEME

was introduced by Guy L. Steele, Jr. (1954–present) (Steele and Sussman 1978). Meanwhile, the language HOPE was introduced by R. M. Burstall, D. B. MacQueen (1946–present), and D. T. Sannella (1956–present); see (Burstall, MacQueen, and Sannella 1980). Another important functional language is ML (Milner 1984), which is the meta-language of LCF (an interactive system for reasoning about computable functions). ML includes the basic system of type assignment to lambda-terms and combinatory terms, and includes an algorithm for calculating the types.

Richard Statman (1946–present) has reversed the usual procedure of calculating the types of terms and calculated the terms assigned to particular types. This has led to some interesting results; see (Statman 1980; Statman 1981).

A more recent functional language is HASKELL (named for Haskell B. Curry), which is due to a team headed by Paul Hudak (1952–present) and Philip Wadler (1956–present); see (Hudak and Wadler 1988).

These languages are all intended to be compiled (or interpreted) on a standard computer, whose design has more in common with imperative languages than with functional languages. But some people have been designing hardware to evaluate the expressions of functional languages directly. These machines have all been combinator reducers. Two important projects of this kind are NORMA at Burroughs (Scheevel 1986),

which has since been discontinued, and the CURRY chip at MITRE Corporation (Ramsdell 1986).

Another approach to the relation between lambda-calculus and computers, which is based on considerations from category theory, is due to P.-L. Curien (1953–present); see (Curien 1986).

The connections between lambda-calculus and computers have led to some important extensions of typed lambda calculus and type assignment. One important such extension is the second-order polymorphic typed lambda-calculus, also known as the «second-order lambda-calculus». It was introduced independently by Jean-Yves Girard (1947–present) (Girard 1971; 1972) and John C. Reynolds (1935–present) (Reynolds 1974). Under the formulas-as-types notion, it represents a special kind of second-order logic: propositional logic with a second-order quantifier over propositions. The strong normalization theorem holds for the second-order lambda calculus.

In the second-order lambda-calculus, it is often important to indicate the type a variable is assumed to have. Thus, instead of « $\lambda x.M$ », one writes « $\lambda x : \alpha . M$ ». There are new abstraction and application operators for types (although now it is common to use the same notation as for the old ones), and the introduction and elimination rules for the second-order quantifier are  $M : \alpha \vdash \lambda a.M : (\forall a)\alpha$  (provided that  $a$  is a type variable not free in any assumption) and  $M : (\forall a)\alpha \vdash M\beta : [\beta/a]\alpha$ , where  $\beta$  is any type and  $[\beta/a]\alpha$  is

the result of substituting  $\beta$  for  $a$  in  $\alpha$ . As an example, from  $\lambda x : a . x : a \rightarrow a$ , (which can be proved without assumption), we can use the introduction rule to deduce  $\lambda a . \lambda x : a . x : (\forall a)(a \rightarrow a)$ . Then, for any type  $\beta$ , we can use the elimination rule to deduce  $(\lambda a . \lambda x : a . x)\beta : \beta \rightarrow \beta$ . There is a second kind of  $\lambda\beta$ -reduction that allows us to reduce  $(\lambda a . \lambda x : a . x)\beta$  to  $\lambda x : \beta . x$ .

Another interesting extension of ordinary type assignment is due to Mario Coppo and Mariangiola Dezani-Ciancaglini, and involves a new type constructor, the intersection of two types (Coppo and Dezani-Ciancaglini 1978). In this system, both the normalization theorem (every term with a type has a normal form) and its converse (every term in normal form has a type) hold. However, intersection types do not correspond to conjunctions under the formulas-as-types notion (Hindley 1984). In further work with Patrick Sallé (dates?) and Betty Venneri, they showed that if this system is further extended by a universal type which is a type of every term, then the nature of the types (and in particular how the universal type occurs in them) can be used to characterize terms that have a normal form and terms that have a «head normal form», where a head normal form is a term that cannot be reduced by applying a reduction rule at the head of the term (Coppo and Dezani-Ciancaglini and Venneri 1981; Sallé 1980). The completeness theorem is given in (Hindley 1982). I. Margaria (1948—pre-

sent) has also done work on these types. For more sources see (Hindley and Seldin 1986 p. 223).

