THE BULLETIN OF SYMBOLIC LOGIC
Volume 3, Number 2, June 1997

THE IMPACT OF THE LAMBDA CALCULUS IN LOGIC AND COMPUTER SCIENCE

HENK BARENDREGT

Abstract. One of the most important contributions of A. Church to logic is his invention of the lambda calculus. We present the genesis of this theory and its two major areas of application: the representation of computations and the resulting functional programming languages on the one hand and the representation of reasoning and the resulting systems of computer mathematics on the other hand.

Acknowledgment. The following persons provided help in various ways. Erik Barendsen, Jon Barwise, Johan van Benthem, Andreas Blass, Olivier Danvy, Wil Dekkers, Marko van Eekelen, Sol Feferman, Andrzej Filinski, Twan Laan, Jan Kuper, Pierre Lescanne, Hans Mooij, Robert Maron, Rinus Plasmeijer, Randy Pollack, Kristoffer Rose, Richard Shore, Rick Statman and Simon Thompson.

§1. Introduction. This paper is written to honor Church's great invention: the lambda calculus. The best way to do this—I think—is to give a description of its genesis (§2) and its impact on two areas of mathematical logic: the representation of computations (§3) and of reasoning (§4). In both cases technological applications have emerged.

The very notion of computability was first formalized in terms of definability on numerals represented in the lambda calculus. Church's Thesis, stating that this is the correct formalization of the notion of computability, has for more than 60 years never seriously been challenged. One of the recent advances in lambda calculus is that computations on other data types, like trees and syntactic structures (e.g., for parsing), can be done by representing these data types directly as lambda terms and not via a coding as Gödel numbers that are then represented as numerals. This resulted in a much more efficient representation of functions defined on these data types.

Received September 12, 1996; revised February 28, 1997.
Partial support came from the European HCM project Typed lambda calculus (CHRXCT-92-0046), the Esprit Working Group Types (21900) and the Dutch NWO project WINST (612-316-607).

1079-8986/97/0302-0003/$4.50

The notion of lambda definability is conceptually the basis for the discipline of functional programming. Recent progress in this area has been the construction of very efficient compilers for functional languages and the capturing of interactive programs (like e.g., text editors) within the functional programming paradigm.

As to the representation of proofs, one of Church's original goals had been to construct a formal system for the foundations of mathematics by having a system of functions together with a set of logical notions. When the resulting system turned out to be inconsistent, this program was abandoned by him. Church then separated out the consistent subsystem that is now called the lambda calculus and concentrated on computability[1]. It turned out later that there are nevertheless consistent ways to represent logical notions in (typed and untyped) lambda calculus so that a foundation for mathematics is obtained. Some of the resulting systems are used in recently developed systems for computer mathematics, i.e., programs for the interactive development and automated verification of mathematical proofs.

We restrict attention to applications of the lambda calculus to the fields of mathematical logic and computer science. Other applications like several forms of grammars studied in linguistics (e.g., Montague (see [45]) and categorial grammars (see [17])) are not treated in this paper.

We end this introduction by telling what seems to be the story how the letter 'λ' was chosen to denote function abstraction. In [100] Principia Mathematica the notation for the function f with $f(x) = 2x + 1$ is $2\hat{x} + 1$. Church originally intended to use the notation $\hat{x}.2x+1$. The typesetter could not position the hat on top of the x and placed it in front of it, resulting in

$$\wedge x.2x + 1.$$

Then another typesetter changed it into $\lambda x.2x + 1$.

Preliminaries. This short subsection with preliminaries is given for readers not familiar with the lambda calculus. For more information see e.g., [6, Chapters 2, 3 and 6], or [8, Sections 2 (untyped lambda calculus) and 3 (simply typed lambda calculus)]. Topics outside these chapters or sections needed in this paper will be explicitly mentioned.

Untyped lambda calculus.

Definition 1.1. The sets of variables and terms of the lambda calculus are defined by the following abstract syntax. (This means that no mention is made of necessary partentheses in order to warrant unique readability; one

[1]Church had been considerably helped by his students in the early development of the lambda calculus, notably by Kleene, see [70] and [99]. Other important influences came from [36] and [37].

thinks about trees instead of strings being generated.)

$$\begin{array}{rcl}\texttt{var} & = & a \mid \texttt{var}' \\ \texttt{term} & = & \texttt{var} \mid \texttt{term}\ \texttt{term} \mid \lambda\ \texttt{var}\ \texttt{term}.\end{array}$$

The syntactic category var is for the collection of variables. Examples of variables are a, a', a''. The letters $x, y, z, \ldots$ range over arbitrary variables. The syntactic category term is for the collection of lambda terms, notation Λ.

NOTATION. (i) $MN_1 \ldots N_k$ stands for $(..((MN_1)N_2) \ldots N_k)$.
(ii) Dually, $\lambda x_1 \ldots x_k.M$ stands for $(\lambda x_1(\lambda x_2(\ldots(\lambda x_k(M))..)))$.

Examples of lambda terms are $x, xy, \lambda x.xy, z(\lambda x.xy), \lambda zy.z(\lambda x.xy)$ and $(\lambda zy.z(\lambda x.xy))(ww)yx$.

A term of the form MN is called an *application*, with the intended interpretation 'the function M applied to the argument N'; a term of the form $\lambda x.M$ is called an *abstraction*, with the intended interpretation 'the function that assigns to x the value M'. In this interpretation the notion of function is to be taken intensional, i.e., as an algorithm. [103] succeeded to give lambda calculus also an extensional interpretation by interpreting lambda terms as (continuous) functions on some topological space D having its space of continuous functions $[D \to D]$ as a retract.

In a lambda term like $\lambda xy.xz$ the variable x is said to occur as a *bound variable* and z occurs as a *free variable*. In $z(\lambda z.z)$ the variable occurs both as free (the first occurrence) and as bound (the second occurrence) variable. The statement $M \equiv N$ stands for syntactic equality modulo a renaming of the bound variables. E.g., $\lambda x.x \equiv \lambda y.y$ or $x(\lambda x.x) \equiv x(\lambda y.y)$, but $\lambda x.xy \not\equiv \lambda y.yy$ because the free occurrence of y in the LHS becomes bound in the RHS.

The lambda calculus is the study of the set Λ modulo so called β-convertibility which is the least congruence relation $=_\beta$ axiomatized by

$$(\lambda x.M)N =_\beta M[x{:=}N].$$

Here $M[x{:=}N]$ stands for the result of substituting N for the free variables of M. In this notation the free variables of N are not allowed to become bound after substitution; for example $(\lambda y.x)[x{:=}yy] \not\equiv (\lambda y.yy)$. By changing the names of bound variables one may obtain

$$(\lambda y.x)[x{:=}yy] \equiv (\lambda z.x)[x{:=}yy] \equiv \lambda z.yy.$$

The notion of β-convertibility is an equivalence relation compatible with the syntactic operations of application and abstraction. That is,

$$C[(\lambda x.M)N] =_\beta C[M[x{:=}N]]$$

holds for arbitrary contexts $C[\ ]$.

The notion of β-reduction is the least compatible reflexive and transitive relation $\twoheadrightarrow_\beta$ axiomatized by

$$(\lambda x.M)N \twoheadrightarrow_\beta M[x:=N].$$

The difference with β-conversion is that one has e.g., $a =_\beta (\lambda x.x)a$, but $a \not\twoheadrightarrow_\beta (\lambda x.x)a$: there is a direction involved in reduction, while conversion is bidirectional.

The reason for the notational convention introduced above can be understood by realizing that e.g.,

$$(\lambda xyz.x(yz)y)XYZ \twoheadrightarrow_\beta X(YZ)Y.$$

A term $M \in \Lambda$ is called *in β-normal form* (β-*nf*) if M has no part of the form $(\lambda x.M)N$. Such part is called a β-*redex*. A term M is said to *have a β-normal form N* if N is in β-normal form and $M =_\beta N$.

THEOREM 1.2 (Church-Rosser theorem). *Let $M, N \in \Lambda$. Then*

$$M =_\beta N \iff \exists Z\, [M \twoheadrightarrow_\beta Z \,\&\, N \twoheadrightarrow_\beta Z].$$

It follows from the Church-Rosser theorem that a term can have at most one β-normal form. Indeed, if M has M' and M'' as β-nf's, then $M' =_\beta M''$ and so $M' \twoheadrightarrow_\beta Z\ {}_\beta\!\twoheadleftarrow M''$. But since M' and M'' are in β-nf, there are no redexes to contract. Therefore $M' \equiv Z \equiv M''$.

Simply typed lambda calculus. Simple types are defined by the abstract syntax

$$\begin{aligned} \texttt{tvar} &= \alpha \mid \texttt{tvar}' \\ \texttt{type} &= \texttt{tvar} \mid \texttt{type} \to \texttt{type}. \end{aligned}$$

We use $\alpha, \beta, \gamma, \ldots$ for type variables and $A, B, C, \ldots$ for types. The set of types is denoted by $\mathbb{T}$. A statement is of the form $M : A$ with $M \in \Lambda$ and $A \in \mathbb{T}$; M is called the subject of the statement. A *basis* is a set of statements with only variables as subjects. $\Gamma, \Delta, \ldots$ range over bases. (For more complicated versions of typed lambda calculus, a basis needs to be ordered and then is called a *context*. This is unfortunately a different notion with the same name as the notion 'context' defined earlier, but that is how it is.)

DEFINITION 1.3. We say that from basis Γ we can prove $M : A$, notation $\Gamma \vdash M : A$, if it can be derived from the following production system.

$$\begin{aligned} (x : A) \in \Gamma &\Rightarrow \Gamma \vdash x : A; \\ \Gamma \vdash M : (A \to B),\ \Gamma \vdash N : A &\Rightarrow \Gamma \vdash (MN) : B; \\ \Gamma, x : A \vdash M : B &\Rightarrow \Gamma \vdash (\lambda x.M) : (A \to B). \end{aligned}$$

EXAMPLE 1.4. (i) $x : (A \to A \to B),\ y : A \vdash xyy : B$.
(ii) $\vdash \lambda xy.xyy : (A \to A \to B) \to (A \to B)$.

This version of the simply typed lambda calculus has implicit types at each abstraction λx and is studied by [37]. In [29] a variant with explicit types at abstractions is introduced. In this theory the rule for introducing abstractions is

$$\Gamma, x : A \vdash M : B \Rightarrow \Gamma \vdash (\lambda x{:}A.M) : (A \to B).$$

An essential difference between the two approaches is that in the explicit case the unique type of a term always can be found easily. In the implicit case types are not unique. For the simply typed lambda calculus the types can be reconstructed even in the implicit case, but for more complicated systems this is not the case.

Inductive types and recursion. Because inductive types are convenient to represent data, both in theories and in programs, some type systems allow the axiomatic introduction of so-called *inductive types*. The following is a simple example.

$$\mathtt{nat} ::= \mathtt{zero} \mid \mathtt{succ}\ \mathtt{nat}.$$

Given this definition one has (axiomatically) $\vdash \mathtt{zero} : \mathtt{nat}$, $\vdash \mathtt{succ} : \mathtt{nat} \to \mathtt{nat}$ and $\vdash \mathtt{succ}(\mathtt{succ}\ \mathtt{zero}) : \mathtt{nat}$. Inductive types come with natural primitive recursive operators. For example, given a type A and assuming $a : A, b : \mathtt{nat} \to A \to A$, we may define $F : \mathtt{nat} \to A$ as follows.

$$\begin{aligned} F\ \mathtt{zero} &\to_\iota a; \\ F\ (\mathtt{succ}\ x) &\to_\iota b\ x\ (F\ x). \end{aligned}$$

This F depends uniformly on a, b. To make this dependence explicit, we write $F \equiv R\ a\ b$ and postulate the following.

$$\begin{aligned} R\ a\ b\ \mathtt{zero} &\to_\iota a; \\ R\ a\ b\ (\mathtt{succ}\ x) &\to_\iota b\ x\ (R\ a\ b\ x). \end{aligned}$$

With this operator one can represent primitive recursive functions. Because of the presence of higher types one can even represent the Ackermann function using R.

