Till Peter Dybjer
med hälsningar från
Per Martin-Löf.

CONSTRUCTIVE MATHEMATICS AND COMPUTER PROGRAMMING

Per Martin-Löf

University of Stockholm, Stockholm, Sweden

Paper read at the 6-th International Congress for Logic, Methodology and Philosophy of Science, Hannover, 22 - 29 August 1979.

During the period of a bit more than 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 one is amply compensated 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 next to nothing 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 relative 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 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 nonnumerical) computational tasks, programs that have to be written in a formally precise notation so as to admit of automatic execution. 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 Dijkstra (A Discipline of Programming, Prentice-Hall, Englewood Cliffs, N. J., 1976, p. 201) has suggested to fix terminologically by switching 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 which will become clear below.)

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 <u>Boolean</u>, 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 Hoare's Notes on Data Structuring (O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Structured Programming, Academic Press, London, 1972, pp. 83-174) 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 (The programming language Pascal, Acta Informatica, Vol. 1, 1971, pp. 35-63). The left column of the following table, which shows some of the key notions of programming and their mathematical counterparts, uses notation from ALGOL 60 and PASCAL.

Programming	Mathematics
program, procedure, algorithm	function
input	argument
output, result	value
x := e	x = e
S_1; S_2	composition of functions
<u>if</u> B <u>then</u> S_1 <u>else</u> S_2	definition by cases
<u>while</u> B <u>do</u> 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, $I \rightarrow T$
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	$\{0, 1\}^T$, $T \rightarrow \{0, 1\}$

As can be seen from this table, or from recent programming texts with their little 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. How come? The reason for this curious situation is, I think, that the mathematical notions have gradually received an interpretation, the interpretation which 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 kind of thing as a computer program. Similarly, if a set is understood in Zermelo's way 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. In the case of 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~~s 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 (Foundations of Constructive Analysis, McGraw-Hill, New York, 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 which 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 (An intuitionistic theory of types: predicative part, Logic Colloquium '73, Edited by H. E. Rose and J. C. Shepherdson, North-Holland, Amsterdam, 1975, pp. 73-118), 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 my talk will be devoted to a fairly complete, albeit condensed, description of this language, emphasizing 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 out of variables

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

by means of various forms of expression

$$\underline{(Fx_1,\ldots,x_n)(a_1,\ldots,a_m)}.$$

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 to say that it has itself as value. Thus, in decimal arithmetic,

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

are canonical (normal) expressions, whereas

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

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. Thus 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, noncanonical 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 noncanonical forms of expression correspond to the constructors and selectors, respectively, of Landin (The mechanical evaluation of expressions, Computer Journal, Vol. 6, 1964, pp. 308-320). In the context of programming, they might also aptly be called data and program forms, respectively. The table below displays the primitive forms of expression used in the theory of types, the canonical ones to the left and the noncanonical ones to the right. New primitive forms of expression may of course be added when there is need of them.

~~not~~ quite

Canonical	Noncanonical
$(\Pi x \in A)B$, $(\lambda x)b$	$c(a)$ $(Fx)(c,d)$
$(\Sigma x \in A)B$, (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	$R_2(c,c_0,c_1)$ if then else
$\vdots$	$\vdots$ case ...

N, 0, a'	(Rx,y)(c,d,e) while..
(W x $\in$ A)B, sup(a,b)	(Tx,y,z)(c,d)
U_0, U_1, ...	