The intersection types obey the following rules:  $M : \alpha, M : \beta \vdash M : \alpha \cap \beta$  and  $M : \alpha \cap \beta \vdash M : \alpha$  and  $\alpha \cap \beta \vdash M : \beta$ . The reason that the formulas-as-types notion fails is that the term does not change in these rules, and so these terms cannot codify proofs involving conjunction.

Another extension of ordinary type assignment involves replacing the arrow type with the dependent function type. This type has the property that the type of the value of a function depends on the argument as well as on the type of the argument.

The dependent function type is often written  $(\forall x : A)B$ . Using the syntax of the second-order lambda-calculus to indicate the types of the bound variables, its introduction and elimination rules are  $\Gamma, x : A \vdash M : B \Rightarrow \Gamma \vdash (\forall x : A)(\lambda x : A . B)$ , provided that  $x$  does not occur free in  $\Gamma$ , and  $M : (\forall x : A)B, N : A \vdash MN : [N/x]B$ . The arrow type  $A \rightarrow B$  can be defined as  $(\forall x : A)B$  for a variable  $x$  which does not occur free in  $B$ .

Curry had the idea for this type in 1956, and the idea is mentioned in (Curry, Hindley, and Seldin 1972 pp. 343, 353ff.), where it is called generalized functionality, but the first real work on Curry's version of this idea was done by Jonathan P. Seldin in (Seldin 1979). But by then the dependent function type had already been used in the AUTOMATH pro-

ject (de Bruijn 1970; de Bruijn 1980), which was a system in which proofs could be verified by writing them in the language; using the formulas-as-types idea, if the coded version of the proof was grammatically correct, then the proof was valid. (Other people who have worked on the AUTOMATH project are Roel C. de Vrijer, L. S. van Benthem Jutting (1927–present), R. P. Nederpelt (1942–present), Diederik T. van Daalen (1949–present), I. Zandleven (dates?), and J. Zucker (1942–present); see (de Bruijn 1980) for references). The dependent function type is also important in the type theory of Per Martin-Löf (1942–present) in his intuitionistic theory of types, which is a predicative intuitionistic type theory (Martin-Löf 1975; 1984). The type theory has, in turn, been used as the basis of the Nuprl proof development system (using the formulas-as-types idea) by Robert L. Constable (1942–present) and his associates at Cornell (Constable et al. 1986).

A proof development system differs from an automatic theorem prover in that in it the user selects the result to be proved and the basic strategy for carrying out the proof on an interactive basis, and the computer does the work of carrying out the strategy. At each stage the user selects the next tactic to be applied, where a tactic is, in a sense, a derived rule run backwards.

Because the type theory of Martin-Löf is predicative and the second-order lambda-calculus is not, it is not possible to in-

interpret the latter in the former. An impredicative system similar to Martin-Löf's type theory in which the second-order lambda-calculus can be interpreted is the calculus of constructions of Thierry Coquand (1961–present) (Coquand and Huet 1988), for which the strong normalization theorem has been proved and which has been proposed as the basis of a proof development system similar to Nuprl. Coquand and Gérard Huet (1947–present) are continuing work on the calculus of constructions at INRIA.

Although the lambda-calculus did not lead to the kind of foundation for logic and mathematics for which its founders were searching, it has shown itself to be extremely useful in several areas of logic and computer science.

#### ACKNOWLEDGEMENT

I would like to thank Peter Grogono and Roger Hindley for their helpful comments and suggestions.

#### REFERENCES

- Ackermann, W., Widerspruchsfreier Aufbau der Logik I. Typenfreies System ohne Tertium non datur, «Journal of Symbolic Logic», 1950, XV, pp. 33–57.
- Ackermann, W., Widerspruchsfreier Aufbau einer typenfreien Logik, «Mathematische Zeitschrift», 1952, LV, pp. 364–384 and 1953, LVI, pp. 155–166.
- Aczel, P., Frege structures and the notions of proposition, truth, and set, in Barwise, J., Kiesler, H., J., Kunen,

