Constructive mathematics and computer programming†

By P. Martin-Löf

Department of Mathematics, University of Stockholm, Box 6701, *S*-113 85 *Stockholm, Sweden*

If programming is understood not as the writing of instructions for this or that computing machine but as the design of methods of computation that it is the computer's duty to execute (a difference that Dijkstra has referred to as the difference between computer science and computing science), then it no longer seems possible to distinguish the discipline of programming from constructive mathematics. This explains why the intuitionistic theory of types (Martin-Löf 1975 In *Logic Colloquium 1973* (ed. H. E. Rose & J. C. Shepherdson), pp. 73–118. Amsterdam: North-Holland), which was originally developed as a symbolism for the precise codification of constructive mathematics, may equally well be viewed as a programming language. As such it provides a precise notation not only, like other programming languages, for the programs themselves but also for the tasks that the programs are supposed to perform. Moreover, the inference rules of the theory of types, which are again completely formal, appear as rules of correct program synthesis. Thus the correctness of a program written in the theory of types is proved formally at the same time as it is being synthesized.

During the period of just over thirty years that has elapsed since the first electronic computers were built, programming languages have developed from various machine codes and assembly languages, now referred to as low level languages, to high level languages, like FORTRAN, ALGOL 60 and 68, LISP and PASCAL. The virtue of a machine code is that a program written in it can be directly read and executed by the machine. Its weakness is that the structure of the code reflects the structure of the machine so closely as to make it unusable for the instruction of any other machine and, what is more serious, very difficult to understand for a human reader, and therefore error prone. With a high level language, it is the other way round. Its weakness is that a program written in it has to be compiled, that is, translated into the code of a particular machine, before it can be executed by it. But there is ample compensation for this by having a language in which the thought of the programmer can be expressed without too much distortion and understood by someone who knows very little about the structure of the hardware, but does know some English and mathematics. The distinction between low and high level programming languages is of course related to available hardware. It may well be possible to turn what is now regarded as a high level programming language into machine code by the invention of new hardware.

Parallel to the development from low to high level programming languages, there has been a change in one's understanding of the programming activity itself. It used to be looked (down) upon as the rather messy job of instructing this or that physically existing machine, by cunning tricks, to perform computational tasks widely surpassing our own physical powers, something

† This paper was previously published in *Proceedings of the Sixth International Congress for Logic, Methodology and Philosophy of Science* (ed. L. J. Cohen, J. Łos, H. Pfeiffer & K.-P. Podewski), pp. 153–175 (1982). It is reprinted here by permission of the North-Holland Publishing Company, Amsterdam.

that might appeal to people with a liking for crossword puzzles or chess problems. But it has grown into the discipline of designing programs for various (numerical as well as non-numerical) computational tasks, programs that have to be written in a formally precise notation to make automatic execution possible. Whether or not machines have been built or compilers have been written by means of which they can be physically implemented is of no importance as long as questions of efficiency are ignored. What matters is merely that it has been laid down precisely how the programs are to be executed or, what amounts to the same, that it has been specified how a machine for the execution of the programs would have to function. This change of programming, which made Dijkstra (1976) change the terminology from computer science to computing science, would not have been possible without the creation of high level languages of a sufficiently clean logical structure. It has made programming an activity akin in rigour and beauty to that of proving mathematical theorems. (This analogy is actually exact in a sense that will become clear later.)

While maturing into a science, programming has developed a conceptual machinery of its own in which, besides the notion of program itself, the notions of data structure and data type occupy central positions. Even in FORTRAN, there were two types of variables, namely integer and floating point variables, the type of a variable being determined by its initial letter. In ALGOL 60, there was added to the two types **integer** and **real** the third type **Boolean**, and the association of the types with the variables was made both more practical and logical by means of type declarations. However, it was only through Professor Hoare's Notes on Data Structuring (Dahl *et al.* 1972) that the notion of type was introduced into programming in a systematic way. In addition to the three types of ALGOL 60, there now appeared types defined by enumeration, Cartesian products, discriminated unions, array types, power types and various recursively defined types. All these new forms of data types were subsequently incorporated into the programming language PASCAL by Wirth (1971). The left column of table 1, which shows some of the key notions of programming together with their mathematical counterparts, uses notation from ALGOL 60 and PASCAL.

TABLE 1. KEY NOTIONS OF PROGRAMMING WITH MATHEMATICAL COUNTERPARTS

programming	mathematics
program, procedure, algorithm	function
input	argument
output, result	value
$x := e$	$x = e$
$S_1; S_2$	composition of functions
if B **then** S_1 **else** S_2	definition by cases
while B **do** S	definition by recursion
data structure	element, object
data type	set, type
value of a data type	element of a set, object of a type
$a:A$	$a \in A$
integer	Z
real	R
Boolean	$\{0, 1\}$
$(c_1, \ldots, c_n)$	$\{c_1, \ldots, c_n\}$
array [I] **of** T	T^I, $I \to T$
record $s_1:T_1; s_2:T_2$ **end**	$T_1 \times T_2$
record case $s:(c_1, c_2)$ **of** $c_1:(s_1:T_1); c_2:(S_2:T_2)$ **end**	$T_1 + T_2$
set of T	$\{0, 1\}^T$, $T \to \{0, 1\}$

As can be seen from table 1, or from recent programming texts with their snippets of set theory prefaced to the corresponding programming language constructions, the whole conceptual apparatus of programming mirrors that of modern mathematics (set theory, that is, not geometry) and yet is supposed to be different from it. The reason for this curious situation is, I think, that the mathematical notions have gradually received an interpretation, the interpretation that we refer to as classical, which makes them unusable for programming. Fortunately, I do not need to enter the philosophical debate as to whether the classical interpretation of the primitive logical and mathematical notions (proposition, truth, set, element, function, etc.) is sufficiently clear, because this much is at least clear, that if a function is defined as a binary relation satisfying the usual existence and unicity conditions, whereby classical reasoning is allowed in the existence proof, or a set of ordered pairs satisfying the corresponding conditions, then a function cannot be the same type of thing as a computer program. Similarly, if a set is understood in the sense of Zermelo, as a member of the cumulative hierarchy, then a set cannot be the same kind of thing as a data type.