§2. Formalizing the notion 'computable'. Church introduced a formal theory, let us call it $\mathcal{T}$, based on the notion of function. This system was intended to be a foundation of mathematics. Predicates were represented as characteristic functions. There were many axioms to deal with logical notions. The system $\mathcal{T}$ turned out to be inconsistent, as was shown by Church's students [71] using a *tour de force* argument involving all the techniques needed to prove Gödel's incompleteness theorem[2]. Then [28] isolated the (untyped) lambda calculus from the system $\mathcal{T}$ by deleting the part dealing with logic and keeping the essence of the part dealing with functions. This system was proved consistent by [31], who showed the confluence of β-reduction. Curry, who also wanted to build a foundation for mathematics based on functions (in his case in the form of combinators that do not mention free or bound variables), found a paradox for a system with a similar aim as $\mathcal{T}$, that is very easy to derive, see e.g., [6, Appendix B[3]].

Church introduced the notion of lambda definability for functions $f : \mathbb{N}^k \to \mathbb{N}$ in order to capture the notion of computability[4]. At first only very elementary functions like addition and multiplication were proved to be lambda definable. Even for a function as simple as the predecessor ($\mathtt{pred}(0) = 0$, $\mathtt{pred}(n + 1) = n$) lambda definability remained an open problem for a while. From our present knowledge it is tempting to explain this as follows. Although the lambda calculus was conceived as an untyped theory, typeable terms are more intuitive. Now the functions addition and multiplication are definable by typeable terms, while [101] and [108] have characterized the lambda definable functions in the (simply) typed lambda calculus and the predecessor is not among them. Be this as it may, Kleene did find a way to lambda define the predecessor function in the untyped lambda calculus, by using an appropriate data type (pairs of integers) as auxiliary device. In [69], he described how he found the solution while being anesthetized by laughing gas (N_2O) for the removal of four wisdom teeth. After Kleene showed the solution to his teacher, Church remarked something like: "But then all intuitively computable functions must be lambda definable. In fact, lambda definability must coincide with intuitive computability." Many years later—it was at the occasion of Robin Gandy's 70-th birthday, I believe—I heard Kleene say: "I would like to be able to say that, at the moment of discovering how to lambda define the predecessor function, I got the idea of Church's Thesis. But I did not, Church did." Later, in [67], he gave some important evidence for Church's Thesis by showing that the lambda definable functions coincide with the μ-recursive ones.

[2]Gödel just had given a series of lectures in Princeton at which Kleene and Rosser were present.

[3]Consistent theories based on functions for the foundations of mathematics have been described by [89] (simplified by [98]). With a similar aim are the theories in [53] and [75]. In all these theories the paradoxes have been avoided by having a partial application. [43], [44] and [16] also discuss formal theories with partial application; they aim at constructive foundations and come close to lambda calculus (partial combinatory algebras).

[4]I remember a story stating that Church started to work on the problem of trying to show that the sequence of Betti numbers for a given algebraic variety is computable. He did not succeed in this enterprise, but came up with the proposal to capture the notion of intuitive computability. I have not been able to verify this story. Readers who can confirm or refute it are kindly requested to inform the author.

Independently of Church, an alternative formalization (in terms of (Turing) machines) of the notion 'computable' was given in [113]. In [114] it was proved that the notions of lambda definability and Turing computability are equivalent, thereby enlarging the credibility of Church's Thesis.

Church's Thesis is plausible but cannot be proved, nor even stated in (classical) mathematical terms, since it refers to the undefined notion of intuitive computability. On the other hand, Church's Thesis can be refuted. If ever a function will be found that is intuitively computable but (demonstrably) not lambda definable, then Church's Thesis is false. For more than 60 years this has not happened. This failure to find a counterexample is given as an argument in favor of Church's Thesis. I think that it is fair to say that most logicians do believe Church's Thesis. One may wonder why doubting Church's Thesis is not a completely academic question. This becomes clear by realizing that [106] had introduced the class of primitive recursive functions that for some time was thought to coincide with that of the intuitively computable ones. But then [2] showed that there is a function that is intuitively computable but not primitive recursive. See also the paper of [46] for arguments in favor of Church's Thesis and [73, 74] for ones casting some doubts.

Church's Thesis is actually used for negative computability results: if a function is shown to be not lambda definable (or Turing computable) then, by Church's Thesis, one can state that it is not intuitively computable. Church and Turing gave examples of undecidable predicates, i.e., ones with non-computable characteristic functions: the questions whether a lambda term has a normal form (the normalization problem) and whether a machine with program p and input x terminates (the halting problem), respectively. Both concluded that provability in arithmetic is undecidable. In fact, the undecidability of many mathematical problems has been established by translating the halting problem into a given problem. A famous example is [82] result that Hilbert's tenth problem[5] is unsolvable.

Finally it is worth mentioning that in intuitionistic mathematics, say in Heyting's arithmetic HA, one can precisely formulate Church's Thesis as a formal statement; this in contrast to the situation in the classical theory. This statement is called CT and is

$$\begin{aligned}\forall x[P(x) \vee \neg P(x)] \;\Rightarrow\; & \exists e \forall x[[P(x) \leftrightarrow \phi_e(x) = 1] \\ & \& \, [\phi_e(x) = 0 \vee \phi_e(x) = 1]],\end{aligned}$$

where $\phi_e(x) = y \iff \exists z[T(e, x, z) \,\&\, U(z) = y]$ states that the e-th partial recursive function with input x terminates with y as value (T is Kleene's computation predicate and U is the value extracting function, see [68]). In this form CT states that if P is a decidable predicate (i.e., the excluded middle

[5]"Is it decidable whether a given Diophantine equation has a solution in the integers?"

holds for P), then P has a recursive characteristic function. See [112] for formal consequences, models, counter-models and an extension of CT.

§**3. Computing.** Lambda calculi are prototype programming languages. As is the case with imperative programming languages, where several examples are untyped (machine code, assembler, Basic) and several are typed (Algol-68, Pascal), systems of lambda calculi exist in untyped and typed versions. There are also other differences in the various lambda calculi. The lambda calculus introduced in [28] is the untyped $\lambda\mathsf{I}$-calculus in which an abstraction $\lambda x.M$ is only allowed if x occurs among the free variables of M. Nowadays, "lambda calculus" refers to the $\lambda\mathsf{K}$-calculus developed under the influence of Curry, in which $\lambda x.M$ is allowed even if x does not occur in M. There are also typed versions of the lambda calculus. Of these, the most elementary are two versions of the simply typed lambda calculus $\lambda\!\to$. One version is due to [37] and has implicit types. Simply typed lambda calculus with explicit types is introduced in [29] (this system is inspired by the theory of types of [100] as simplified by [95]). In order to make a distinction between the two versions of simply typed lambda calculus, the version with explicit types is sometimes called the *Church* version and the one with implicit types the *Curry* version. The difference is that in the Church version one explicitly types a variable when it is bound after a lambda, whereas in the Curry version one does not. So for example in Church's version one has $\mathsf{I}_A = (\lambda x : A.x) : A \to A$ and similarly $\mathsf{I}_{A\to B} : (A \to B) \to (A \to B)$, while in Curry's system one has $\mathsf{I} = (\lambda x.x) : A \to A$ but also $\mathsf{I} : (A \to B) \to (A \to B)$ for the same term I. See [8] for more information about these and other typed lambda calculi. Particularly interesting are the second and higher order calculi $\lambda 2$ and $\lambda\omega$ introduced by [49] (under the names 'system F' and 'system $F\omega$') for applications to proof theory and the calculi with dependent types introduced by [26] for proof verification.

3.1. Computing on data types. In this subsection we explain how it is possible to represent data types in a very direct manner in the various lambda calculi.

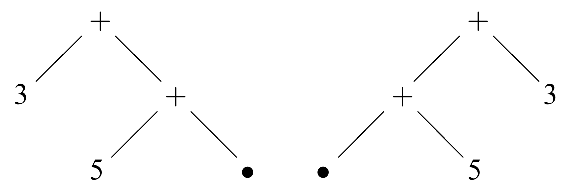
Lambda definability was introduced for functions on the set of natural numbers $\mathbb{N}$. In the resulting mathematical theory of computation (recursion theory) other domains of input or output have been treated as second class citizens by coding them as natural numbers. In more practical computer science, algorithms are also directly defined on other data types like trees or lists.

Instead of coding such data types as numbers one can treat them as first class citizens by coding them directly as lambda terms *while preserving their* *structure*. Indeed, lambda calculus is jstrong enough to do this, as was emphasized in [21] and [23]. As a result, a much more efficient representation of algorithms on these data types can be given, than when these types were represented via numbers. This methodology was perfected in two different ways in [22] and [24] or [19]. The first paper does the representation in a way that can be typed; the other papers in an essentially stronger way, but one that cannot be typed. We present the methods of these papers by treating labeled trees as an example.

Let the (inductive) data-type of labeled trees be defined by the following abstract syntax.

$$
\begin{aligned}
\texttt{tree} &= \bullet \,|\, \texttt{leaf}\,\texttt{nat} \,|\, \texttt{tree} + \texttt{tree} \\
\texttt{nat} &= 0 \,|\, \texttt{succ}\,\texttt{nat}.
\end{aligned}
$$

We see that a label can be either a bud ($\bullet$) or a leaf with a number written on it. A typical such tree is $(\texttt{leaf}\ 3) + ((\texttt{leaf}\ 5) + \bullet)$. This tree together with its mirror image look as follows.



Operation on such trees can be defined by recursion. For example the action of mirroring can be defined by

$$
\begin{aligned}
f_{\texttt{mir}}(\bullet) &= \bullet; \\
f_{\texttt{mir}}(\texttt{leaf}\ n) &= \texttt{leaf}\ n; \\
f_{\texttt{mir}}(t_1 + t_2) &= f_{\texttt{mir}}(t_2) + f_{\texttt{mir}}(t_1).
\end{aligned}
$$

Then one has for example that

$$
f_{\texttt{mir}}((\texttt{leaf}\ 3) + ((\texttt{leaf}\ 5) + \bullet)) = ((\bullet + \texttt{leaf}\ 5) + \texttt{leaf}\ 3).
$$

We will now show in two different ways how trees can be represented as lambda terms and how operations like $f_{\texttt{mir}}$ on these objects become lambda definable. The first method is from [22]. The resulting data objects and functions can be represented by lambda terms typeable in the second order lambda calculus $\lambda 2$, see [51] or [8].

DEFINITION 3.1. (i) Let b, l, p be variables (used as mnemonics for `bud`, `leaf` and `plus`). Define $\phi = \phi^{b,l,p} : \mathtt{tree} \to \mathtt{term}$, where `term` is the collection of untyped lambda terms, as follows.

$$\begin{array}{rcl} \phi(\bullet) & = & b; \\ \phi(\mathtt{leaf}\ n) & = & l\ulcorner n\urcorner; \\ \phi(t_1 + t_2) & = & p\,\phi(t_1)\phi(t_2). \end{array}$$

Here $\ulcorner n\urcorner \equiv \lambda f x. f^n x$ is Church's numeral representing n as lambda term.