- K. (a cura di) The Kleene Symposium, Amsterdam, 1980, pp. 31–59.
- Andrews, P. B., A Transfinite Type Theory with Type Variables, Amsterdam, 1965.
- Andrews, P. B., An Introduction to Mathematical Logic and Type Theory: to Truth through Proof, Orlando, Florida, 1986.
- Andrews, P. B., Issar, S., Nesmith, D., Pfenning, F., The TPS Theorem Proving System, in Lusk, E., Overbeek, R. (a cura di), 9th International Conference on Automated Deduction, Berlin, 1988 («Lecture Notes in Computer Science» (Springer-Verlag), 1988, CCCX), pp. 760–761.
- Backus, J., Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, «Communications of the A.C.M.», 1978, XXI, pp. 613–641.
- Barendregt, H., The Lambda Calculus: its Syntax and Semantics, Revised edition, Amsterdam, 1984.
- Böhm, C., The CUCH as a formal and description language, in Steel, T. B. Jr. (a cura di) Formal Language Description Languages for Computer Programming, Amsterdam, 1966, pp. 179–197.
- de Bruijn, N. G., A survey of the project AUTOMATH, in Hindley, J.P., Seldin, J.P. (a cura di), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, London, 1980, pp. 579–606.

- Bunder, M. W., Set Theory Based on Combinatory Logic, Ph.D. Dissertation, Amsterdam, 1969. Published (in five parts) «Notre Dame Journal of Formal Logic», 1970, XI, pp. 467–470; 1973, XIV, pp. 53–54, 341–346; 1974, XV, pp. 25–34, 192–206.
- Bunder, M. W., Predicate calculus of arbitrarily high finite order, «Archiv für mathematische Logik», 1983, XXIII, pp. 1–10.
- Bunder, M. W., A one axiom set theory based on higher order predicate calculus, «Archiv für mathematische Logik», 1983, XXIII, pp. 99–107.
- Bunder, M. W., Set theory in predicate calculus with equality, «Archiv für mathematische Logik», 1983, XXIII, pp. 109–113.
- Bunder, M. W., A weak absolute consistency proof for some systems of illative combinatory logic, «Journal of Symbolic Logic», 1983, XLVII, pp. 771–776.
- Bunder, M. W., Category theory based on combinatory logic, «Archiv für mathematische Logik», 1984, XXIV, pp. 1–15.
- Bunder, M. W., Some generalizations to two systems of set theory based on combinatory logic, «Archiv für mathematische Logik», 1986/7, XXVI, pp. 5–12.
- Burstall, R. M., Proving properties of programs by structural induction, «Computer Journal», 1969, XII, pp. 41–48.

- Burstall, R. M., MacQueen, D. B., Sannella, D. T., HOPE: an experimental applicative language, in Conference Record of the 1980 LISP Conference, 1980, pp. 136–143.
- Church, A., A set of postulates for the foundation of logic, «Annals of Mathematics» Second Series, 1932, XXXIII, pp. 346–366 and 1933, XXXIV, pp. 839–864.
- Church, A., An unsolvable problem of elementary number theory, «American Journal of Mathematics», 1936, LVIII, pp. 345–363.
- Church, A., A note on the *Entscheidungsproblem*, «Journal of Symbolic Logic», 1936, I, pp. 40–41.
- Church, A., Correction to A note on the *Entscheidungsproblem*, «Journal of Symbolic Logic», 1936, I, pp. 101–102.
- Church, A., A formulation of the simple theory of types, «Journal of Symbolic Logic», 1940, V, pp. 56–68.
- Church, A., The Calculi of Lambda Conversion, Princeton, New Jersey, 1941.
- Church, A., A formulation of the logic of sense and denotation, in Henle, P., Kallen, H. M., Langer, S. (a cura di), Structure, Method and Meaning, New York, 1951, pp. 3–24.
- Church, A., The need for abstract entities in semantic analysis, «Proceedings of the American Academy of Arts and Sciences», 1951–54, LXXX, pp. 100–112.