Now, it is the contention of the intuitionists (or constructivists, I shall use these terms synonymously) that the basic mathematical notions, above all the notion of function, ought to be interpreted in such a way that the cleavage between mathematics (classical mathematics, that is) and programming that we are witnessing at present disappears. For the mathematical notions of function and set, it is not so much a question of providing them with new meanings as of restoring old ones, whereas the logical notions of proposition, proof, truth, etc. are given genuinely new interpretations. It was Brouwer who realized the necessity of so doing: the true source of the uncomputable functions of classical mathematics is not the axiom of choice (which *is* valid intuitionistically) but the law of excluded middle and the law of indirect proof. Had it not been possible to interpret the logical notions in such a way as to validate the axiom of choice, the prospects of constructive mathematics would have been dismal.

The difference, then, between constructive mathematics and programming does not concern the primitive notions of the one or the other, because they are essentially the same, but lies in the programmer's insistence that his programs be written in a formal notation so that they can be read and executed by a machine, whereas, in constructive mathematics as practised by Bishop (1967), for example, the computational procedures (programs) are normally left implicit in the proofs, so that considerable further work is needed to bring them into a form that makes them fit for mechanical execution.

What I have just said about the close connection between constructive mathematics and programming explains why the intuitionistic type theory (Martin-Löf 1975), which I began to develop solely with the philosophical motive of clarifying the syntax and semantics of intuitionistic mathematics, may equally well be viewed as a programming language. But for a few concluding remarks, the rest of this paper will be devoted to a fairly complete, albeit condensed, description of this language, with emphasis on its character of programming language. As such, it resembles ALGOL 68 and PASCAL in its typing facilities, whereas the way the programs are written and executed makes it more reminiscent of LISP.

The expressions of the theory of types are formed from variables

$$x, y, z, \ldots$$

by means of various forms of expression

$$(Fx_1, \ldots, x_n)\,(a_1, \ldots, a_m).$$

In an expression of such a form, not all of the variables $x_1, \ldots, x_n$ need become bound in all of the parts $a_1, \ldots, a_m$. Thus, for each form of expression, it must be laid down what variables become bound in what parts. For example,

$$\int_a^b f\,dx$$

is a form of expression (Ix) (a, b, f), with m = 3 and n = 1, which binds all free occurrences of the single variable x in the third part f, and

$$\frac{df}{dx}(a)$$

is a form of expression (Dx) (a, f), with m = 2 and n = 1, which binds all free occurrences of the variable x in the second part f.

I shall call an expression, in whatever notation, canonical or normal if it is already fully evaluated, which is the same as saying that it has itself as value. Thus, in decimal arithmetic,

$$0, 1, \ldots, 9, 10, 11, \ldots$$

are canonical (normal) expressions, whereas

$$2+2, 2\times 2, 2^2, 3!, 10^{10^{10}}, \ldots$$

are not. An arbitrarily formed expression need not have a value, but, if an expression has a value, then that value is necessarily canonical. This may be expressed by saying that evaluation is idempotent. When you evaluate the value of an expression, you get that value back.

In the theory of types, it depends only on the outermost form of an expression whether it is canonical or not. So there are certain forms of expression, which I shall call canonical forms, such that an expression of one of those forms has itself as value, and there are other, non-canonical forms for which it is laid down in some other way how an expression of such a form is evaluated. What I call canonical and non-canonical forms of expression correspond to the constructors and selectors, respectively, of Landin (1964). In the context of programming, they might also aptly be called data and program forms, respectively. Table 2 displays the primitive forms of expression used in the theory of types. New primitive forms of expression may of course be added when there is need of them.

The conventions as to what variables become bound in what parts are as follows. Free occurrences of x in B become bound in $(\Pi x \in A)\,B$, $(\Sigma x \in A)\,B$ and $(W x \in A)\,B$. Free occurrences

TABLE 2. PRIMITIVE FORMS OF EXPRESSION USED IN THE THEORY OF TYPES

canonical	non-canonical
$(\Pi x \in A)\,B$, $(\lambda x)\,b$	$c(a)$
$(\Sigma x \in A)\,B$, (a, b)	$(Ex, y)\,(c, d)$
$A+B$, $i(a)$, $j(b)$	$(Dx, y)\,(c, d, e)$
$I(A, a, b)$, r	$J(c, d)$
N_0	$R_0(c)$
N_1, 0_1	$R_1(c, c_0)$
N_2, 0_2, 1_2	$R_2(c, c_0, c_1)$
⋮	⋮
N, 0, a'	$(Rx, y)\,(c, d, e)$
$(W x \in A)\,B$, $\sup(a, b)$	$(Tx, y, z)\,(c, d)$
U_0, U_1, ...	

of x in b become bound in (λx) b. Free occurrences of x and y in d become bound in (Ex, y) (c, d). Free occurrences of x in d and y in e become bound in (Dx, y) (c, d, e). Free occurrences of x and y in e become bound in (Rx, y) (c, d, e). And, finally, free occurrences of x, y and z in d become bound in (Tx, y, z) (c, d).

Expressions of the various forms displayed in table 2 are evaluated according to the following rules. I use

$$b(a_1, \ldots, a_n/x_1, \ldots, x_n)$$

to denote the result of simultaneously substituting the expressions $a_1, \ldots, a_n$ for the variables $x_1, \ldots, x_n$ in the expression b. Substitution is the process whereby a program is supplied with its input data, which need not necessarily be in evaluated form.