(ii) Define $\psi_1 : \mathtt{tree} \to \mathtt{term}$ as follows.

$$\psi_1(t) = \lambda blp.\phi(t).$$

PROPOSITION 3.2. *Define*

$$\begin{array}{rcl} B_1 & \equiv & \lambda blp.b; \\ L_1 & \equiv & \lambda nblp.ln; \\ P_1 & \equiv & \lambda t_1 t_2 blp.p\,(t_1 blp)(t_2 blp). \end{array}$$

Then one has

(i) $\psi_1(\bullet) = B_1$.
(ii) $\psi_1(\mathtt{leaf}\ n) = L_1\ulcorner n\urcorner$.
(iii) $\psi_1(t_1 + t_2) = P_1\,\psi_1(t_1)\psi_1(t_2)$.

PROOF.
(i) Trivial.
(ii) We have

$$\begin{array}{rcl} \psi_1(\mathtt{leaf}\ n) & = & \lambda blp.\phi(\mathtt{leaf}\ n) \\ & = & \lambda blp.l\ulcorner n\urcorner \\ & = & (\lambda nblp.ln)\ulcorner n\urcorner \\ & = & L_1\ulcorner n\urcorner. \end{array}$$

(iii) Similarly, using that $\psi_1(t)blp = \phi(t)$. $\dashv$

This proposition states that the trees we considered are representable as lambda terms in such a way that the constructors ($\bullet$, `leaf` and $+$) are lambda definable. In fact, the lambda terms involved can be typed in $\lambda 2$. A nice connection between these terms and proofs in second order logic is given in [79].

Now we will show that iterative functions over these trees, like $f_{\mathtt{mir}}$, are lambda definable.

PROPOSITION 3.3 (Iteration). *Given lambda terms A_0, A_1, A_2 there exists a lambda term F such that (for variables n, t_1, t_2)*

$$\begin{aligned} FB_1 &= A_0; \\ F(L_1\, n) &= A_1\, n; \\ F(P_1 t_1 t_2) &= A_2(Ft_1)(Ft_2). \end{aligned}$$

PROOF. Take $F \equiv \lambda w.wA_0A_1A_2$. ⊣

As is well known, primitive recursive functions can be obtained from iterative functions.

There is a way of coding a finite sequence of lambda terms $M_1, \dots, M_k$ as one lambda term

$$\langle M_1, \dots, M_k\rangle \equiv \lambda z.zM_1 \dots M_k$$

such that the components can be recovered. Indeed, take

$$U_k^i \equiv \lambda x_1 \dots x_k.x_i,$$

then

$$\langle M_1, \dots, M_k\rangle U_k^i = M_i.$$

COROLLARY 3.4 (Primitive recursion). *Given lambda terms C_0, C_1, C_2 there exists a lambda term H such that*

$$\begin{aligned} HB_1 &= C_0; \\ H(L_1\, n) &= C_1\, n; \\ H(P_1 t_1 t_2) &= C_2 t_1 t_2 (Ht_1)(Ht_2). \end{aligned}$$

PROOF. Define the auxiliary function $F \equiv \lambda t.\langle t, Ht\rangle$. Then by the proposition F can be defined using iteration. Indeed,

$$F(P_1 t_1 t_2) = \langle Pt_1t_2, H(Pt_1t_2)\rangle = A_2(Ft_1)(Ft_2),$$

with

$$A_2 \equiv \lambda t_1 t_2.\langle P(t_1 U_2^1)(t_2 U_2^1), C_2(t_1 U_2^1)(t_2 U_2^1)(t_1 U_2^2)(t_2 U_2^2)\rangle.$$

Now take $H = \lambda t.FtU_2^2$. [This was the trick Kleene found at the dentist.] ⊣

Now we will present the method of [24] and [19] to represent data types. Again we consider the example of labeled trees.

DEFINITION 3.5. Define $\psi_2 : \texttt{tree} \to \texttt{term}$ as follows.

$$\begin{aligned} \psi_2(\bullet) &= \lambda e.eU_3^1 e; \\ \psi_2(\texttt{leaf}\ n) &= \lambda e.eU_3^2 ne; \\ \psi_2(t_1 + t_2) &= \lambda e.eU_3^3 \psi_2(t_1)\psi_2(t_2)e. \end{aligned}$$

Then the basic constructors for labeled trees are definable by

$$\begin{aligned} B_2 &\equiv \lambda e.eU_3^1 e; \\ L_2 &\equiv \lambda n \lambda e.eU_3^2 ne; \\ P_2 &\equiv \lambda t_1 t_2 \lambda e.eU_3^3 t_1 t_2 e. \end{aligned}$$

PROPOSITION 3.6. *Given lambda terms A_0, A_1, A_2 there exists a term F such that*

$$\begin{aligned} FB_2 &= A_0 F; \\ F(L_2 n) &= A_1 nF; \\ F(P_2 xy) &= A_2 xyF. \end{aligned}$$

PROOF. Try $F \equiv \langle\langle X_0, X_1, X_2\rangle\rangle$, the 1-tuple of a triple. Then we must have

$$\begin{aligned} FB_2 &= B_2\langle X_0, X_1, X_2\rangle \\ &= U_3^1 X_0 X_1 X_2 \langle X_0, X_1, X_2\rangle \\ &= X_0\langle X_0, X_1, X_2\rangle \\ &= A_0\langle\langle X_0, X_1, X_2\rangle\rangle \\ &= A_0 F, \end{aligned}$$

provided $X_0 = \lambda x.A_0\langle x\rangle$. Similarly one can find X_1, X_2. ⊣

This second representation is essentially untypeable, at least in typed lambda calculi in which all typeable terms are normalizing. This follows from the following consequence of a result similar to Proposition 3.6. Let $\mathsf{K} = \lambda xy.x$, $\mathsf{K}_* = \lambda xy.y$ represent true and false respectively. Then writing

$$\underline{\texttt{if}}\ \texttt{bool}\ \underline{\texttt{then}}\ X\ \underline{\texttt{else}}\ Y\ \underline{\texttt{fi}}$$

for

$$\texttt{bool}\ X\ Y,$$

the usual behavior of the conditional is obtained. Now if we represent the natural numbers as a data type in the style of the second representation, we immediately get that the lambda definable functions are closed under minimalization. Indeed, let

$$\chi(x) = \mu y[g(x, y) = 0],$$

and suppose that g is lambda defined by G. Then there exists a lambda term H such that

$$Hxy = \underline{\texttt{if}}\ \underline{\texttt{zero}}_?\,(Gxy)\ \underline{\texttt{then}}\ y\ \underline{\texttt{else}}\ (Hx(\underline{\texttt{succ}}\ y))\ \underline{\texttt{fi}}.$$

Indeed, we can write this as $Hx = AxH$ and apply Proposition 3.6, but now formulated for the inductively defined type num. Then $F \equiv \lambda x.Hx\ulcorner 0\urcorner$ does represent χ. Here $\underline{\texttt{succ}}$ represents the successor function and $\underline{\texttt{zero}}_?$ a test for zero; both are lambda definable, again by the analogon to Proposition 3.6. Since minimalization anables us to define all partial recursive functions, the terms involved cannot be typed in a normalizing system.

Self-interpretation. A lambda term M can be represented internally as a lambda term $\ulcorner M \urcorner$. This representation should be such that, for example, one has lambda terms P_1, P_2 satisfying $P_i \ulcorner X_1 X_2 \urcorner = X_i$. [67] already showed that there is a ('meta-circular') self-interpreter E such that, for closed terms M one has $\mathsf{E}\ulcorner M \urcorner = M$. The fact that data types can be represented directly in the lambda calculus was exploited by [85] to find a simpler representation for $\ulcorner M \urcorner$ and E.

The difficulty of representing lambda terms internally is that they do not form a first order algebraic data type due to the binding effect of the lambda. [85] solved this problem as follows. Consider the data type with signature

$$\mathtt{const},\ \mathtt{app},\ \mathtt{abs}$$

where `const` and `abs` are unary constructors and `app` a binary constructor. Let $\underline{\mathtt{const}}$, $\underline{\mathtt{app}}$ and $\underline{\mathtt{abs}}$ be a representation of these in lambda calculus (according to Definition 3.5).

PROPOSITION 3.7 ([85]). *Define*

$$\begin{aligned} \ulcorner x \urcorner &\equiv \underline{\mathtt{const}}\, x; \\ \ulcorner PQ \urcorner &\equiv \underline{\mathtt{app}}\, \ulcorner P \urcorner \ulcorner Q \urcorner; \\ \ulcorner \lambda x.P \urcorner &\equiv \underline{\mathtt{abs}}(\lambda x. \ulcorner P \urcorner). \end{aligned}$$

Then there exists a self-interpreter E *such that for all lambda terms M (possibly containing variables) one has*

$$\mathsf{E}\ulcorner M \urcorner = M.$$

PROOF. By an analogon to Proposition 3.6 there exists a lambda term E such that

$$\begin{aligned} \mathsf{E}(\underline{\mathtt{const}}\, x) &= x; \\ \mathsf{E}(\underline{\mathtt{app}}\, p\, q) &= (\mathsf{E}p)(\mathsf{E}q); \\ \mathsf{E}(\underline{\mathtt{abs}}\, z) &= \lambda x.\mathsf{E}(zx). \end{aligned}$$

Then by an easy induction one can show that $\mathsf{E}\ulcorner M \urcorner = M$ for all terms M.⊣

Following the construction of Proposition 3.6 in [24], this term E is given the following very simple form:

$$\mathsf{E} \equiv \langle\langle \mathsf{K}, \mathsf{S}, \mathsf{C} \rangle\rangle,$$

where $\mathsf{S} \equiv \lambda xyz.xz(yz)$ and $\mathsf{C} \equiv \lambda xyz.x(zy)$. This is a good improvement over [67] or [6]. See also [7], [9] and [10] for more about self-interpreters.

3.2. Functional programming. In this subsection a short history is presented of how lambda calculi (untyped and typed) inspired (either consciously or unconsciously) the creation of functional programming.

Imperative versus functional programming. While Church had captured the notion of computability via the lambda calculus, Turing had done the same via his model of computation based on Turing machines. When in the second world war computational power was needed for military purposes, the first electronic devices were built basically as Turing machines with random access memory. Statements in the instruction set for these machines, like $x := x+1$, are directly related to the instructions of a Turing machine. Such statements are much more easily interpreted by hardware than the act of substitution fundamental to the lambda calculus. In the beginning, the hardware of the early computers was modified each time a different computational job had to be done. Then von Neumann, who must have known[6] Turing's concept of a universal Turing machine, suggested building one machine that could be programmed to do all possible computational jobs using *software*. In the resulting computer revolution, almost all machines are based on this so called von Neumann computer, consisting of a programmable universal machine. It would have been more appropriate to call it the Turing computer.

The model of computability introduced by Church (lambda definability)—although equivalent to that of Turing—was harder to interpret in hardware. Therefore the emergence of the paradigm of functional programming, that is based essentially on lambda definability, took much more time. Because functional programs are closer to the specification of computational problems than imperative ones, this paradigm is more convenient than the traditional imperative one. Another important feature of functional programs is that parallelism is much more naturally expressed in them, than in imperative programs. See [117] and [64] for some evidence for the elegance of the functional paradigm. The implementation difficulties for functional programming have to do with memory usage, compilation time and actual run time of functional programs. In the contemporary state of the art of implementing functional languages, these problems have been solved satisfactorily.[7]

Classes of functional languages. Let us describe some languages that have been—and in some cases still are—influential in the expansion of functional programming. These languages come in several classes.

[6]Church had invited Turing to the United States in the mid 1930's. After his first year it was von Neumann who invited Turing to stay for a second year. See [60].

[7]Logical programming languages also have the mentioned advantages. But so far pure logical languages of industrial quality have not been developed. (Prolog is not pure and λ-Prolog, see [87], although pure, is presently a prototype.)