- Church, A., Kleene, S. C., Formal definitions in the theory of ordinal numbers, «Fundamenta Mathematicae», 1937, XXVIII, pp. 11–21.
- Church, A., Rosser, J. B., Some properties of conversion, «Transactions of the American Mathematical Society», 1936, XXXIX, pp. 472–482.
- Constable, R. L., Allen, S. F., Bromley, H. M., Cleaveland, W. R., Cremer, J. F., Harper, R. W., Howe, D. J., Knoblock, T. B., Mendler, N. P., Panangaden, P., Sasaki, J. T., Smith, S. F., Implementing Mathematics with the Nuprl Proof Development System, Englewood Cliffs, New Jersey, 1986.
- Coppo, M., Dezani-Ciancaglini, M., A new type-assignment for  $\lambda$ -terms, «Archiv für mathematische Logik», 1978, XIX, pp. 139–156.
- Coppo, M., Dezani-Ciancaglini, M., Venneri, B., Functional characters of solvable terms, «Zeitschrift für mathematische Logik und Grundlagen der Mathematik», 1981, XXVII, pp. 45–58.
- Coquand, T., Huet, G., The calculus of constructions, «Information and Computation», 1988, LXXVI, pp. 95–120.
- Curien, P-L., Categorical Combinators, Sequential Algorithms and Functional Programming, London, 1986.
- Curry, H. B., An analysis of logical substitution, «American Journal of Mathematics» 1929, LI, pp. 363–384.

- Curry, H. B., Grundlagen der kombinatorischen Logik, «American Journal of Mathematics», 1930, LII, pp. 509–536, 789–834.
- Curry, H. B., The universal quantifier in combinatory logic, «Annals of Mathematics» Second Series, 1931, XXXII, pp. 154–180.
- Curry, H. B., Some additions to the theory of combinators, «American Journal of Mathematics», 1932, LIV, pp. 551–558.
- Curry, H. B., Apparent variables from the standpoint of combinatory logic, «Annals of Mathematics» Second Series, 1933, XXXIV, pp. 381–404.
- Curry, H. B., Some properties of equality and implication in combinatory logic, «Annals of Mathematics» Second Series, 1934, XXXV, pp. 849–860.
- Curry, H. B., Functionality in combinatory logic, «Proceedings of the National Academy of Sciences (U.S.A.)», 1936, XX, pp. 584–590.
- Curry, H. B., Foundations of the theory of abstract sets from the standpoint of combinatory logic (Abstract), «Bulletin of the American Mathematical Society», 1934, XL, p. 654.
- Curry, H. B., First properties of functionality in combinatory logic, «Tôhoku Mathematical Journal», 1936, XLI, Part II, pp. 371–401.

- Curry, H. B., A revision of the fundamental rules of combinatory logic, «Journal of Symbolic Logic», 1941, VI, pp. 41–53.
- Curry, H. B., The consistency and completeness of the theory of combinators, «Journal of Symbolic Logic», 1941, VI, pp. 54–61.
- Curry, H. B., The paradox of Kleene and Rosser, «Transactions of the American Mathematical Society», 1941, L, pp. 454–516.
- Curry, H. B., The combinatory foundations of mathematical logic, «Journal of Symbolic Logic», 1942, VII, pp. 49–64.
- Curry, H. B., The inconsistency of certain formal logics, «Journal of Symbolic Logic», 1942, VII, pp. 115–117.
- Curry, H. B., Some advances in the combinatory theory of quantification, «Proceedings of the National Academy of Sciences (U.S.A.)», 1942, XXVIII, pp. 564–569.
- Curry, H. B., A simplification of the theory of combinators, «Synthese», 1949, VII, pp. 391–399.
- Curry, H. B., The logic of program composition, in Applications Scientifiques de la Logique Mathématique: Actes du 2<sup>e</sup> Colloque International de Logique Mathématique, Paris – 25–30 Août 1952, Institut Henri Poincaré, Paris, 1954, pp. 97–102.

- Curry, H. B., The inconsistency of the full theory of functionality (Abstract), «Journal of Symbolic Logic», 1955, XX, p. 91.
- Curry, H. B., The consistency of the theory of functionality (Abstract), «Journal of Symbolic Logic», 1956, XXI, p. 110.
- Curry, H. B., The deduction theorem in the combinatory theory of restricted generality, «Logique et Analyse», 1960, III, pp. 15–39.
- Curry, H. B., Basic verifiability in the combinatory theory of restricted generality, in Bar-Hillel, Y., Poznanski, E. I. J., Rabin, M. O., Robinson, A. (a cura di) Essays on the Foundation of Mathematics, Jerusalem, 1961, pp. 165–189.
- Curry, H. B., Modified basic functionality in combinatory logic, «Dialectica», 1969, XXIII, pp. 83–92.
- Curry, H. B., The consistency of a system of combinatory restricted generality, «Journal of Symbolic Logic», 1973, XXXVII, pp. 489–492.
- Curry, H. B., Feys, R., Combinatory Logic, Vol. I, Amsterdam, 1958.
- Curry, H. B., Hindley, J. R., Seldin, J. P., Combinatory Logic, Vol. II, Amsterdam, 1972.
- Curry, H. B., Hindley, J. R., Seldin, J. P., Beta strong reduction in combinatory logic: preliminary report