An expression of canonical form has itself as value. This has already been intimated.

To execute c(a), first execute c. If you get (λx) b as result, then continue by executing b(a/x). Thus c(a) has value d if c has value (λx) b and b(a/x) has value d.

To execute (Ex, y) (c, d), first execute c. If you get (a, b) as result, then continue by executing d(a, b/x, y). Thus (Ex, y) (c, d) has value e if c has value (a, b) and d(a, b/x, y) has value e.

To execute (Dx, y) (c, d, e), first execute c. If you get i(a) as result, then continue by executing d(a/x). If, on the other hand, you get j(b) as result of executing c, then continue by executing e(b/y) instead. Thus (Dx, y) (c, d, e) has value f if either c has value i(a) and d(a/x) has value f, or c has value j(b) and e(b/y) has value f.

To execute J(c, d), first execute c. If you get r as result, then continue by executing d. Thus J(c, d) has value e if c has value r and d has value e.

To execute $R_n(c, c_0, \ldots, c_{n-1})$, first execute c. If you get m_n as result for some $m = 0, \ldots, n-1$, then continue by executing c_m. Thus $R_n(c, c_0, \ldots, c_{n-1})$ has value d if c has value m_n and c_m has value d for some $m = 0, \ldots, n-1$. In particular, $R_0(c)$ has no value. It corresponds to the statement

abort

introduced by Dijkstra (1976). The pair of forms 0_1 and $R_1(c, c_0)$ together operate in exactly the same way as the pair of forms r and J(c, d). To have them both in the language constitutes a redundancy. $R_2(c, c_0, c_1)$ corresponds to the usual conditional statement

if B **then** S_1 **else** S_2

and $R_n(c, c_0, \ldots, c_{n-1})$ for arbitrary $n = 0, 1, \ldots$ to the statement

with e **do** $\{c_1 : S_1, \ldots, c_n : S_n\}$;

introduced by Hoare (Dahl *et al.* 1972, p. 113) and realized by Wirth in PASCAL as the case statement

case e **of** $c_1 : S_1; \ldots; c_n : S_n$ **end**.

To execute (Rx, y) (c, d, e), first execute c. If you get 0 as result, then continue by executing d. If, on the other hand, you get a′ as result, then continue by executing e(a, (Rx, y) (a, d, e)/x, y) instead. Thus (Rx, y) (c, d, e) has value f if either c has value 0 and d has value f, or c has value a′ and e(a, (Rx, y) (a, d, e)/x, y) has value f. The closest analogue of the recursion form (Rx, y) (c, d, e) in traditional programming languages is the repetitive statement form

while B **do** S.

To execute (Tx, y, z) (c, d), first execute c. If you get sup (a, b) as result, then continue by executing d(a, b, (λv) (Tx, y, z) (b(v), d)/x, y, z). Thus (Tx, y, z) (c, d) has value e if c has value sup (a, b) and d(a, b, (λv) (Tx, y, z) (b(v), d)/x, y, z) has value e. The transfinite recursion form (Tx, y, z) (c, d) has not yet found any applications in programming. It has, as far as I know, no counterpart in other programming languages.

The traditional way of evaluating an arithmetical expression is to evaluate the parts of the expression before the expression itself is evaluated, as shown in the example

$$\begin{array}{c} \underbrace{(3+2)}_{5}!\times 4. \\ \underbrace{\qquad\qquad}_{120} \\ 480 \end{array}$$

Thus, traditionally, expressions are evaluated from within, which in programming has come to be known as the applicative order of evaluation. When expressions are evaluated in this way, it is obvious that an expression cannot have a value unless all its parts have values. Moreover, as was explicitly stated as a principle by Frege, the value (Ger.: Bedeutung) of an expression depends only on the values of its parts. In other words, if a part of an expression is replaced by one that has the same value, the value of the whole expression is left unaffected.

When variable binding forms of expression are introduced, as they are in the theory of types, it is no longer possible, in general, to evaluate the expressions from within. To evaluate (λx) b, for example, we would first have to evaluate b. But b cannot be evaluated, in general, until a value has been assigned to the variable x. In the theory of types, this difficulty has been overcome by reversing the order of evaluation: instead of evaluating the expressions from within, they are evaluated from without. This is known as head reduction in combinatory logic and normal order or lazy evaluation in programming. For example, (λx) b is simply assigned itself as value. The term lazy is appropriate because only as few computation steps are made as are absolutely necessary to bring an expression into canonical form. However, what turns out to be of no significance, it is no longer the case that an expression cannot have a value unless all its parts have values. For example, a′ has itself as value even if a has no value. What is significant, though, is that the principle of Frege's referred to earlier, namely that the value of an expression depends only on the values of its parts, is irretrievably lost. To make the language work in spite of this loss has been one of the most serious difficulties in the design of the theory of types.

So far, I have merely displayed the various forms of expression used in the theory of types and explained how expressions composed out of those forms are evaluated. The inferential or, as termed in combinatory logic, illative part of the language consists of rules for making judgments of the four forms

A is a type,

A and B are equal types,

a is an object of type A,

a and b are equal objects of type A,

abbreviated

$$A \text{ type},$$

$$A = B,$$

$$a \in A,$$

$$a = b \in A,$$

respectively. A judgment of any one of these forms is in general hypothetical, that is, made under assumptions or, to use the terminology of AUTOMATH (de Bruijn 1970), in a context

$$x_1 \in A_1, \ldots, x_n \in A_n.$$

In such a context, it is always the case that A_1 is a type, ..., A_n is a type under the preceding assumptions $x_1 \in A_1, \ldots, x_{n-1} \in A_{n-1}$. When there is a need to indicate explicitly the assumptions of a hypothetical judgment, it will be written

$$A \text{ type } (x_1 \in A_1, \ldots, x_n \in A_n),$$

$$A = B(x_1 \in A_1, \ldots, x_n \in A_n),$$

$$a \in A(x_1 \in A_1, \ldots, x_n \in A_n),$$

$$a = b \in A(x_1 \in A_1, \ldots, x_n \in A_n).$$

These, then, are the full forms of judgment of the theory of types.