The conventions as to what variables become bound in what parts are as follows. Free occurrences of x in B become bound in $(\prod x \in A)B$, $(\sum x \in A)B$ and (W x $\in$ A)B. Free occurrences of x in b become bound in $(\lambda x)b$. 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 the table 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, ..., a_n for the free occurrences variables x_1, ..., 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 $(\lambda x)b$ λ(b) as result, then continue by executing b(a/x) b(a). Thus c(a) has value d if c has value $(\lambda x)b$ 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, ..., 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, ..., n-1. In particular, $R_0(c)$ has no value. It corresponds to the statement

<u>abort</u>

introduced by Dijkstra (A Discipline of Programming, p. 26). 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

<u>if</u> B <u>then</u> S_1 <u>else</u> S_2

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

<u>with</u> e <u>do</u> $\{c_1: S_1, \ldots, c_n: S_n\}$;

introduced by Hoare (Notes on Data Structuring, p. 113) and realized by Wirth in PASCAL as the case statement

$$\underline{\text{case}}\ e\ \underline{\text{of}}\ c_1 : S_1 ;\ \ldots ;\ c_n : S_n\ \underline{\text{end}}.$$

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

$$\underline{\text{while}}\ B\ \underline{\text{do}}\ S.$$

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 following example.

$$\underbrace{\underbrace{\underbrace{(3+2)}_{5}!}_{120}\cdot 4}_{480}$$

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 which 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 $(\lambda x)b$, 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, $(\lambda x)b$ is simply assigned itself as value. The term lazy is appropriate since only as few computation steps are performed 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 above, 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 1) various forms of expression used in the theory of types and explained how expressions composed out of those forms are evaluated. 2) The inferential or, as one says 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 (N. G. de Bruijn, The mathematical language AUTOMATH, its usage, and some of its extensions, Symposium on Automatic Demonstration, Lecture Notes in Mathematics, Vol. 125, Springer-Verlag, Berlin, 1970, pp. 29-61), 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 $x_1 \in A_1$, ..., $x_{n-1} \in A_{n-1}$. When there is 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 which 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 wellknown correspondence between propositions and types discovered by Curry (Combinatory Logic, Vol. I, North-Holland, Amsterdam, 1958, pp. 312-315) and Howard (The formulae-as-types notion of construction, 1969), whereas the transition from the second to the third is Kolmogorov's interpretation (Zur Deutung der intuitionistischen Logik, Mathematische Zeitschrift, Vol. 35, 1932, pp. 58-65) 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 type

A is true, $\exists a \in A$

a is an individual term. $a \in$ INDIVIDUALS
INDIVIDUALS is a TYPE

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 which 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 (Untersuchungen über das logische Schliessen, Mathematische Zeitschrift, Vol. 39, 1934, pp. 176-210, 405-431) that a canonical type (proposition) is defined by its introduction rules. For non-canonical A, a judgment of the form

A 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 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~~ that is the value of 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 $x_1 \in A_1, \ldots, x_n \in A_n$ 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

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

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

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

Thus it is in the nature of a <u>family of types</u> (<u>propositional function</u>) 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

$$a_1 \in A_1,$$

$$\vdots$$

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

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

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

provided

$$a_1 = b_1 \in A_1,$$

$$\vdots$$

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

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

$$a_1 \in A_1,$$

$$\vdots$$

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

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

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

Thus, just as in the case of 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 $x_1 \in A_1$, ..., $x_n \in A_n$. 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

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

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

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

provided

$$a_1 = b_1 \in A_1,$$

$$\vdots$$

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

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 which 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

$$\frac{\overset{\text{hyp}_1}{a \in A} \quad \begin{matrix} \text{hyp}_2 \\ (x \in A) \\ B \text{ type} \end{matrix}}{B(a/x) \text{ type}} \qquad \frac{a = c \in A \quad \begin{matrix} (x \in A) \\ B = D \end{matrix}}{B(a/x) = D(c/x)}$$

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

Assumption

$$x \in A$$

CARTESIAN PRODUCT OF A FAMILY OF TYPES

Π-formation

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

Π-introduction

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

extensionality

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

$app(\lambda(b), a) = b(a)$ $\qquad$ $\lambda((x)\,app(c,x)) = c$

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

β-reduction $\qquad$ η-reduction

DISJOINT UNION OF A FAMILY OF TYPES

Σ-formation

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

Σ-introduction

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

Σ-elimination

Example? Generalization of projection. B independent of x

$\pi_1 = \lambda c.\, split(c, \lambda(x,y).x)$

"split"