- (Abstract), «Journal of Symbolic Logic», 1984, XLIX, p. 688.
- Fitch, F. B., A system of formal logic without an analogue to the Curry W operator, «Journal of Symbolic Logic», 1936, I, pp. 92–100.
- Fitch, F. B., The system  $CA$  of combinatory logic, «Journal of Symbolic Logic», 1963, XXVIII, pp. 87–97.
- Fitch, F. B., Elements of Combinatory Logic, New Haven, Connecticut, 1974.
- Fitch, F. B., A consistent combinatory logic with an inverse to equality, «Journal of Symbolic Logic», 1980, XLV, pp. 529–543.
- Fitch, F. B., An extension of a system of combinatory logic, in Hindley, J.P., Seldin, J.P. (a cura di), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, London, 1980, pp. 125–140.
- Frege, G., Funktion und Begriff, in Kleine Schriften, Darmstadt 1967, pp. 124–142. (Originally Jena 1891.)
- Gandy, R., An early proof of normalization by A.M. Turing, in Hindley, J.P., Seldin, J.P. (a cura di), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, London, 1980, pp. 453–455.
- Gentzen, G., Untersuchungen über das logische Schliessen, «Mathematische Zeitschrift», 1934, XXXIX, pp. 176–210.
- Gentzen, G., Die Widerspruchsfreiheit der reinen Zahlentheorie, «Mathematische Annalen», 1936, CXII, pp. 493–565.

- Girard, J-Y., Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types, in Fenstad, J. E. (a cura di) Proceedings of the Second Scandinavian Logic Symposium, Amsterdam, 1971, pp. 63–92.
- Girard, J-Y., Interprétation Fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur, Thèse de Doctorat d'État, Paris, 1972.
- Gödel, K., Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, «Monatshefte für Mathematik und Physik», 1931, XXXVIII, 173–198.
- Gödel, K., Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes, «Dialectica», 1958, XII, pp. 280–287.
- Henkin, L., Completeness in the theory of types, «Journal of Symbolic Logic», 1950, XV, pp. 81–91.
- Henkin, L., A theory of propositional types, «Fundamenta Mathematicae», 1963, LIII, pp. 323–344.
- Hindley, J. R., The principal type scheme of an object in combinatory logic, «Transactions of the American Mathematical Society», 1969, CXLVI, pp. 29–60.
- Hindley, J. R., The simple semantics for Coppo-Dezani-Sallé types, in Dezani-Ciancaglini, M., Montanari, U. (a cura di) International Symposium on Programming: 5th Colloquium, Turin, April 6–8, 1982, Proceedings, Berlin, 1982

- («Lecture Notes in Computer Science» (Springer-Verlag), 1982, CXXXVII) pp. 212–226 pp.
- Hindley, J. R., The completeness theorem for typing  $\lambda$ -terms, «Theoretical Computer Science», 1983, XXII, pp. 1–17.
- Hindley, J. R., Curry's type-rules are complete with respect to the F-semantics too, «Theoretical computer Science», 1983, XXII, pp. 127–133.
- Hindley, J. R., Coppo-Dezani types do not correspond to propositional logic, «Theoretical Computer Science», 1984, XXVIII, pp. 235–236.
- Hindley, J. R., Seldin, J. P., Introduction to Combinators and  $\lambda$ -Calculus, Cambridge, 1986.
- Hudak, P., Wadler, P. (a cura di) Report on the Functional Programming Language HASKELL, Hartford, Connecticut, 1988.
- Howard, W. A., The formulae-as-types notion of construction, in Hindley, J.P., Seldin, J.P. (a cura di), To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, London, 1980, pp. 479–490 (originally written in 1969 and privately circulated).
- Kleene, S. C., Proof by cases in formal logic, «Annals of Mathematics» Second Series, 1934, XXXV, pp. 529–544.
- Kleene, S. C., A theory of positive integers in formal logic, «American Journal of Mathematics», 1935, LVII, pp. 153–173, 219–244.