The first form of judgment admits not only the readings

A is a type (set),

A is a proposition,

but also, and this is the reading that is most natural when the language is thought of as a programming language,

A is a problem (task).

Correlatively, the third form of judgment may be read not only

a is an object of type (element of the set) A,

a is a proof of the proposition A,

but also

a is a program for the problem (task) A.

The equivalence of the first two readings is the, by now well known, correspondence between propositions and types discovered by Curry (1958) and Howard (1969), whereas the transition from the second to the third is Kolmogorov's (1932) interpretation of propositions as problems or tasks (Ger.: Aufgabe).

The four forms of judgment used in the theory of types should be compared with the three forms of judgment used (although usually not so called) in standard presentations of first order predicate calculus, whether classical or intuitionistic, namely

A is a formula,

A is true,

a is an individual term.

The first of these corresponds to the form A is a type (proposition), the second is obtained from the form a is an object of type (a proof of the proposition) A by suppressing a, and the third is again obtained from the form a is an object of type A, this time by choosing for A the type of individuals.

In explaining what a judgment of one of the above four forms means, I shall first limit myself to assumption-free judgments. Once it has been explained what meanings they carry, the explanations can readily be extended so as to cover hypothetical judgments as well.

A canonical type A is defined by prescribing how a canonical object of type A is formed as well as how two equal canonical objects of type A are formed. There is no limitation on this prescription except that the relation of equality that it defines between canonical objects of type A must be reflexive, symmetric and transitive. If the rules for forming canonical objects as well as equal canonical objects of a certain type are called the introduction rules for that type, we may thus say with Gentzen (1934) that a canonical type (proposition) is defined by its introduction rules. For non-canonical A, a judgment of the form

$$A \text{ is a type}$$

means that A has a canonical type as value.

Two canonical types A and B are equal if a canonical object of type A is also a canonical object of type B and, moreover, equal canonical objects of type A are also equal canonical objects of type B, and vice versa. For arbitrary (not necessarily canonical) types A and B, a judgment of the form

$$A = B$$

means that A and B have equal canonical types as values. This finishes the explanations of what a type is and what it means for two types to be equal.

Let A be a type. Remember that this means that A denotes a canonical type, that is, has a canonical type as value. Then a judgment of the form

$$a \in A$$

means that a has a canonical object of the canonical type denoted by A as value. Of course, this explanation is not comprehensible unless we know that A has a canonical type as value as well as what a canonical object of that type is. But we do know this because of the presupposition that A is a type: it is part of the definition of a canonical type how a canonical object of that type is formed, and hence we cannot know a canonical type without knowing what a canonical object of that type is.

Let A be a type and a and b be objects of type A. Then a judgment of the form

$$a = b \in A$$

means that a and b have equal canonical objects of the canonical type denoted by A as values. This explanation makes sense since A was presupposed to be a type, that is, to have a canonical type as value, and it is part of the definition of a canonical type how equal canonical objects of that type are formed.

These meaning explanations are extended to hypothetical judgments by an induction on the number of assumptions. Let it be given as premises for all of the following four explanations that $x_1 \in A, \ldots, x_n \in A_n$ is a context, that is, that A_1 is a type, $\ldots, A_n$ is a type under the assumptions $x_1 \in A_1, \ldots, x_{n-1} \in A_{n-1}$. By induction hypothesis, we know what this means.

A judgment of the form

$$A \text{ type } (x_1 \in A_1, \ldots, x_n \in A_n)$$

means that

$$A(a_1, \ldots, a_n/x_1, \ldots, x_n) \text{ type}$$

provided

$$a_1 \in A_1, \ldots, a_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}),$$

and, moreover,

$$A(a_1, \ldots, a_n/x_1, \ldots, x_n) = A(b_1, \ldots, b_n/x_1, \ldots, x_n)$$

provided

$$a_1 = b_1 \in A_1, \ldots, a_n = b_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}).$$

Thus it is in the nature of a family of types (propositional function) to be extensional in the sense just described.

Suppose that A and B are types under the assumptions $x_1 \in A_1, \ldots, x_n \in A_n$. Then

$$A = B(x_1 \in A_1, \ldots, x_n \in A_n)$$

means that

$$A(a_1, \ldots, a_n/x_1, \ldots, x_n) = B(a_1, \ldots, a_n/x_1, \ldots, x_n)$$

provided

$$a_1 \in A_1, \ldots, a_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}).$$

From this definition, the extensionality of a family of types and the evident transitivity of equality between types, it also follows that

$$A(a_1, \ldots, a_n/x_1, \ldots, x_n) = B(b_1, \ldots, b_n/x_1, \ldots, x_n)$$

provided

$$a_1 = b_1 \in A_1, \ldots, a_n = b_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}).$$

Let A be a type under the assumptions $x_1 \in A_1, \ldots, x_n \in A_n$. Then

$$a \in A(x_1 \in A_1, \ldots, x_n \in A_n)$$

means that

$$a(a_1, \ldots, a_n/x_1, \ldots, x_n) \in A(a_1, \ldots, a_n/x_1, \ldots, x_n)$$

provided

$$a_1 \in A_1, \ldots, a_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}),$$

and, moreover,

$$a(a_1, \ldots, a_n/x_1, \ldots, x_n) = a(b_1, \ldots, b_n/x_1, \ldots, x_n) \in A(a_1, \ldots, a_n/x_1, \ldots, x_n)$$

provided

$$a_1 = b_1 \in A_1, \ldots, a_n = b_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}).$$

Thus, just as for a family of types, it is in the nature of a function to be extensional in the sense of yielding equal objects of the range type when equal objects of the domain types are substituted for the variables of which it is a function.