Lambda calculus by itself is not yet a complete model of computation, since an expression M may be evaluated by different so-called reduction strategies that indicate which sub-term of M is evaluated first (see [6, Chapter 12]). By the Church-Rosser theorem this order of evaluation is not important for the final result: the normal form of a lambda term is unique if it exists. But the order of evaluation makes a difference for efficiency (both time and space) and also for the question whether or not a normal form is obtained at all.

So called 'eager' functional languages have a reduction strategy that evaluates an expression like FA by first evaluating F and A (in no particular order) to, say, $F' \equiv \lambda a.\ \cdots a \cdots a \cdots$ and A' and then contracting $F'A'$ to $\ldots A' \ldots A' \ldots$. This evaluation strategy has definite advantages for the efficiency of the implementation. The main reason for this is that if A is large, but its normal form A' is small, then it is advantageous both for time and space efficiency to perform the reduction in this order. Indeed, evaluating FA directly to

$$\cdots A \cdots A \cdots$$

takes more space and if A is now evaluated twice, it also takes more time.

Eager evaluation, however, is not a normalizing reduction strategy in the sense of [6, Chapter 12]. For example, if $F \equiv \lambda x.\mathsf{I}$ and A does not have a normal form, then evaluating FA eagerly diverges, while

$$FA \equiv (\lambda x.\mathsf{I})A = \mathsf{I},$$

if it is evaluated leftmost outermost (roughly 'from left to right'). This kind of reduction is called 'lazy evaluation'.

It turns out that eager languages are, nevertheless, computationally complete, as we will soon see. The implementation of these languages was the first milestone in the development of functional programming. The second milestone consisted of the efficient implementation of lazy languages.

In addition to the distinction between eager and lazy functional languages there is another one of equal importance. This is the difference between untyped and typed languages. The difference comes directly from the difference between the untyped lambda calculus and the various typed lambda calculi, see [8]. Typing is useful, because many programming bugs (errors) result in a typing error that can be detected automatically prior to running one's program. On the other hand, typing is not too cumbersome, since in many cases the types need not be given explicitly. The reason for this is that, by the type reconstruction algorithm of [38] and [59] (later rediscovered by [84]), one can automatically find the type (in a certain context) of an untyped but typeable expression. Therefore, the typed versions of functional programming languages are often based on the implicitly typed lambda calculi *à la*

Curry. Types also play an important role in making implementations of lazy languages more efficient, see below.

Besides the functional languages that will be treated below, the languages APL and FP have been important historically. The language APL, introduced in [65], has been, and still is, relatively widespread. The language FP was designed by Backus, who gave, in his lecture ([5]) at the occasion of receiving his Turing award (for his work on imperative languages) a strong and influential plea for the use of functional languages. Both APL and FP programs consist of a set of basic functions that can be combined to define operations on data structures. The language APL has, for example, many functions for matrix operations. In both languages composition is the only way to obtain new functions and, therefore, they are less complete than a full functional language in which user defined functions can be created. As a consequence, these two languages are essentially limited in their ease of expressing algorithms.

Eager functional languages. Let us first give the promised argument that eager functional languages are computationally complete. Every computable (recursive) function is lambda definable in the λI-calculus (see [30] or [6, Theorem 9.2.16]). In the λI-calculus a term having a normal form is strongly normalizing (see [31] or [6, Theorem 9.1.5]). Therefore an eager evaluation strategy will find the required normal form.

The first functional language, LISP, was designed and implemented by [83]. The evaluation of expressions in this language is eager. LISP had (and still has) considerable impact on the art of programming. Since it has a good programming environment, many skillful programmers were attracted to it and produced interesting programs (so called 'artificial intelligence'). LISP is not a pure functional language for several reasons. Assignment is possible in it; there is a confusion between local and global variables[8] ('dynamic binding'; some LISP users even like it); LISP uses the '`Quote`', where (`Quote` M) is like $\ulcorner M \urcorner$. In later versions of LISP, Common LISP (see [110]) and Scheme (see [32]), dynamic binding is no longer present. The '`Quote`' operator, however, is still present in these languages. Since $\mathsf{I}a = a$ but $\ulcorner \mathsf{I}a \urcorner \neq \ulcorner a \urcorner$ adding '`Quote`' to the lambda calculus is inconsistent. As one may not reduce in LISP within the scope of a '`Quote`', however, having a '`Quote`' in LIPS is not inconsistent. '`Quote`' is not an available function but only a constructor. That is, if M is a well-formed expression, so is

[8]This means substitution of an expression with a free variable into a context in which that variable becomes bound. The originators of LISP were in good company: in [58] the same mistake was made.

$(\texttt{Quote}\ M)$[9]. Also, LISP has a primitive fixed-point operator 'LABEL' (implemented as a cycle) that is also found in later functional languages.

In the meantime, [77] developed an abstract machine—the SECD machine—for the implementation of reduction. Many implementations of eager functional languages, including some versions of LISP, have used, or are still using, this computational model. (The SECD machine also can be modelled for lazy functional languages, see [57].) Another way of implementing functional languages is based on the so called CPS-translation. This was introduced in [96] and used in a compilers by [109] and [3]. See also [93] and [97].

The first important typed functional language with an eager evaluation strategy is Standard ML, see [84]. This language is based on the Curry variant of $\lambda\!\rightarrow$, the simply typed lambda calculus with implicit typing, see [8]. Expressions are type-free, but are only legal if a type can be derived for them. By the algorithm of Curry and Hindley cited above, it is decidable whether an expression does have a type and, moreover, its most general type can be computed. Milner added two features to $\lambda\!\rightarrow$. The first is the addition of new primitives. One has the fixed-point combinator Y as primitive, with essentially all types of the form $(A \to A) \to A$, with $A \equiv (B \to C)$, assigned to it. Indeed, if $f : A \to A$, then $\mathsf{Y}f$ is of type A so that both sides of

$$f(\mathsf{Y}f) = \mathsf{Y}f$$

have type A. Primitives for basic arithmetic operations are also added. With these additions, ML becomes a universal programming language, while $\lambda\!\rightarrow$ is not (since all its terms are normalizing). The second addition to ML is the 'let' construction

$$\underline{\texttt{let}}\ x\ \underline{\texttt{be}}\ N\ \underline{\texttt{in}}\ M\ \underline{\texttt{end}}. \tag{1}$$

This language construct has as its intended interpretation

$$M[x := N], \tag{2}$$

so that one may think that the let construction is not necessary. If, however, N is large, then this translation of (1) becomes space inefficient. Another

[9] Using '$\texttt{Quote}$' as a function would violate the Church-Rosser property. An example is

$$(\lambda x.x(\mathsf{I}a))\ \texttt{Quote}$$

that then would reduce to both

$$\texttt{Quote}\ (\mathsf{I}a) \to \ulcorner \mathsf{I}a \urcorner$$

and to

$$(\lambda x.xa)\ \texttt{Quote} \to \texttt{Quote}\ a \to \ulcorner a \urcorner$$

and there is no common reduct for these two expressions $\ulcorner \mathsf{I}a \urcorner$ and $\ulcorner a \urcorner$.

interpretation of (1) is

$$(3) \qquad (\lambda x.M)N.$$

But this interpretation has its limitations, as N has to be given one fixed type, whereas in (2) the various occurrences of N may have different types. The expression (1) is a way to make use of both the space reduction ('sharing') of the expression (3) and the 'implicit polymorphism' in which N can have more than one type of (2). An example of the let expression is

$$\underline{\texttt{let}}\ \texttt{id}\ \underline{\texttt{be}}\ \lambda x.x\ \underline{\texttt{in}}\ \lambda f x.(\texttt{id}\ f)(\texttt{id}\ x)\ \underline{\texttt{end}}.$$

This is typeable by

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

if the second occurrence of `id` gets type $(A \to A) \to (A \to A)$ and the third $(A \to A)$.

Because of its relatively efficient implementation and the possibility of type checking at compile time (for finding errors), the language ML has evolved into important industrial variants (like Standard ML of New Jersey).

Although not widely used in industry, a more efficient implementation of ML is based on the abstract machine CAML, see [34]. CAML was inspired by the categorical foundations of the lambda calculus, see [107], [72] and [35]. All of these papers have been inspired by the work on denotational semantics of Scott, see [103] and [54].

Lazy functional languages. Although all computable functions can be represented in an eager functional programming language, not all reductions in the full $\lambda\mathsf{K}$-calculus can be performed using eager evaluation. We already saw that if $F \equiv \lambda x.\mathsf{I}$ and A does not have a normal form, then eager evaluation of FA does not terminate, while this term does have a normal form. In 'lazy' functional programming languages the reduction of FA to I is possible, because the reduction strategy for these languages is essentially leftmost outermost reduction which is normalizing.

One of the advantages of having lazy evaluation is that one can work with 'infinite' objects. For example there is a legal expression for the potentially infinite lists of primes

$$[2, 3, 5, 7, 11, 13, 17, \dots],$$

of which one can take the n-th projection in order to get the n-th prime. See [117] and [64] for interesting uses of the lazy programming style.

Above we explained why eager evaluation can be implemented more efficiently than lazy evaluation: copying large expressions is expensive because of space and time costs. In [119] the idea of *graph reduction* was introduced in order to also do lazy evaluation efficiently. In this model of computation, an expression like $(\lambda x.\cdots x \cdots x \cdots)A$ does not reduce to $\cdots A \cdots A \cdots$

but to $\cdots @ \cdots @ \cdots\ ;\ @ : A$, where the first two occurrences of @ are pointers referring to the A behind the third occurrence. In this way lambda expressions become dags (directed acyclic graphs).[10]

Based on the idea of graph reduction, using carefully chosen combinators as primitives, the experimental language SASL, see [115] and [116], was one of the first implemented lazy functional languages. The notion of graph reduction was extended by Turner by implementing the fixed-point combinator (one of the primitives) as a cyclic graph. (Cyclic graphs were already described in [119] but were not used there.) Like LISP, the language SASL is untyped. It is fair to say that—unlike programs written in the eager languages such as LISP and Standard ML—the execution of SASL programs was orders of magnitude slower than that of imperative programs in spite of the use of graph reduction.

In the 1980s typed versions of lazy functional languages did emerge, as well as a considerable speed-up of their performance. A lazy version of ML, called Lazy ML (LML), was implemented efficiently by a group at Chalmers University, see [66]. As underlying computational model they used the so called G-machine, that avoids building graphs whenever efficient. For example, if an expression is purely arithmetical (this can be seen from type information), then the evaluation can be done more efficiently than by using graphs. Another implementation feature of the LML is the compilation into super-combinators, see [63], that do not form a fixed set, but are created on demand depending on the expression to be evaluated. Emerging from SASL, the first fully developed typed lazy functional language called Miranda™ was developed by [118]. Special mention should be made of its elegance and its functional I/O interface (see below).

Notably, the ideas in the G-machine made lazy functional programming much more efficient. In the late 1980s very efficient implementations of two typed lazy functional languages appeared that we will discuss below: Clean, see [40], and Haskell, see [92], [62]. These languages, with their implementations, execute functional programs in a way that is comparable to the speed of contemporary imperative languages such as C.

Interactive functional languages. The versions of functional programming that we have considered so far could be called 'autistic'. A program consists of an expression M, its execution of the reduction of M and its output of the normal form $M^{\mathtt{nf}}$ (if it exists). Although this is quite useful for many

[10]Robin Gandy mentioned at a meeting for the celebration of his seventieth birthday that already in the early 1950s Turing had told him that he wanted to evaluate lambda terms using graphs. In Turing's description of the evaluation mechanism he made the common oversight of confusing free and bound variables. Gandy pointed this out to Turing, who then said: "Ah, this remark is worth 100 pounds a month!"

purposes, no interaction with the outside world is made. Even just dealing with input and output (I/O) requires interaction.