- Kleene, S. C.,  $\lambda$ -definability and recursiveness, «Duke Mathematical Journal», 1936, II, pp. 340–353.
- Kleene, S. C., General recursive functions of natural numbers, «Mathematische Annalen», 1936, CXII, pp. 727–742.
- Kleene, S. C., On notation for ordinal numbers, «Journal of Symbolic Logic», 1938, III, pp. 150–155.
- Kleene, S. C., Rosser, J. B., The inconsistency of certain formal logics, «Annals of Mathematics» Second Series, 1935, XXXVI, pp. 630–636.
- Klop, J. W., Combinatory Reduction Systems, Dissertation, Utrecht, 1980.
- Klop, J. W. , de Vrijer, R. C., Unique normal forms for lambda calculus with surjective pairing, «Information and Computation», 1989, LXXX, pp. 97–113.
- Lambek, J., Deductive systems and categories I, «Journal of Mathematical Systems Theory», 1968, II, pp., 278–318.
- Lambek, J., Deductive systems and categories II, in Hilton, P. (a cura di) Category Theory, Homology Theory and their Applications I, Berlin, 1969 («Lecture Notes in Mathematics» (Springer-Verlag), 1969, LXXXVI) pp. 76–122.
- Lambek, J., Deductive systems and categories III, in Lawvere, F. W. (a cura di) Toposes, Algebraic Geometry and Logic, Berlin, 1972 («Lecture Notes in Mathematics» (Springer-Verlag), 1972, CCLVIV) pp. 57–82.

- Lambek, J., Functional completeness of cartesian categories, «Annals of Mathematical Logic», 1974, VI, pp. 259–292.
- Lambek, J., Scott, P. J., Introduction to Higher Order Categorical Logic, Cambridge, 1986.
- Landin, P., The mechanical evaluation of expressions, «Journal of the A.C.M.», 1964, VI, pp. 308–320.
- Landin, P., A correspondence between Algol 60 and Church's lambda calculus, «Communications of the A.C.M.», 1965, VIII, pp. 89–101, 158–165.
- Landin, P., The next 700 programming languages, «Communications of the A.C.M.», 1966, IX, pp. 157–164.
- McCarthy, J., Recursive functions of symbolic expressions and their computation by machine, Part I, «Communications of the A. C. M.», 1960, III, pp. 184–195.
- McCarthy, J., A basis for a mathematical theory of computation, in Braffort, P., Hirschberg, D. (a cura di) Computer Programming and Formal Systems, Amsterdam, 1963, pp. 33–70.
- McCarthy, J., History of LISP, in Wexelblat, R. (a cura di) History of Programming Languages, London, 1981, pp. 173–196.
- Martin-Löf, P., An intuitionistic theory of types: predicative part, in Rose, H. E. (a cura di) Logic Colloquium '73, Amsterdam, 1975, pp. 73–118.
- Martin-Löf, P., Intuitionistic Type Theory, Naples, 1984.

- Meyer, A. R., What is a model of the lambda calculus?,  
«Information and Control», 1982, LII, pp. 87–122.
- Mezghiche, M., Une nouvelle  $C\beta$ -réduction dans la logique combinatoire, «Theoretical Computer Science», 1984, XXXI, pp. 151–165.
- Mezghiche, M., On pseudo- $c\beta$ normal form in combinatory logic, «Theoretical Computer Science», 1989, LXVI, pp. 323–334.
- Milner, R., A theory of type polymorphism and programming, «Journal of Computer and System Sciences», 1978, XVII, pp. 348–375.
- Milner, R., A proposal for standard ML, in Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, 1984, pp. 184–197.
- Mycroft, A., Abstract Interpretation and Optimising Transformations for Applicative Programs, Dissertation, University of Edinburgh, 1981.
- Peyton Jones, S., The Implementation of Functional Programming Languages, Englewood Cliffs, New Jersey, 1987.
- Pottinger, G., The Church-Rosser theorem for typed  $\lambda$ -calculus with surjective pairing, «Notre Dame Journal of Formal Logic», 1981, XXII, pp. 264–268.
- Ramsdell, J. D., The CURRY chip, in Proceedings of the 1986 ACM Conference on LISP and Functional Programming, 1986, pp. 122–131.