Let A be a type and a and b objects of type A under the assumptions $x_1 \in A_1, \ldots, x_n \in A_n$. Then

$$a = b \in A(x_1 \in A_1, \ldots, x_n \in A_n)$$

means that

$$a(a_1, \ldots, a_n/x_1, \ldots, x_n) = b(a_1, \ldots, a_n/x_1, \ldots, x_n) \in A(a_1, \ldots, a_n/x_1, \ldots, x_n)$$

provided

$$a_1 \in A_1, \ldots, a_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}).$$

Again, from this definition, the extensionality of a function and the transitivity of equality between objects of whatever type, there follows the stronger property that

$$a(a_1, \ldots, a_n/x_1, \ldots, x_n) = b(b_1, \ldots, b_n/x_1, \ldots, x_n) \in A(a_1, \ldots, a_n/x_1, \ldots, x_n)$$

provided

$$a_1 = b_1 \in A_1, \ldots, a_n = b_n \in A_n(a_1, \ldots, a_{n-1}/x_1, \ldots, x_{n-1}).$$

This finishes my explanations of what judgments of the four forms used in the theory of types mean in the presence of assumptions.

Now to the rules of inference or proof rules, as they are called in programming. They will be presented in natural deduction style, suppressing as usual all assumptions other than those that are discharged by an inference of the particular form under consideration. Moreover, in those rules whose conclusion has one of the forms $a \in A$ and $a = b \in A$, only those premises will be explicitly shown that have these very same forms. This is in agreement with the practice of writing, say, the rules of disjunction introduction in predicate calculus simply

$$\frac{A \text{ true}}{A \vee B \text{ true}} \qquad \frac{B \text{ true}}{A \vee B \text{ true}}$$

without showing explicitly the premises that A and B are formulas. For each of the rules of inference, the reader is asked to try to make the conclusion evident to himself on the presupposition that he knows the premises. This does not mean that further verbal explanations are of no help in bringing about an understanding of the rules, only that this is not the place for such detailed explanations. But there are also certain limits to what verbal explanations can do when it comes to justifying axioms and rules of inference. In the end, everybody must understand for himself.

General rules

Reflexivity

$$\frac{a \in A}{a = a \in A} \qquad \frac{A \text{ type}}{A = A}$$

Symmetry

$$\frac{a = b \in A}{b = a \in A} \qquad \frac{A = B}{B = A}$$

Transitivity

$$\frac{a = b \in A \quad b = c \in A}{a = c \in A} \qquad \frac{A = B \quad B = C}{A = C}$$

Equality of types

$$\frac{a \in A \quad A = B}{a \in B} \qquad \frac{a = b \in A \quad A = B}{a = b \in B}$$

Substitution

$$\frac{(x \in A)}{a \in A \quad B \text{ type}} \qquad \frac{(x \in A)}{a = c \in A \quad B = D}$$
$$\frac{B(a/x) \text{ type}}{B(a/x) = D(c/x)}$$

$$\frac{(x \in A)}{a \in A \quad b \in B} \qquad \frac{(x \in A)}{a = c \in A \quad b = d \in B}$$
$$\frac{}{b(a/x) \in B(a/x)} \qquad \frac{}{b(a/x) = d(c/x) \in B(a/x)}$$

Assumption

$$x \in A$$

Cartesian product of a family of types

Π-*formation*

$$\frac{(x \in A)}{A \text{ type} \quad B \text{ type}} \qquad \frac{(x \in A)}{A = C \quad B = D}$$
$$\frac{}{(\Pi x \in A)\, B \text{ type}} \qquad \frac{}{(\Pi x \in A)\, B = (\Pi x \in C)\, D}$$

Π-*introduction*

$$\frac{(x \in A)}{b \in B} \qquad \frac{(x \in A)}{b = d \in B}$$
$$\frac{}{(\lambda x)\, b \in (\Pi x \in A)\, B} \qquad \frac{}{(\lambda x)\, b = (\lambda x)\, d \in (\Pi x \in A)\, B}$$

Π-*elimination*

$$\frac{c \in (\Pi x \in A)\, B \quad a \in A}{c(a) \in B(a/x)} \qquad \frac{c = f \in (\Pi x \in A)\, B \quad a = d \in A}{c(a) = f(d) \in B(a/x)}$$

Π-equality

$$\frac{\begin{array}{c}(x\in A)\\ a\in A \quad b\in B\end{array}}{((\lambda x)\,b)\,(a) = b(a/x)\in B(a/x)} \qquad \frac{c\in(\Pi x\in A)\,B}{(\lambda x)\,(c(x)) = c\in(\Pi x\in A)\,B}$$

Disjoint union of a family of types

Σ-formation

$$\frac{\begin{array}{c}(x\in A)\\ A \text{ type} \quad B \text{ type}\end{array}}{(\Sigma x\in A)\,B \text{ type}} \qquad \frac{\begin{array}{c}(x\in A)\\ A = C \quad B = D\end{array}}{(\Sigma x\in A)\,B = (\Sigma x\in C)\,D}$$

Σ-introduction

$$\frac{a\in A \quad b\in B(a/x)}{(a,b)\in(\Sigma x\in A)\,B} \qquad \frac{a = c\in A \quad b = d\in B(a/x)}{(a,b) = (c,d)\in(\Sigma x\in A)\,B}$$