We need the concept of a 'process' as opposed to a function. Intuitively a process is something that (in general) is geared towards continuation while a function is geared towards termination. Processes have an input channel on which an input stream (a potentially infinite sequence of tokens) is coming in and an output channel on which an output stream is coming out. A typical process is the control of a traffic light system: it is geared towards continuation, there is an input stream (coming from the pushbuttons for pedestrians) and an output stream (regulating the traffic lights). Text editing is also a process. In fact, even the most simple form of I/O is already a process.

A primitive way to deal with I/O in a functional language is used in some versions of ML. There is an input stream and an output stream. Suppose one wants to perform the following process P:

read the first two numbers x, y of the input stream;
put their difference $x - y$ onto the output stream.

Then one can write in ML the following program

$$\texttt{write}\ (\texttt{read} - \texttt{read}).$$

This is not very satisfactory, since it relies on a fixed order of evaluation of the expression '$\texttt{read} - \texttt{read}$'.

A more satisfactory way consists of so-called continuations, see [52]. To the lambda calculus one adds primitives `Read, Write` and `Stop`. The operational semantics of an expression is now as follows:

$$\begin{array}{rcll} M & \Rightarrow & M^{\texttt{hnf}}, & \text{where } M^{\texttt{hnf}} \text{ is the head normal form}^{11} \text{ of } M; \\ \texttt{Read}\ M & \Rightarrow & M\ a, & \text{where } a \text{ is taken off the input stream;} \\ \texttt{Write}\ b\ M & \Rightarrow & M, & \text{and } b \text{ is put into the output stream;} \\ \texttt{Stop} & \Rightarrow & & \text{i.e., do nothing.} \end{array}$$

Now the process P above can be written as

$$P = \texttt{Read}\ (\lambda x.\,\texttt{Read}\ (\lambda y.\,\texttt{Write}\ (x - y)\ \texttt{Stop})).$$

If, instead, one wants a process Q that continuously takes two elements of the input stream and put the difference on the output stream, then one can write as a program the following extended lambda term

$$Q = \texttt{Read}\ (\lambda x.\,\texttt{Read}\ (\lambda y.\,\texttt{Write}\ (x - y)\ Q)),$$

[11] A head nf in lambda calculus is of the form $\lambda\vec{x}.yM_1 \ldots M_n$, with the $M_1 \ldots M_n$ possibly not in nf.

which can be found using the fixed-point combinator.

Now, every interactive program can be written in this way, provided that special commands written on the output stream are interpreted. For example one can imagine that writing

$$\text{`}\mathtt{echo}\text{'}\ 7 \quad \text{or} \quad \text{`}\mathtt{print}\text{'}\ 7$$

on the output channel will put 7 on the screen or print it out respectively. The use of continuations is equivalent to that of monads in programming languages like Haskell, as shown in [52]. (The present version of Haskell I/O is more refined than this; we will not consider this issue.)

If $A_0, A_1, A_2, \ldots$ is an effective sequence of terms (i.e., $A_n = F \ulcorner n \urcorner$ for some F), then this infinite list can be represented as a lambda term

$$\begin{aligned} [A_0, A_1, A_2, \ldots] &\equiv [A_0, [A_1, [A_2, \ldots]]] \\ &= H \ulcorner 0 \urcorner, \end{aligned}$$

where $[M, N] \equiv \lambda z.zMN$ and

$$H \ulcorner n \urcorner = [F \ulcorner n \urcorner, H \ulcorner n+1 \urcorner].$$

This H can be defined using the fixed-point combinator.

Now the operations `Read, Write` and `Stop` can be made explicitly lambda definable if we use

$$\begin{aligned} \mathtt{In} &= [A_0, A_1, A_2, \ldots], \\ \mathtt{Out} &= [\ldots, B_2, B_1, B_0], \end{aligned}$$

where `In` is a representation of the potentially infinite input stream given by 'the world' (i.e., the user and the external operating system) and `Out` of the potentially infinite output stream given by the machine running the interactive functional language. Every interactive program M should be acting on $[\mathtt{In}, \mathtt{Out}]$ as argument. So M in the continuation language becomes

$$M\ [\mathtt{In}, \mathtt{Out}].$$

The following definition then matches the operational semantics.

$$(1) \qquad \begin{cases} \mathtt{Read}\ F\ [[A, \mathtt{In}'], \mathtt{Out}] & = \quad F\ A\ [\mathtt{In}', \mathtt{Out}]; \\ \mathtt{Write}\ F\ B\ [\mathtt{In}, \mathtt{Out}] & = \quad F\ [\mathtt{In}, [B, \mathtt{Out}]] \\ \mathtt{Stop}\ [\mathtt{In}, \mathtt{Out}] & = \quad [\mathtt{In}, \mathtt{Out}]. \end{cases}$$

In this way $[\mathtt{In}, \mathtt{Out}]$ acts as a dynamic state. An operating system should take care that the actions on $[\mathtt{In}, \mathtt{Out}]$ are actually performed to the I/O channels. Also we have to take care that statements like `'echo'` 7 are being interpreted. It is easy to find pure lambda terms `Read, Write` and `Stop` satisfying (1). This seems to be a good implementation of the continuations and therefore a good way to deal with interactive programs.

There is, however, a serious problem. Define

$$M \equiv \lambda p.[\texttt{Write}\ b_1\ \texttt{Stop}\ p, \texttt{Write}\ b_2\ \texttt{Stop}\ p].$$

Now consider the evaluation

$$\begin{aligned} M\ [\texttt{In}, \texttt{Out}] &= [\texttt{Write}\ b_1\ \texttt{Stop}\ [\texttt{In}, \texttt{Out}], \texttt{Write}\ b_2\ \texttt{Stop}\ [\texttt{In}, \texttt{Out}]] \\ &= [[\texttt{In}, [b_1,\ \texttt{Out}]], [\texttt{In}, [b_2,\ \texttt{Out}]]. \end{aligned}$$

Now what will happen to the actual output channel: should b_1 be added to it, or perhaps b_2?

The dilemma is caused by the duplication of the I/O channels $[\texttt{In}, \texttt{Out}]$. One solution is not to explicitly mention the I/O channels, as in the lambda calculus with continuations. This is essentially what happens in the method of monads in the interactive functional programming language Haskell. If one writes something like

$$\texttt{Main}\ f_1 \circ \cdots \circ f_n$$

the intended interpretation is $(f_1 \circ \cdots \circ f_n)[\texttt{In}, \texttt{Out}]$.

The solution put forward in the functional language Clean is to use a typing system that guarantees that the I/O channels are never duplicated. For this purpose a so-called 'uniqueness' typing system is designed, see [14, 15], that is related to linear logic (see [50]). Once this is done, one can improve the way in which parts of the world are used explicitly. A representation of all aspects of the world can be incorporated in lambda calculus. Instead of having just $[\texttt{In}, \texttt{Out}]$, the world can now be extended to include (a representation of) the screen, the printer, the mouse, the keyboard and whatever gadgets one would like to add to the computer periphery (e.g., other computers to form a network). So interpreting

$$\texttt{`print'}\ 7$$

now becomes simply something like

$$\texttt{put}\ 7\ \texttt{printer}.$$

This has the advantage that if one wants to echo a 7 and to print a 3, but the order in which this happens is immaterial, then one is not forced to make an over-specification, like sending first `'print'` 3 and then `'echo'` 7 to the output channel:

$$[\ \dots, \texttt{`echo'}\ 7,\ \texttt{`print'}\ 3].$$

By representing inside the lambda calculus with uniqueness types as many gadgets of the world as one would like, one can write something like

$$F\ [\texttt{keyboard}, \texttt{mouse}, \texttt{screen}, \texttt{printer}]$$

$$= [\,\texttt{keyboard}, \texttt{mouse}, \texttt{put}\ 3\ \texttt{screen}, \texttt{put}\ 7\ \texttt{printer}\,].$$

What happens first depends on the operating system and parameters, that we do not know (for example on how long the printing queue is). But we are not interested in this. The system satisfies the Church-Rosser theorem and the eventual result (7 is printed and 3 is echoed) is unambiguous. This makes Clean somewhat more natural than Haskell (also in its present version) and definitely more appropriate for an implementation on parallel hardware.

Both Clean and Haskell are state of the art functional programming languages producing efficient code; as to compiling time Clean belongs to the class of fast compilers (including those for imperative languages). Many serious applications are written in these languages. The interactive aspect of both languages is made possible by lazy evaluation and the use of higher type[12] functions, two themes that are at the core of the lambda calculus (λK, that is). It is to be expected that they will have a significant impact on the production of modern (interactive window based) software.

§4. Reasoning.

Computer mathematics. Modern systems for computer algebra (CA) are able to represent mathematical notions on a machine and compute with them. These objects can be integers, real or complex numbers, polynomials, integrals and the like. The computations are usually symbolic, but can also be numerical to a virtually arbitrary degree of precision. It is fair to say—as is sometimes done—that "a system for CA can represent $\sqrt{2}$ exactly". In spite of the fact that this number has an infinite decimal expansion, this is not a miracle. The number $\sqrt{2}$ is represented in a computer just as a symbol (as we do on paper or in our mind), and the machine knows how to manipulate it. The common feature of these kind of notions represented in systems for CA is that in some sense or another they are all computable. Systems for CA have reached a high level of sophistication and efficiency and are commercially available. Scientists and both pure and applied mathematicians have made good use of them for their research.

There is now emerging a new technology, namely that of systems for Computer Mathematics (CM). In these systems virtually all mathematical notions can be represented exactly, including those that do not have a computational nature. How is this possible? Suppose, for example, that we want to represent a non-computable object like the co-Diophantine set

$$X = \{n \in \mathbb{N} \mid \neg\exists \vec{x}\ D(\vec{x}, n) = 0\}.$$

[12]In the functional programming community these are called 'higher *order* functions'. We prefer to use the more logically correct expression 'higher *type*', since 'higher order' refers to quantification over types (like in the system $\lambda 2$).

Then we can do as before and represent it by a special symbol. But now the computer in general cannot operate on it because the object may be of a non-computational nature.

Before answering the question in the previous paragraph, let us first analyze where non-computability comes from. It is always the case that this comes from the quantifiers $\forall$ (for all) and $\exists$ (exists). Indeed, these quantifiers usually range over an infinite set and therefore one loses decidability.

Nevertheless, for ages mathematicians have been able to obtain interesting information about these non-computable objects. This is because there is a notion of *proof*. Using proofs one can state with confidence that e.g.,

$$3 \in X, \text{ i.e., } \neg\exists\vec{x}\, D(\vec{x}, 3) = 0.$$

Aristotle had already remarked that it is often hard to find proofs, but the verification of a putative one can be done in a relatively easy way. Another contribution of Aristotle was his quest for the formalization of logic. After about 2300 years, when Frege had found the right formulation of predicate logic and Gödel had proved that it is complete, this quest was fulfilled. Mathematical proofs can now be completely formalized and verified by computers. This is the underlying basis for the systems for CM.

Present day prototypes of systems for CM are able to help a user to develop from primitive notions and axioms many theories, consisting of defined concepts, theorems and proofs.[13] All the systems of CM have been inspired by the AUTOMATH project of de Bruijn (see [26] and [27] and [88]) for the automated verification of mathematical proofs.

Representing proofs as lambda terms. Now that mathematical proofs can be fully formalized, the question arises how this can be done best (for efficiency reasons concerning the machine and pragmatic reasons concerning the human user). Hilbert represented a proof of statement A from a set of axioms Γ as a finite sequence $A_0, A_1 \dots, A_n$ such that $A = A_n$ and each A_i, for $0 \leq i \leq n$, is either in Γ or follows from previous statements using the rules of logic.