- Reynolds, J. C., Towards a theory of type structure, in Programming Symposium. Proceeding, Colloque sur la Programmation, Paris, 1974, Berlin, 1974 («Lecture Notes in Computer Science» (Springer-Verlag), 1974, XIX) pp. 408–425.
- Rezus, A., Lambda-Conversion and Logic, Dissertation, Utrecht, 1981.
- Rosser, J. B., A mathematical logic without variables, «Annals of Mathematics» Second series, 1935, XXXVI, pp. 127–150 and «Duke Mathematical Journal», 1935, I, pp. 328–355.
- Rosser, J. B., New sets of postulates for combinatory logics, «Journal of Symbolic Logic», 1942, VII, pp. 18–27.
- Rosser, J. B., Highlights of the History of the Lambda-Calculus, «Annals of the History of Computing», 1984, VI, pp. 337–349.
- Russell, B., Mathematical logic as based on the theory of types, «American Journal of Mathematics» 1908, XXX, pp. 222–262.
- Sallé, P., Une généralisation de la théorie des types en  $\lambda$ -calcul, «R.A.I.R.O. Informatique théorique/Theoretical Informatics», 1980, XIV, pp. 143–167, 301–314.
- Scheevel, M., NORMA: a graph reduction processor, in Proceedings of the 1986 ACM Conference on LISP and Functional Programming, 1986, pp. 212–219.
- Schönfinkel, M., Über die Bausteine der mathematischen Logik, «Mathematische Annalen» 1924, XCII, pp. 305–316.

- Schwichtenberg, Helmut, Definierbare Funktionen im  $\lambda$ -Kalkül mit Typen, «Archiv für mathematische Logik», 1975-6, XVII, pp. 113-114.
- Scott, D. S., Combinators and Classes, in Böhm, C. (a cura di)  $\lambda$ -Calculus and Computer Science Theory, Berlin, 1975 («Lecture Notes in Computer Science» (Springer-Verlag), 1975, XXXVII) pp. 1-26.
- Seldin, J. P., Studies in Illative Combinatory Logic, Ph.D. Dissertation, Amsterdam, 1968.
- Seldin, J. P., Progress report on generalized functionality, «Annals of Mathematical Logic», 1979, XVII, pp. 29-59.
- Statman, R., On the existence of closed terms in the typed  $\lambda$ -calculus I, in Hindley, J. R., Seldin, J. P., To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, London, 1980, pp. 511-534.
- Statman, R., On the existence of closed terms in the typed  $\lambda$ -calculus II, «Theoretical Computer Science», 1981, XV, pp. 329-338.
- Steele, G. L. Jr.; Sussman, Gerald J., Revised Report on SCHEME, a Dialect of LISP, Cambridge, Massachusetts, 1978.
- Strachey, C., Towards a formal semantics, in Steel, T. B. Jr. (a cura di) Formal Language Description Languages for Computer Programming, Amsterdam, 1966, pp. 198-220.

- Stoy, J. E., Denotational Semantics: the Scott Strachey Approach to Programming Language Theory, Cambridge, Massachusetts, 1977.
- Tait, W. W., Intensional interpretations of functionals of finite type I, «Journal of Symbolic Logic», 1967, XXXII, pp. 198–212.
- Turing, A., On computable numbers, with an application to the Entscheidungsproblem, «Proceedings of the London Mathematical Society» 1936, XLII, 230–265. A correction, «Ibid.», 1937, XLIII, pp. 544–546.
- Turing, A., Computability and  $\lambda$ -definability, «Journal of Symbolic Logic», 1937, II, pp. 153–163.
- Turner, D., The SASL Language Manual, University of St. Andrews, Scotland, 1976.
- Turner, D., A new implementation technique for applicative languages, «Software: Practice and Experience», 1979, IX, pp. 31–49.
- Turner, D., KRC Language Manual, University of Kent, 1981.
- Turner, D., Miranda: a non-strict functional language with polymorphic types, in Proceedings of the Second International Conference on Functional Programming Languages and Computer Architecture, Berlin, 1985 («Lecture Notes in Computer Science» (Springer-Verlag), 1985, CCCI).
- Whitehead, A. N., Russell, B., Principia Mathematica, Cambridge, 1910–1913, 3 vols.