Σ-elimination

$$\frac{\begin{array}{c}(x\in A, y\in B)\\ c\in(\Sigma x\in A)\,B \quad d\in C((x,y)/z)\end{array}}{(Ex,y)\,(c,d)\in C(c/z)}$$

$$\frac{\begin{array}{c}(x\in A, y\in B)\\ c = e\in(\Sigma x\in A)\,B \quad d = f\in C((x,y)/z)\end{array}}{(Ex,y)\,(c,d) = (Ex,y)\,(e,f)\in C(c/z)}$$

Σ-equality

$$\frac{\begin{array}{c}(x\in A, y\in B)\\ a\in A \quad b\in B(a/x) \quad d\in C((x,y)/z)\end{array}}{(Ex,y)\,((a,b),d) = d(a,b/x,y)\in C((a,b)/z)}$$

Disjoint union of two types

+-formation

$$\frac{A \text{ type} \quad B \text{ type}}{A+B \text{ type}} \qquad \frac{A = C \quad B = D}{A+B = C+D}$$

+-introduction

$$\frac{a\in A}{i(a)\in A+B} \qquad \frac{a = c\in A}{i(a) = i(c)\in A+B}$$

$$\frac{b\in B}{j(b)\in A+B} \qquad \frac{b = d\in B}{j(b) = j(d)\in A+B}$$

+ -*elimination*

$$\frac{c \in A + B \quad \begin{matrix}(x \in A)\\ d \in C(i(x)/z)\end{matrix} \quad \begin{matrix}(y \in B)\\ e \in C(j(y)/z)\end{matrix}}{(Dx, y)\,(c, d, e) \in C(c/z)}$$

$$\frac{c = f \in A + B \quad \begin{matrix}(x \in A)\\ d = g \in C(i(x)/z)\end{matrix} \quad \begin{matrix}(y \in B)\\ e = h \in C(j(y)/z)\end{matrix}}{(Dx, y)\,(c, d, e) = (Dx, y)\,(f, g, h) \in C(c/z)}$$

+ -*equality*

$$\frac{a \in A \quad \begin{matrix}(x \in A)\\ d \in C(i(x)/z)\end{matrix} \quad \begin{matrix}(y \in B)\\ e \in C(j(y)/z)\end{matrix}}{(Dx, y)\,(i(a), d, e) = d(a/x) \in C(i(a)/z)}$$

$$\frac{b \in B \quad \begin{matrix}(x \in A)\\ d \in C(i(x)/z)\end{matrix} \quad \begin{matrix}(y \in B)\\ e \in C(j(y)/z)\end{matrix}}{(Dx, y)\,(j(b), d, e) = e(b/y) \in C(j(b)/z)}$$

Identity relation

I-*formation*

$$\frac{A \text{ type} \quad a \in A \quad b \in A}{I(A, a, b) \text{ type}} \qquad \frac{A = C \quad a = c \in A \quad b = d \in A}{I(A, a, b) = I(C, c, d)}$$

I-*introduction*

$$\frac{a = b \in A}{r \in I(A, a, b)} \qquad \frac{a = b \in A}{r = r \in I(A, a, b)}$$

I-*elimination*

$$\frac{c \in I(A, a, b)}{a = b \in A}$$

$$\frac{c \in I(A, a, b) \quad d \in C(r/z)}{J(c, d) \in C(c/z)} \qquad \frac{c = e \in I(A, a, b) \quad d = f \in C(r/z)}{J(c, d) = J(e, f) \in C(c/z)}$$

I-*equality*

$$\frac{a = b \in A \quad d \in C(r/z)}{J(r, d) = d \in C(r/z)}$$

Finite types

N_n-*formation*

$$N_n \text{ type} \qquad N_n = N_n$$

N_n-*introduction*

$$m_n \in N_n \ (m = 0, \ldots, n-1) \qquad m_n = m_n \in N_n \ (m = 0, \ldots, n-1)$$

N_n-*elimination*

$$\frac{c \in N_n \quad c_m \in C(m_n/z)\ (m = 0, \ldots, n-1)}{R_n(c, c_0, \ldots, c_{n-1}) \in C(c/z)}$$

$$\frac{c = d \in N_n \quad c_m = d_m \in C(m_n/z)\ (m = 0, \ldots, n-1)}{R_n(c, c_0, \ldots, c_{n-1}) = R_n(d, d_0, \ldots, d_{n-1}) \in C(c/z)}$$

N_n-*equality*

$$\frac{c_m \in C(m_n/z)\ (m = 0, \ldots, n-1)}{R_n(m_n, c_0, \ldots, c_{n-1}) = c_m \in C(m_n/z)}\ (m = 0, \ldots, n-1)$$

Natural numbers

N-*formation*

$$N \text{ is a type} \qquad N = N$$

N-*introduction*

$$0 \in N \qquad 0 = 0 \in N$$

$$\frac{a \in N}{a' \in N} \qquad \frac{a = b \in N}{a' = b' \in N}$$

N-*elimination*

$$\frac{c \in N \quad d \in C(0/z) \qquad \begin{matrix}(x \in N, y \in C(x/z)) \\ e \in C(x'/z)\end{matrix}}{(Rx, y)\,(c, d, e) \in C(c/z)}$$

$$\frac{c = f \in N \quad d = g \in C(0/z) \qquad \begin{matrix}(x \in N, y \in C(x/z)) \\ e = h \in C(x'/z)\end{matrix}}{(Rx, y)\,(c, d, e) = (Rx, y)\,(f, g, h) \in C(c/z)}$$