A more efficient way to represent proofs employs typed lambda terms and is called the *propositions-as-types* interpretation discovered by Curry, Howard and de Bruijn. This interpretation maps propositions into types and proofs into the corresponding inhabitants. The method is as follows. A statement A is transformed into the type (i.e., collection)

$$[A] = \text{ the set of proofs of } A.$$

So A is provable if and only if $[A]$ is 'inhabited' by a proof p. Now a proof of $A \Rightarrow B$ consists (according to the Brouwer-Heyting interpretation of

[13]This way of doing mathematics, the axiomatic method, was also described by Aristotle. It was [42] who first used this method very successfully in his Elements.

implication) of a function having as argument a proof of A and as value a proof of B. In symbols

$$[A \Rightarrow B] = [A] \to [B].$$

Similarly

$$[\forall x \in X.Px] = \Pi x : X.[Px],$$

where $\Pi x : A.[Px]$ is the Cartesian product of the $[Px]$, because a proof of $\forall x \in A.Px$ consists of a function that assigns to each element $x \in A$ a proof of Px. In this way proof-objects become isomorphic with the intuitionistic natural deduction proofs of [48]. Using this interpretation, a proof of $\forall y \in A.Py \Rightarrow Py$ is $\lambda y{:}A\lambda x{:}Py.x$. Here $\lambda x{:}A.B(x)$ denotes the function that assigns to input $x \in A$ the output $B(x)$. A proof of

$$(A \Rightarrow A \Rightarrow B) \Rightarrow A \Rightarrow B$$

is

$$\lambda p{:}(A \Rightarrow A \Rightarrow B)\lambda q{:}A.pqq.$$

A description of the typed lambda calculi in which these types and inhabitants can be formulated is given in [8], which also gives an example of a large proof object. Verifying whether p is a proof of A boils down to verifying whether, in the given context, the type of p is equal (convertible) to $[A]$. The method can be extended by also representing connectives like & and $\neg$ in the right type system. Translating propositions as types has as default intuitionistic logic. Classical logic can be dealt with by adding the excluded middle as an axiom.

If a complicated computer system claims that a certain mathematical statement is correct, then one may wonder whether this is indeed the case. For example, there may be software errors in the system. A satisfactory methodological answer has been given by de Bruijn. Proof-objects should be public and written in such a formalism that a reasonably simple proof-checker can verify them. One should be able to verify the program for this proof-checker 'by hand'. We call this the *de Bruijn criterion*. The proof-development systems Lego (see [80]) and Coq (see [33]) satisfy this criterion.

A way to keep proof-objects from growing too large is to employ the so-called Poincaré principle. [94, p. 12] stated that an argument showing that $2 + 2 = 4$ "is not a proof in the strict sense, it is a verification" (actually he claimed that an arbitrary mathematician will make this remark). In the AUTOMATH project of de Bruijn the following interpretation of the Poincaré principle was given. If p is a proof of $A(t)$ and $t =_R t'$, then the same p is also a proof of $A(t')$. Here R is a notion of reduction consisting of ordinary β-reduction and δ-reduction in order to deal with the unfolding of definitions. Since β-δ-reduction is not too complicated to be programmed, the type systems enjoying this interpretation of the Poincaré principle still satisfy the de Bruijn criterion[14].

In spite of the compact representation in typed lambda calculi and the use of the Poincaré principle, proof-objects become large, something like 10 to 30 times the length of a complete informal proof. Large proof-objects are tiresome to generate by hand. With the necessary persistence [18] has written lambda after lambda to obtain the proof-objects showing that all proofs (but one) in [76] are correct. Using a modern system for CM one can do better. The user introduces the context consisting of the primitive notions and axioms. Then necessary definitions are given to formulate a theorem to be proved (the goal). The proof is developed in an interactive session with the machine. Thereby the user only needs to give certain 'tactics' to the machine. (The interpretation of these tactics by the machine does nothing mathematically sophisticated, only the necessary bookkeeping. The sophistication comes from giving the right tactics.) The final goal of this research is that the necessary effort to interactively generate formal proofs is not more complicated than producing a text in, say, LaTeX. This goal has not been reached yet. See [11] for references, including those about other approaches to computer mathematics. (These include the systems NuPrl, HOL, Otter, Mizar and the Boyer-Moore theorem prover. These systems do not satisfy the de Bruijn criterion, but some of them probably can be modified easily so that they do.)

Computations in proofs. The following is taken from [12]. There are several computations that are needed in proofs. This happens, for example, if we want to prove formal versions of the following intuitive statements.

(1) $[\sqrt{45}] = 6$ where $[r]$ is the integer part of a real;
(2) $\mathtt{Prime}(61)$
(3) $(x+1)(x+1) = x^2 + 2x + 1$.

A way to handle (1) is to use the Poincaré principle extended to the reduction relation $\twoheadrightarrow_\iota$ for primitive recursion on the natural numbers. Operations like $f(n) = [\sqrt{n}\,]$ are primitive recursive and hence are lambda definable (using $\twoheadrightarrow_{\beta\iota}$) by a term, say F, in the lambda calculus extended by an operation for primitive recursion R satisfying

$$\begin{aligned} R\,A\,B\,\mathtt{zero} &\to_\iota A \\ R\,A\,B\,(\mathtt{succ}\,x) &\to_\iota B\,x\,(R\,A\,B\,x). \end{aligned}$$

[14] The reductions may sometimes cause the proof-checking to be of an unacceptable time complexity. We have that p is a proof of A iff $\mathtt{type}(p) =_{\beta\delta} A$. Because the proof is coming from a human, the necessary conversion path is feasible, but to find it automatically may be hard. The problem probably can be avoided by enhancing proof-objects with hints for a reduction strategy.

Then, writing $\ulcorner 0 \urcorner = \texttt{zero}, \ulcorner 1 \urcorner = \texttt{succ zero}, \dots$, as

$$\ulcorner 6 \urcorner = \ulcorner 6 \urcorner$$

is formally derivable, it follows from the Poincaré principle that the same is true for

$$F \ulcorner 45 \urcorner = \ulcorner 6 \urcorner$$

(with the same proof-object), since $F \ulcorner 45 \urcorner \twoheadrightarrow_{\beta\iota} \ulcorner 6 \urcorner$. Usually, a proof obligation arises that F is adequately constructed. For example, in this case it could be

$$\forall n \; (F\,n)^2 \;\leq\; n \;< ((F\,n) + 1)^2.$$

Such a proof obligation needs to be formally proved, but only once; after that reductions like

$$F \ulcorner n \urcorner \twoheadrightarrow_{\beta\iota} \ulcorner f(n) \urcorner$$

can be used freely many times.

In a similar way, a statement like (2) can be formulated and proved by constructing a lambda defining term $K_{\texttt{Prime}}$ for the characteristic function of the predicate $\texttt{Prime}$. This term should satisfy the following statement

$$\begin{aligned} \forall n \;\; [(&\texttt{Prime}\, n \;\leftrightarrow\; K_{\texttt{Prime}}\, n = \ulcorner 1 \urcorner) \,\& \\ &(K_{\texttt{Prime}}\, n = \ulcorner 0 \urcorner \;\vee\; K_{\texttt{Prime}}\, n = \ulcorner 1 \urcorner)]. \end{aligned}$$

which is the proof obligation.

Statement (3) corresponds to a symbolic computation. This computation takes place on the syntactic level of formal terms. There is a function g acting on syntactic expressions satisfying

$$g((x+1)(x+1)) = x^2 + 2x + 1,$$

that we want to lambda define. While $x + 1 : \mathsf{Nat}$ (in context $x : \mathsf{Nat}$), the expression on a syntactic level represented internally satisfies '$x + 1$' : $\texttt{term}(\mathsf{Nat})$, for the suitably defined inductive type $\texttt{term}(\mathsf{Nat})$. After introducing a reduction relation $\twoheadrightarrow_\iota$ for primitive recursion over this data type, one can use techniques similar to those of §3 to lambda define g, say by G, so that

$$G\,\text{‘}(x+1)(x+1)\text{’} \twoheadrightarrow_{\beta\iota} \text{‘}x^2 + 2x + 1\text{’}.$$

Now in order to finish the proof of (3), one needs to construct a self-interpreter E, such that for all expressions $p : \mathsf{Nat}$ one has

$$\mathsf{E}\,\text{‘}p\text{’} \;\twoheadrightarrow_{\beta\iota}\; p$$

and prove the proof obligation for G which is

$$\forall t : \texttt{term}(\mathsf{Nat})\; \mathsf{E}(G\,t) \;=\; \mathsf{E}\,t.$$

It follows that

$$\mathsf{E}(G\,`(x+1)(x+1)\,') \;=\; \mathsf{E}\,`(x+1)(x+1)\,';$$

now since

$$\begin{aligned} \mathsf{E}(G\,`(x+1)(x+1)\,') &\twoheadrightarrow_{\beta\iota} \mathsf{E}\,`x^2+2x+1' \\ &\twoheadrightarrow_{\beta\iota} x^2+2x+1 \\ \mathsf{E}\,`(x+1)(x+1)\,' &\twoheadrightarrow_{\beta\iota} (x+1)(x+1), \end{aligned}$$

we have by the Poincaré principle

$$(x+1)(x+1) \;=\; x^2+2x+1.$$

The use of inductive types like Nat and $\mathtt{term}(\mathsf{Nat})$ and the corresponding reduction relations for primitive reduction was suggested by [102] and the extension of the Poincaré principle for the corresponding reduction relations of primitive recursion by [81]. Since such reductions are not too hard to program, the resulting proof checking still satisfies the de Bruijn criterion.

In [90] a program is presented that, for every primitive recursive predicate P, constructs the lambda term K_P defining its characteristic function and the proof of the adequacy of K_P. The resulting computations for $P = \mathtt{Prime}$ are not efficient, because a straightforward (non-optimized) translation of primitive recursion is given and the numerals (represented numbers) used are in a unary (rather than n-ary) representation; but the method is promising. In [41], a more efficient ad hoc lambda definition of the characteristic function of $\mathtt{Prime}$ is given, using Fermat's small theorem about primality. Also the required proof obligation has been given.

Choice of formal systems. There are several possibilities for the choice of a formal system to be used for the representation of theories in systems of computer mathematics. Since, in constructing proof-objects, cooperation between researchers is desirable, this choice has to be made with some care in order to reach an international standard. As a first step towards this, one may restrict attention to systems of typed lambda calculi, since they provide a compact representation and meet de Bruijn's criterion of having a simple proof-checker. In their simplest form, these systems can be described in a uniform way as pure type systems (PTS's) of different strength, see [8]. The PTS's should be extended by a definition mechanism to become DPTS's (PTS's with definitions), see [104]. The DPTS's are good for describing several variants of logic: many sorted predicate logic in its first, second or higher order versions. As stated before, the default logic is intuitionistic, but can be made classical by assuming the excluded middle.

The next step consists of adding inductive types (IT's) and the corresponding reduction relations in order to capture primitive recursion. We suggest that the right formal systems to be used for computer mathematics are the type systems (TS), consisting of DPTS's extended by IT's, as described e.g., in [91]. TS's come with two parameters. The first is the specification $\mathcal{A}$ of the underlying PTS specifying its logical strength, see [8]. The second is $\mathcal{B}$ the collection of inductive types and their respective notions of reduction $\twoheadrightarrow_\iota$ specifying its mathematical and computational strength. In my opinion, a system for proof-checking should be able to verify proof-objects written in all the systems $\mathrm{TS}(\mathcal{A}, B)$ (for a 'reasonable' choice spectrum of the parameters). If someone wants to use it for only a subclass of the choice of parameters—dictated by that person's foundational views—then the proof-checker will do its work anyway. I believe that this generality will not be too expensive in terms of the complexity of the checking.[15]

Illative lambda calculus. Curry and his students continued to look for a way to represent functions and logic into one adequate formal system. Some of the proposed systems turned out to be inconsistent, other ones turned out to be incomplete. Research in TS's for the representation of logic has resulted in an unexpected side effect. By making a modification inspired by the TS's, it became possible, after all, to give an extension of the untyped lambda calculus, called *Illative Lambda Calculi* (ILC; 'illative' from the Latin word *inferre* which means to infer), such that first order logic can be faithfully and completely embedded into it. The method can be extended for an arbitrary PTS[16], so that higher order logic can be represented too.