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

$$\frac{(a,b) \in (\Sigma x \in A)B \qquad d \in C((x,y)/z)}{(E x,y)((a,b),d) \in C((a,b)/z)}$$

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

Σ-equality

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

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 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

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

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

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 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

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

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

N-equality

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

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

WELLORDERINGS

W -formation

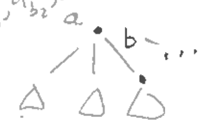
$$\frac{A \text{ type} \qquad \overset{(x \in A)}{B \text{ type}}}{(Wx \in A)B \text{ type}} \qquad\qquad \frac{A = C \qquad \overset{(x \in A)}{B = D}}{(Wx \in A)B = (Wx \in C)D}$$

W -introduction

$B(a/x) = \{b_1, \dots b_n\}$

$B(a/x) \to (Wx \in A)B = \{W_{b_1}, W_{b_2}, \dots W_{b_n}\}$

sup(a,b)



$$\frac{a \in A \qquad b \in B(a/x) \to \overbrace{(Wx \in A)B}^{?}}{\sup(a,b) \in (Wx \in A)B}$$

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

W -elimination

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

$$\frac{c = e \in (\mathrm{W}x \in A)B \qquad d = f \in C(\sup(x,y)/w) \quad \overset{(x \in A,\ y \in B \to (\mathrm{W}x \in A)B,\ z \in (\prod v \in B)C(y(v)/w))}{}}{(Tx,y,z)(c,d) = (Tx,y,z)(e,f) \in C(c/w)}$$

W -equality

$$\frac{a \in A \qquad b \in B(a/x) \to (\mathrm{W}x \in A)B \qquad \overset{(x \in A,\ y \in B \to (\mathrm{W}x \in A)B,\ z \in (\prod v \in B)C(y(v)/w))}{d \in C(\sup(x,y)/w)}}{(Tx,y,z)(\sup(a,b),d) = d(a,b,(\lambda v)(Tx,y,z)(b(v),d)/x,y,z) \in C(\sup(a,b)/w)}$$

UNIVERSES

U_n-formation

U_n is a type $\qquad\qquad U_n = U_n$

U_n-introduction

$$\frac{A \in U_n \qquad \overset{(x \in A)}{B \in U_n}}{(\prod x \in A)B \in U_n} \qquad\qquad \frac{A = C \in U_n \qquad \overset{(x \in A)}{B = D \in U_n}}{(\prod x \in A)B = (\prod x \in C)D \in U_n}$$

$$\frac{A \in U_n \qquad \overset{(x \in A)}{B \in U_n}}{(\sum x \in A)B \in U_n} \qquad\qquad \frac{A = C \in U_n \qquad \overset{(x \in A)}{B = D \in U_n}}{(\sum x \in A)B = (\sum x \in C)D \in U_n}$$

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

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

$\frac{A \text{ type}}{A \in U_n}$ is not ok if $A = U_n$

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

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

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

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

$$\frac{A \in U_n \quad \begin{matrix}(x \in A)\\ B \in U_n\end{matrix}}{(\mathrm{W}x \in A)B \in U_n} \qquad\qquad \frac{A = C \in U_n \quad \begin{matrix}(x \in A)\\ B = D \in U_n\end{matrix}}{(\mathrm{W}x \in A)B = (\mathrm{W}x \in C)D \in U_n}$$

$$U_0 \in U_n \qquad\qquad U_0 = U_0 \in U_n$$

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

$$U_{n-1} \in U_n \qquad\qquad U_{n-1} = U_{n-1} \in U_n$$

U_n-elimination

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

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

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

A type,

B type $(x \in A)$,

C 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 which 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

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

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,

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

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