N-*equality*

$$\frac{d \in C(0/z) \qquad \begin{matrix}(x \in N, y \in C(x/z)) \\ e \in C(x'/z)\end{matrix}}{(Rx, y)\,(0, d, e) = d \in C(0/z)}$$

$$\frac{a \in N \quad d \in C(0/z) \qquad \begin{matrix}(x \in N, y \in C(x/z)) \\ e \in C(x'/z)\end{matrix}}{(Rx, y)\,(a', d, e) = e(a, (Rx, y)\,(a, d, e)/x, y) \in C(a'/z)}$$

Wellorderings

W-*formation*

$$\frac{A \text{ type} \quad \begin{matrix}(x \in A) \\ B \text{ type}\end{matrix}}{(Wx \in A)\,B \text{ type}} \qquad \frac{A = C \quad \begin{matrix}(x \in A) \\ B = D\end{matrix}}{(Wx \in A)\,B = (Wx \in C)\,D}$$

W-introduction

$$\frac{a\in A \quad b\in B(a/x)\to(Wx\in A)\,B}{\sup(a,b)\in(Wx\in A)\,B}$$

$$\frac{a=c\in A \quad b=d\in B(a/x)\to(Wx\in A)\,B}{\sup(a,b)=\sup(c,d)\in(Wx\in A)\,B}$$

W-elimination

$$\frac{\begin{array}{c}c\in(Wx\in A)\,B \qquad (x\in A,\,y\in B\to(Wx\in A)\,B,\,z\in(\Pi v\in B)\,C(y(v)/w))\\ d\in C(\sup(x,y)/w)\end{array}}{(Tx,y,z)\,(c,d)\in C(c/w)}$$

$$\frac{\begin{array}{c}c=e\in(Wx\in A)\,B \qquad (x\in A,\,y\in B\to(Wx\in A)\,B,\,z\in(\Pi v\in B)\,C(y(v)/w))\\ d=f\in C(\sup(x,y)/w)\end{array}}{(Tx,y,z)\,(c,d)=(Tx,y,z)\,(e,f)\in C(c/w)}$$

W-equality

$$\frac{\begin{array}{c}a\in A \qquad b\in B(a/x)\to(Wx\in A)\,B \qquad (x\in A,\,y\in B\to(Wx\in A)\,B,\,z\in(\Pi v\in B)\,C(y(v)/w))\\ d\in C(\sup(x,y)/w)\end{array}}{(Tx,y,z)\,(\sup(a,b),d)=d(a,b,(\lambda v)\,(Tx,y,z)\,(b(v),d)/x,y,z)\in C(\sup(a,b)/w)}$$

Universes

U_n-*formation*

$$U_n \text{ is a type} \qquad U_n = U_n$$

U_n-*introduction*

$$\frac{A\in U_n \quad B\in U_n}{(\Pi x\in A)\,B\in U_n} \qquad \frac{A=C\in U_n \quad B=D\in U_n}{(\Pi x\in A)\,B=(\Pi x\in C)\,D\in U_n}$$

$$\frac{A\in U_n \quad B\in U_n}{(\Sigma x\in A)\,B\in U_n} \qquad \frac{A=C\in U_n \quad B=D\in U_n}{(\Sigma x\in A)\,B=(\Sigma x\in C)\,D\in U_n}$$

$$\frac{A\in U_n \quad B\in U_n}{A+B\in U_n} \qquad \frac{A=C\in U_n \quad B=D\in U_n}{A+B=C+D\in U_n}$$

$$\frac{A\in U_n \quad a\in A \quad b\in A}{I(A,a,b)\in U_n} \qquad \frac{A=C\in U_n \quad a=c\in A \quad b=d\in A}{I(A,a,b)=I(C,c,d)\in U_n}$$

$$N_0\in U_n \qquad N_0=N_0\in U_n$$

$$N_1\in U_n \qquad N_1=N_1\in U_n$$

$$\vdots \qquad\qquad \vdots$$

$$N\in U_n \qquad N=N\in U_n$$

$$\frac{A \in U_n \quad B \in U_n}{(Wx \in A)\, B \in U_n} \qquad \frac{A = C \in U_n \quad B = D \in U_n}{(Wx \in A)\, B = (Wx \in C)\, D \in U_n}$$

$$\begin{array}{ll} U_0 \in U_n & U_0 = U_0 \in U_n \\ \vdots & \vdots \\ U_{n-1} \in U_n & U_{n-1} = U_{n-1} \in U_n \end{array}$$

U_n-*elimination*

$$\frac{A \in U_n}{A \text{ type}} \qquad \frac{A = B \in U_n}{A = B}$$

$$\frac{A \in U_n}{A \in U_{n+1}} \qquad \frac{A = B \in U_n}{A = B \in U_{n+1}}$$

An example will demonstrate how the language-works. Let the premises

$$A \text{ type},$$

$$B \text{ type } (x \in A),$$

$$C \text{ type } (x \in A, y \in B)$$

be given. Make the abbreviation

$$\underbrace{(\Pi x \in A)\, B}_{A \to B}$$

provided the variable x does not occur free in B. Then

$$(\Pi x \in A)\,(\Sigma y \in B)\, C \to (\Sigma f \in (\Pi x \in A)\, B)\,(\Pi x \in A)\, C(f(x)/y)$$

is a type which, when read as a proposition, expresses the axiom of choice. I shall construct an object of this type, an object that may at the same time be interpreted as a proof of the axiom of choice. Assume

$$x \in A,$$

$$z \in (\Pi x \in A).(\Sigma y \in B)\, C.$$

By Π-elimination,

$$z(x) \in (\Sigma y \in B)\, C.$$

Make the abbreviations

$$\underbrace{(Ex, y)\,(c, x)}_{p(c)}, \quad \underbrace{(Ex, y)\,(c, y)}_{q(c)}.$$