The resulting ILC's are in fact simpler than the TS's. But doing computer mathematics via ILC is probably not very practical, as it is not clear how to do proof-checking for these systems.

One nice thing about the ILC is that the old dream of Church and Curry came true, namely, there is one system based on untyped lambda calculus (or combinators) on which logic, hence mathematics, can be based. More importantly there is a 'combinatory transformation' between the ordinary interpretation of logic and its propositions-as-types interpretation. Basically, the situation is as follows. The interpretation of predicate logic in ILC is such that

$$\begin{aligned} \vdash_{\text{logic}} A \text{ with proof } p &\iff \forall r \vdash_{\text{ILC}} [A]_r[p] \\ &\iff \vdash_{\text{ILC}} [A]_{\mathsf{I}}[p] \\ &\iff \vdash_{\text{ILC}} [A]_{\mathsf{K}}[p] = \mathsf{K}[A]'_{\mathsf{I}}[p] = [A]'_{\mathsf{I}}, \end{aligned}$$

[15]It may be argued that the following list of features is so important that they deserve to be present in TS's as primitives and be implemented: quotient types (see [61]), subtypes (see [4]) and type inclusion (see [80]). This is an interesting question and experiments should be done to determine whether this is the case or whether these can be translated into the more basic TS's in a sufficiently efficient way (possibly using some macros in the system for CM).

[16]For first order logic, the embedding is natural, but e.g., for second order logic this is less so. It is an open question whether there exists a natural representation of second and higher order logic in ILC.

where r ranges over untyped lambda terms. Now if $r = \mathsf{I}$, then this translation is the propositions-as-types interpretation; if, on the other hand, one has $r = \mathsf{K}$, then the interpretation becomes an isomorphic version of first order logic denoted by $[A]_\mathsf{I}'$. See [13] and [39] for these results. A short introduction to ILC (in its combinatory version) can be found in [6, Appendix B].

REFERENCES

[1] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum (editors), ***Handbook of logic in computer science, Volume 2: Background: Computational structures***, Oxford University Press, 1992.

[2] W. Ackermann, *Zum Hilbertschen Aufbau der reellen Zahlen*, ***Mathematische Annalen***, vol. 99 (1928), pp. 118–133.

[3] A. W. Appel, ***Compiling with continuations***, Cambridge University Press, 1992.

[4] D. Aspinall and A. Compagnoni, *Subtyping dependent types*, ***Proceedings of the 11th annual symposium on logic in computer science*** (New Brunswick, New Jersey) (E. Clarke, editor), IEEE Computer Society Press, July 1996, pp. 86–97.

[5] J. W. Backus, *Can programming be liberated from the von Neuman style?*, ***Comm. ACM***, vol. 21 (1978), pp. 613–641.

[6] H. P. Barendregt, ***The lambda calculus: its syntax and semantics***, revised ed., North-Holland, Amsterdam, 1984.

[7] ———, *Theoretical pearls: Self-interpretation in lambda calculus*, ***Journal of Functional Programming***, vol. 1 (1991), no. 2, pp. 229–233.

[8] ———, *Lambda calculi with types*, 1992, in [1], pp. 117–309.

[9] ———, *Discriminating coded lambda terms*, ***From universal morphisms to megabytes: A Baayen space-odyssey*** (K. R. Apt, A. A. Schrijver, and N. M. Temme, editors), CWI, Kruislaan 413, 1098 SJ Amsterdam, 1994, pp. 141–151.

[10] ———, *Enumerators of lambda terms are reducing constructively*, ***Annals of Pure and Applied Logic***, vol. 73 (1995), pp. 3–9.

[11] ———, *The quest for correctness*, ***Images of SMC research***, Stichting Mathematisch Centrum, P.O. Box 94079, 1090 GB Amsterdam, 1996, pp. 39–58.

[12] H. P. Barendregt and E. Barendsen, *Efficient computations in formal proofs*, to appear, 1997.

[13] H. P. Barendregt, M. Bunder, and W. Dekkers, *Systems of illative combinatory logic complete for first order propositional and predicate calculus*, ***Journal of Symbolic Logic***, vol. 58 (1993), no. 3, pp. 89–108.

[14] E. Barendsen and J. E. W. Smetsers, *Conventional and uniqueness typing in graph rewrite systems (extended abstract)*, 1993, in [105], pp. 41–51.

[15] ———, *Uniqueness typing for functional languages with graph rewriting semantics*, to appear in ***Mathematical Structures in Computer Science***, 1997.

[16] M. J. Beeson, ***Foundations of constructive mathematics***, Springer-Verlag, Berlin, 1980.

[17] J. F. A. K. van Benthem, ***Language in action: Categories, lambdas and dynamic logic***, Studies in Logic and the Foundations of Mathematics, vol. 130, North-Holland, Amsterdam, 1991.

[18] L. S. van Benthem Jutting, *Checking Landau's "Grundlagen" in the AUTOMATH system*, ***Ph.D. thesis***, Eindhoven University of Technology, 1977.

[19] A. Berarducci and C. Böhm, *A self-interpreter of lambda calculus having a normal form*, ***Lecture Notes in Computer Science***, vol. 702 (1993), pp. 85–99.

[20] M. Bezem and J. F. Groote (editors), *Typed lambda calculi and applications, TLCA'93*, Lecture Notes in Computer Science, vol. 664, Berlin and New York, Springer-Verlag, 1993.

[21] C. Böhm, *The CUCH as a formal and description language*, ***Annual review in automatic programming*** (Richard Goodman, editor), vol. 3, Pergamon Press, Oxford, 1963, pp. 179–197.

[22] C. Böhm and A. Berarducci, *Automatic synthesis of typed λ-programs on term algebras*, ***Theoretical Computer Science***, vol. 39 (1985), pp. 135–154.

[23] C. Böhm and W. Gross, *Introduction to the CUCH*, ***Automata theory*** (E. R. Caianiello, editor), Academic Press, New York, 1966, pp. 35–65.

[24] C. Böhm, A. Piperno, and S. Guerrini, *Lambda-definition of function(al)s by normal forms*, ***Esop'94*** (Berlin) (D. Sanella, editor), vol. 788, Springer-Verlag, 1994, pp. 135–154.

[25] R. B. Braithwaite (editor), ***F. P. Ramsay: The foundations of mathematics and other logical essays***, Routledge & Kegan Paul, London, 1960.

[26] N. G. de Bruijn, *The mathematical language AUTOMATH, its usage and some of its extensions*, ***Symposium on automatic demonstration*** (Berlin and New York) (M. Laudet, D. Lacombe, and M. Schuetzenberger, editors), Lecture Notes in Mathematics, vol. 125, Springer-Verlag, 1970, pp. 29–61, also in [88], pp. 73–100.

[27] ———, *Reflections on Automath*, Eindhoven University of Technology, 1990, also in [88], pp. 201–228.

[28] A. Church, *An unsolvable problem of elementary number theory*, ***American Journal of Mathematics***, vol. 58 (1936), pp. 354–363.

[29] ———, *A formulation of the simple theory of types*, ***Journal of Symbolic Logic***, vol. 5 (1940), pp. 56–68.

[30] ———, ***The calculi of lambda conversion***, Princeton University Press, 1941.

[31] A. Church and J. B. Rosser, *Some properties of conversion*, ***Transactions of the American Mathematical Society***, vol. 39 (1936), pp. 472–482.

[32] W. Clinger and J. Rees (editors), ***Revised report on the algorithmic language Scheme***, vol. IV, LISP Pointers, no. 3, 1991.

[33] T. Coquand and G. Huet, *The calculus of constructions*, ***Information and Computation***, vol. 76 (1988), no. 2/3, pp. 95–120.

[34] G. Cousineau, P.-L. Curien, and M. Mauny, *The categorical abstract machine*, ***Science of Computer Programming***, vol. 8 (1987), no. 2, pp. 173–202.

[35] P.-L. Curien, ***Categorical combinators, sequential algorithms, and functional programming***, Research Notes in Theoretical Computer Science, Pitman, London, 1986.

[36] H. B. Curry, *Grundlagen der kombinatorischen Logik*, ***American Journal of Mathematics***, vol. 52 (1930), pp. 509–536, 789–834, in German.

[37] ———, *Functionality in combinatory logic*, ***Proceedings of the National Academy of Science of the USA***, vol. 20 (1934), pp. 584–590.

[38] ———, *Modified basic functionality in combinatory logic*, ***Dialectica***, vol. 23 (1969), pp. 83–92.

[39] W. Dekkers, M. Bunder, and H. P. Barendregt, *Completeness of the propositions-as-types interpretation of intuitionistic logic into illative combinatory logic*, ***Journal of Symbolic Logic*** (1997), to appear.

[40] M. C. J. D. van Eekelen and M. J. Plasmeijer, ***Functional programming and parallel graph rewriting***, Addison-Wesley, Reading, Massachusetts, 1993.

[41] H. Elbers, *Personal communication*, 1996.

[42] Euclid, *The elements*, 325 B.C. English translation in [55], 1956.

[43] S. Feferman, *A language and axioms for explicit mathematics*, ***Proof theory symposium*** (Berlin) (J. H. Diller and G. H. Müller, editors), Lecture Notes in Mathematics, vol. 500, Springer-Verlag, 1975, pp. 87–139.

[44] ———, *Definedness*, ***Erkentniss***, vol. 43 (1995), pp. 295–320.

[45] L. T. F. GAMUT, ***Logic, language and meaning***, Chicago University Press, Chicago, 1992.

[46] R. O. GANDY, *Church's Thesis and principles for mechanisms*, ***The Kleene symposium***, North-Holland Publishing Company, Amsterdam, 1980, pp. 123–148.

[47] G. GENTZEN, *Investigations into logical deduction*, in [111], 1969.

[48] ———, *Untersuchungen über das logische Schliessen*, ***Mathematische Zeitschrift***, vol. 39 (1935), pp. 176–210, 405–431, also available in [111], pp 68–131.

[49] J.-Y. GIRARD, *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*, ***Ph.D. thesis***, Universite Paris VII, 1972.

[50] ———, *Linear logic: its syntax and semantics*, ***Advances in linear logic*** (J.-Y. Girard, Y. Lafont, and L. Regnier, editors), London Mathematical Society Lecture Note Series, Cambridge University Press, 1995, available by anonymous ftp from lmd.univ-mrs.fr as /pub/girard/Synsem.ps.Z.

[51] J-Y. GIRARD, Y. G. A. LAFONT, and P. TAYLOR, ***Proofs and types***, Cambridge Tracts in Theoretical Computer Science, vol. 7, Cambridge University Press, 1989.

[52] A. D. GORDON, ***Functional programming and Input/Output***, Distinguished Dissertations in Computer Science, Cambridge University Press, 1994.

[53] K. GRUE, *Map theory*, ***Theoretical Computer Science*** (1992), pp. 1–133.

[54] C. A. GUNTER and D. S. SCOTT, *Semantic domains*, ***Handbook of theoretical computer science***, vol. B, in [78], 1990, pp. 633–674.