By Σ-elimination,

$$p(z(x)) \in B,$$

$$q(z(x)) \in C(p(z(x))/y).$$

By Π-introduction,

$$(\lambda x)\, p(z(x)) \in (\Pi x \in A)\, B,$$

and, by Π-equality,

$$((\lambda x)\, p(z(x)))\,(x) = p(z(x)) \in B.$$

By symmetry,

$$p(z(x)) = ((\lambda x)\, p(z(x)))\,(x) \in B,$$

and, by substitution,

$$C(p(z(x))/y) = C(((\lambda x)\, p(z(x)))\,(x)/y).$$

By equality of types,

$$q(z(x)) \in C(((\lambda x)\, p(z(x)))\,(x)/y),$$

and, by Π-introduction,

$$(\lambda x)\, q(z(x)) \in (\Pi x \in A)\, C(((\lambda x)\, p(z(x)))\,(x)/y).$$

By Σ-introduction,

$$((\lambda x)\, p(z(x)), (\lambda x)\, q(z(x))) \in (\Sigma f \in (\Pi x \in A)\, B)\, (\Pi x \in A)\, C(f(x)/y).$$

Finally, by Π-introduction,

$$(\lambda z)\, ((\lambda x)\, p(z(x)), (\lambda x)\, q(z(x))) \in (\Pi x \in A)\, (\Sigma y \in B)\, C \to (\Sigma f \in (\Pi x \in A)\, B)\, (\Pi x \in A)\, C(f(x)/y).$$

Thus

$$(\lambda z)\, ((\lambda x)\, p(z(x)), (\lambda x)\, q(z(x)))$$

is the sought for proof of the axiom of choice.

To conclude, relating constructive mathematics to computer programming seems to me to have a beneficial influence on both parties. Among the benefits to be derived by constructive mathematics from its association with computer programming, one is that you see immediately why you cannot rely upon the law of excluded middle: its uninhibited use would lead to programs that you did not know how to execute. Another is that you see the point of introducing a formal notation not only for propositions, as in propositional and predicate logic, but also for their proofs: this is necessary to make the methods of computation implicit in intuitionistic (constructive) proofs fit for automatic execution. And a third is that you see the point of formalizing the process of reasoning: this is necessary to have the possibility of automatically verifying the programs' correctness. In fact, if the AUTOMATH proof checker had been written for the theory of types instead of the language AUTOMATH, we would already have a language with the facility of automatic checking of the correctness of the programs formed according to its rules.

In the other direction, by choosing to program in a formal language for constructive mathematics, like the theory of types, one gets access to the whole conceptual apparatus of pure mathematics, neglecting those parts that depend critically on the law of excluded middle,

whereas even the best high level programming languages so far designed are wholly inadequate as mathematical languages (and, of course, nobody has claimed them to be so). In fact, I do not think that the search for high level programming languages that are more and more satisfactory from a logical point of view can stop short of anything but a language in which (constructive) mathematics can be adequately expressed.

References

Bishop, E. 1967 *Foundations of constructive analysis*. New York: McGraw-Hill.
de Bruijn, N. G. 1970 The mathematical language AUTOMATH, its usage, and some of its extensions. In *Symposium on automatic demonstration, lecture notes in mathematics*, vol. 125, pp. 29–61. Berlin: Springer-Verlag.
Curry, H. B. 1958 *Combinatory logic*, vol. 1, pp. 312–315. Amsterdam: North Holland.
Dahl, O.-J., Dijkstra, E. W. & Hoare, C. A. R. 1972 *Structured programming*, pp. 83–174. London: Academic Press.
Dijkstra, E. W. 1976 *A discipline of programming*, pp. 26, 201. Englewood Cliffs. N.J.: Prentice-Hall.
Gentzen, G. 1934 Untersuchungen über das logische Schliessen. *Math. Z.* **39**, 176–210, 405–431.
Howard, W. A. 1969 The formulae-as-types notion of construction. In *To H B. Curry: Essays on combinatory logic, lambda calculus and formalism* (ed. J. P. Seldin & J. R. Hindley), pp. 479–490. London: Academic Press.
Kolmogorov, A. N. 1932 Zur Deutung der intuitionistischen Logik *Math. Z.* **35**, 58–65.
Landin, P. J. 1964 The mechanical evaluation of expressions. *Computer J.* **6**, 308–320.
Martin-Löf, P. 1975 An intuitionistic theory of types: predicative part. In *Logic colloquium 1973* (ed. H. E. Rose & J. C. Shepherdson), pp. 73–118. Amsterdam: North Holland.
Wirth, N. 1971 The programming language Pascal. *Acta informatica* **1**, 35–63.

Discussion

Z. A. Lozinski (*Department of Computing, Imperial College, London, U.K.*). Does Professor Martin-Löf's emphasis on terminating programs come about because of his interest in intuitionistic type theory; or does his use of intuitionistic type theory arise from a belief in termination, i.e. total correctness?

P. Martin-Löf. At the time I designed intuitionistic type theory, I was committed to the interpretation of propositions as types, truth as non-emptiness, absurdity as the empty type, conjunction as the Cartesian product and disjunction as the disjoint union of two types, and so on. Therefore there was no question of having but total elements and functions (that is, hereditarily terminating programs) in the theory. Thus I was in the same predicament as Kreisel in his work on a theory of constructions adequate for the interpretation of intuitionistic logic and Scott in his constructive validity paper. If types are interpreted as domains and elements and functions are taken to be partial, as they are in Scott's mathematical theory of computation, then it is no longer possible to interpret propositions as types, because every type contains an element, namely the bottom element. Hence, if propositions were interpreted as domains and truth as non-emptiness, every proposition would come out true. This is why I could not think of dealing with partial elements and functions, that is, possibly non-terminating programs, before I had freed myself from the interpretation of propositions as types.