[55] T. L. HEATH, ***The thirteen books of Euclid's elements***, Dover Publications, Inc., New York, 1956.

[56] J. van Heijenoort (editor), ***From Frege to Gödel: A source book in mathematical logic, 1879–1931***, Harvard University Press, Cambridge, Massachusetts, 1967.

[57] P. HENDERSON, ***Functional programming: Application and implementation***, Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

[58] D. HILBERT and W. ACKERMANN, ***Grundzüge der theoretischen logik***, first ed., Die Grundlehren der Mathematischen Wissenschaften in Einzeldars tellungen, Band XXVII, Springer-Verlag, Berlin and New York, 1928.

[59] R. HINDLEY, *The principal type-scheme of an object in combinatory logic*, ***Transactions of the American Mathematical Society***, vol. 146 (1969), pp. 29–60.

[60] A. HODGES, ***The enigma of intelligence***, Unwin paperbacks, London, 1983.

[61] M. HOFMANN, *A simple model for quotient types*, ***Typed lambda calculi and applications*** (Berlin and New York), Lecture Notes in Computer Science, Springer-Verlag, 1977, pp. 216–234.

[62] P. HUDAK et al., *Report on the programming language Haskell: A non-strict, purely functional language* (*Version* 1.2), ***ACM SIGPLAN Notices***, vol. 27 (1992), no. 5, pp. Ri–Rx, R1–R163.

[63] R. J. M. HUGHES, *The design and implementation of programming languages*, ***Ph.D. thesis***, University of Oxford, 1984.

[64] ———, *Why functional programming matters*, ***The Computer Journal***, vol. 32 (1989), no. 2, pp. 98–107.

[65] K. E. IVERSON, ***A programming language***, Wiley, New York, 1962.

[66] T. JOHNSSON, *Efficient compilation of lazy evaluation*, ***SIGPLAN Notices***, vol. 19 (1984), no. 6, pp. 58–69.

[67] S. C. KLEENE, *Lambda-definability and recursiveness*, ***Duke Mathematical Journal***, vol. 2 (1936), pp. 340–353.

[68] ———, ***Introduction to metamathematics***, The University Series in Higher Mathematics, D. Van Nostrand Comp., New York, Toronto, 1952.

[69] ———, *Reminiscences of logicians*, ***Algebra and logic* (*Fourteenth summer res. inst., Austral. Math. Soc., Monash Univ., Clayton,* 1974)** (J. N. Crossley, editor), Lecture Notes in Mathematics, vol. 450, Springer-Verlag, Berlin and New York, 1975, pp. 1–62.

[70] ———, *Origins of recursive function theory*, ***Annals of the History of Computing***, vol. 3 (1981), no. 1, pp. 52–67.

[71] S. C. Kleene and J. B. Rosser, *The inconsistency of certain formal logics*, ***Annals of Mathematics***, vol. 36 (1935), pp. 630–636.

[72] C. P. J. Koymans, *Models of the lambda calculus*, ***Information and Control***, vol. 52 (1982), no. 3, pp. 306–323.

[73] G. Kreisel, *Church's thesis: A kind of reducibility axiom for constructive mathematics*, in [86], pp. 121–150.

[74] ———, *The formalist-positivist doctrine of mathematical precision in the light of experience*, ***L'age de la Science***, vol. 3 (1970), pp. 17–46.

[75] J. Kuper, *An axiomatic theory for partial functions*, ***Information and Computation*** (1993), pp. 104–150.

[76] E. Landau, ***Grundlagen der analysis***, third ed., Chelsea Publishing Company, 1960.

[77] P. J. Landin, *The mechanical evaluation of expressions*, ***The Computer Journal***, vol. 6 (1964), no. 4, pp. 308–320.

[78] J. van Leeuwen (editor), ***Handbook of theoretical computer science***, vol. A, B, North-Holland, MIT-Press, 1990.

[79] D. Leivant, *Reasoning about functional programs and complexity classes associated with type disciplines*, ***24th annual symposium on foundations of computer science***, IEEE, 1983, pp. 460–469.

[80] Z. Luo and R. Pollack, *The LEGO proof development system: A user's manual*, ***Technical Report ECS-LFCS-92-211***, University of Edinburgh, may 1992.

[81] P. Martin-Löf, ***Intuitionistic type theory***, Studies in Proof Theory, Bibliopolis, Napoli, 1984.

[82] Yu.V. Matijasevič, *On recursive unsolvability of hilbert's tenth problem*, ***Fourth international congress for logic, methodology and philosophy of science***, Studies in Logic and the Foundations of Mathematics, vol. 74, North-Holland, Amsterdam, 1971, pp. 89–110.

[83] J. McCarthy et al., ***Lisp* 1.5 *programmer's manual***, MIT Press, Cambridge, Massachusetts, 1962.

[84] R. Milner, *A theory of type polymorphism in programming*, ***Journal of Computer and System Sciences***, vol. 17 (1978), pp. 348–375.

[85] T.Æ. Mogensen, *Theoretical pearls: Efficient self-interpretation in lambda calculus*, ***Journal of Functional Programming***, vol. 2 (1992), no. 3, pp. 345–364.

[86] J. Myhill, R. E. Vesley, and A. Kino (editors), ***Intuitionism and proof theory***, Studies in Logic and the Foundations of Mathematics, North-Holland, Amsterdam, 1970.

[87] G. Nadathur and D. Miller, *An overview of λProlog*, ***Logic programming: Proceedings of the fifth international conference and symposium, Volume 1*** (Cambridge, Massachusetts) (Robert A. Kowalski and Kenneth A. Bowen, editors), MIT Press, August 1988, pp. 810–827.

[88] R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer (editors), ***Selected papers on automath***, Studies in Logic and the Foundations of Mathematics, vol. 133, North-Holland, Amsterdam, 1994.

[89] J. von Neumann, *Eine axiomatisierung der mengenlehre*, ***Journal für die Reine und Angewandte Mathematik***, vol. 154 (1925), pp. 219–240.

[90] M. Oostdijk, *Proof by calculation*, ***Master's thesis***, **385**, Universitaire School voor Informatica, Catholic University Nijmegen, 1996.

[91] C. Paulin-Mohring, *Inductive definitions in the system Coq; rules and properties*,

1993, in [20], pp. 328–345.

[92] S. L. PEYTON JONES and P. WADLER, *Imperative functional programming*, ***Conference record of the twentieth annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, Charleston, South Carolina, January 10-13, 1992***, ACM Press, 1993, pp. 71–84 (English).

[93] G. D. PLOTKIN, *Call-by-name, call-by-value and the λ-calculus*, ***Theoretical Computer Science***, vol. 1 (1975), pp. 125–159.

[94] H. POINCARÉ, ***La science et l'hypothèse***, Flammarion, Paris, 1902.

[95] F. P. RAMSEY, *The foundations of mathematics*, ***Proceedings of the London Mathematical Society, Series 2***, vol. 25 (1925), pp. 338–384, translated in [25].

[96] J. C. REYNOLDS, *Definitional interpreters for higher-order programming languages*, ***Proceedings of 25th ACM national conference*** (Boston, Massachusetts), 1972, pp. 717–740.

[97] ———, *The discoveries of continuations*, ***LISP and Symbolic Computation***, vol. 6 (1993), no. 3/4, pp. 233–247.

[98] R. M. ROBINSON, *The theory of classes—a modification of von Neumann's system*, ***Journal of Symbolic Logic***, vol. 2 (1937), pp. 29–36.

[99] J. B. ROSSER, *Highlights of the history of lambda-calculus*, ***ACM symposium on Lisp and functional programming*** (Pennysylvania), ACM Press, August 1982, pp. 216–225.

[100] B. A. W. RUSSELL and A. N. WHITEHEAD, ***Principia mathematica***, vol. 1 and 2, Cambridge University Press, 1910–13.

[101] H. SCHWICHTENBERG, *Definierbare Funktionen im λ-Kalkül mit Typen*, ***Archief für Mathematische Logik***, vol. 25 (1976), pp. 113–114.

[102] D. S. SCOTT, *Constructive validity*, ***Symposium on automated demonstration*** (D. Lacombe M. Laudet and M. Schuetzenberger, editors), Lecture Notes in Mathematics, vol. 125, Springer-Verlag, Berlin, 1970, pp. 237–275.

[103] ———, *Continuous lattices*, ***Toposes, algebraic geometry, and logic*** (F. W. Lawvere, editor), Lecture Notes in Mathematics, vol. 274, Springer-Verlag, Berlin and New York, 1972, pp. 97–136.

[104] P. SEVERI and E. POLL, *Pure type systems with definitions*, ***Proceedings of LFCS'94*** (Berlin and New York) (A. Nerode and Yu.V. Matijasevič, editors), Lecture Notes in Computer Science, vol. 813, LFCS'94, St. Petersburg, Springer-Verlag, 1994, pp. 316–328.

[105] R. K. Shyamasundar (editor), ***Proceedings of the 13th conference on foundations of software technology and theoretical computer science***, Lecture Notes in Computer Science, vol. 761, Berlin and New York, Bombay, India, Springer-Verlag, 1993.

[106] T. SKOLEM, *Begründung der elementaren Arithmetik durch die rekurrierende Denkweise ohne Anwendung scheinbarer Verënderlichen mit unendlichem Ausdehnungsbereich*, ***Videnskapsselskapets skrifter, I. Matematisk-naturvidenskabelig klasse***, vol. 6 (1923), English translation in [56], pp. 302–333.

[107] M. B. SMYTH and G. D. PLOTKIN, *The category-theoretic solution of recursive domain equations*, ***SIAM Journal on Computing***, vol. 11 (1982), no. 4, pp. 761–783.

[108] R. STATMAN, *The typed lambda calculus is not elementary recursive*, ***Theoretical Computer Science***, vol. 9 (1979), pp. 73–81.

[109] GUY L. STEELE JR., *Rabbit: A compiler for Scheme*, ***Technical Report AI-TR-474***, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[110] ———, ***Common Lisp: The language***, Digital Press, 1984.

[111] M. E. Szabo (editor), ***The collected papers of Gerhard Gentzen***, North-Holland, Amsterdam, 1969.

[112] A. S. Troelstra (editor), ***Metamathematical investigation of intuitionistic arithmetic and analysis***, Lecture Notes in Mathematics, vol. 344, Springer-Verlag, Berlin and New York,

1973.

[113] A. M. TURING, *On computable numbers with an application to the Entscheidungsproblem*, ***Proceeding of the London Mathematical Society. Second Series.***, vol. 42 (1936), pp. 230–265.

[114] ———, *Computability and lambda definability*, ***Journal of Symbolic Logic***, vol. 2 (1937), pp. 153–163.

[115] D. A. TURNER, *The SASL language manual*, 1976.

[116] ———, *A new implementation technique for applicative languages*, ***Software—Practice and Experience***, vol. 9 (1979), pp. 31–49.

[117] ———, *The semantic elegance of functional languages*, ***Proceedings of the ACM/MIT conference on functional languages and computer architecture***, ACM Press, Pennsylvania, 1981, pp. 85–92.

[118] ———, *Miranda—a non-strict functional language with polymorphic types*, ***Functional programming languages and computer architectures*** (Berlin and New York) (J. P. Jouannaud, editor), Lecture Notes in Computer Science, vol. 201, Springer-Verlag, 1985, pp. 1–16.

[119] C. WADSWORTH, *Semantics and pragmatics of the lambda-calculus*, ***D. Phil thesis***, University of Oxford, Programming Research Group, Oxford, U.K., 1971.

COMPUTING SCIENCE INSTITUTE
NIJMEGEN UNIVERSITY
THE NETHERLANDS

E-mail: henk@cs.kun.nl