LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

Computer Assisted Proof for Mathematics: an Introduction Using the LEGO Proof System

by

Rod Burstall

Computer Assisted Proof for Mathematics:

LFCS Report Series ECS-LFCS-91-132

LFCS January 1991

Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

The Mathematical Revolution Inspired by Computing. J H Johnson & M J Loomes (eds)

To appear.

Computer Assisted Proof for Mathematics: an Introduction Using the LEGO Proof System

Rod Burstall,
Laboratory for Foundations of Computer Science,
King's Building, Edinburgh University, Edinburgh, EH9 3JZ

Abstract

We give brief account of the use of computers to help us develop mathematical proofs, acting as a clerical assistant with knowledge of logical rules. The paper then focusses on one such system, Pollack's LEGO, based on the Calculus of Constructions, and it shows how this may be used to define mathematical concepts and express proofs. We aim at a gentle introduction, rather than a technical exposition.

1 Automatic Proof and Proof Checking

Attempts to use computers to produce mathematical proofs have followed two main paradigms. In the first, "automatic theorem proving", the machine searches for a proof of a theorem given some premises [4]. Although a considerable amount of work has been done on formulating proof systems which reduce the search space as far as possible, mainly using Robinson's resolution method [13], it is still hard to do interesting proofs. Steady progress has been made but the limitations of machine search are quite severe, and the technology has not been adopted by mathematicians. In the second approach the proof is invented by the human user and the machine is used to check that it has no incorrect or missing steps. This greatly reduces the demand on the machine, but transfers it to the user who has to provide a proof in a notation sufficiently formal to be understood by the machine and sufficiently detailed for each step to be recognised as correct. This is very much more burdensome than the level of proof required for normal communication between mathematicians.

Consider the analogy between formal proofs and programs. A major difference is that we can mechanically recognise the correctness of a proof, but for a program we can only recognise syntactic and type correctness and there is no algorithm for checking that it fulfils its intended purpose. There was a time in the sixties when the term "Automatic Programming" was in vogue, but it proved chimerical and the term has more or less dropped out of the technical vocabulary. However, non-automatic programming is very much with us, a major industry, even though programs are formal objects which people find difficult to produce. People do not write correct programs, but they get rapid feedback from the machine when they

make mistakes. In a slow process over the last thirty years the level of languages used in programming has been raised, and the speed of machine response has been reduced from hours to seconds. It seems that there is a similar development for proofs; the logical languages in which we can express proofs to the machine have become more pithy and expressive, and the machine can notify us of mistakes very fast, so that without any change in principle the process of developing a machine checked proof has speeded up by a couple of orders of magnitude. A traditional proof is a series of statements starting with the premises and finishing with the theorem, each statement justified as obtained by the application of some logical rule to some previous ones. For example, if we use "....." to represent various mathematical formulas, then

Premise 1	:	
Premise 2	:	
Step 1	:	by ruleA from premise 1
Step 2	:	by ruleB from premise 2
Step 3	:	by ruleC from step 1 and step 2
Theorem	:	by rule C from premise 1 and step 3

When typing a proof into the machine the most tedious part is typing in the formulas represented above by ".....". Of course we must put in the premises, but the step formulas and the theorem can then be computed from these if we say which rules are to be use on which arguments. Given the two premises we can write the remainder of the proof as follows

step 1 = A(premise 1)
step 2 = B(premise 2)
step 3 = C(step 1, step 2)
theorem = C(premise 1, step 3)

In fact all we really need to write down is the following expression which gives the proof of the theorem from the premises

C(premise 1, C(A(premise 1), B(premise 2)))

We can think of such expressions as being proofs; a proof expression is either the name of a premise or the name of a proof rule applied to some proof expression. From the proof expression and the premise formulas we can calculate the formula which we have proved (the theorem) by applying the inference rules in the manner indicated by the proof expression. Of course proof expressions are rather hard to write down if one does not write out the intermediate formulas, but this is a task which is mechanical and easily accomplished by the computer. So the technique is to interact with the machine as follows, where everything is typed in by the user except the underlined formulas which are typed out by the machine:

premise 1 = ...
premise 2 = ...
step 1 = A(premise 1)
.....
step 2 = B(premise 2)
.....
step 3 = C(step 1,step 2)
.....
theorem = C(premise 1, step 3)
.....

With a graphical interface one can do a little better: the machine displays a menu of proof rules and the user simply points to the rule she wants to use and the formulas it is to be applied to; the names "step 1", "step 2" etc. can be dispensed with. In fact, once the premises have been typed in the user does not have to type anything else, just point. This is altogether a more relaxed way of doing business with the computer.

In practice it is helpful to work back from the theorem as well as forward from the premises, a method often called *Proof by Refinement.* It is also helpful to use the *Natural Deduction* style of proof system in which trying to prove a fact "if P then Q" we can add P temporarily to our list of premises and just try to prove Q. The proof expressions in a natural deduction system have a somewhat more elaborate structure than those indicated in the example above. They involve expressions for functions, and they use local variables standing for the (as yet unknown) proofs of the assumptions; in fact these expressions are what logicians call "lambda expressions".

It turns out that the problem solved by the machine above is "What formula does the following proof expression prove?", that is filling in the underlined dots above, is a well known one. In fact it is just the same as finding the *type* of an expression in programming languages, provided that we adopt a sufficiently rich notion of type. This idea goes back to Curry and Howard, and it is often referred to as *Propositions as Types.* It forms the basis for the proof systems currently popular in Computer Science, known as *Type Theory* [10] or *Calculus of Constructions* [5][12]. In this paper we will give an overview of one implementation of the Calculus of Constructions, namely the "LEGO" system developed at Edinburgh by Pollack [7,9]; for the pioneering implementation of Type Theory see [6]. A recent tutorial on Type Theory is given in [1] , and a more extensive treatment is given in [11].

Having distinguished the automatic proof approach from the proof checking approach, my own preference is for the latter, rather following the story of Achilles and the tortoise. Automatic proof can accomplish some proofs very quickly, but when the search fails to find a proof, the user is left at the point of failure in a machine oriented world and may find it difficult to know what the difficulty was and how to proceed further. In the proof checking paradigm we can work with a more human-understandable representation, and although progress may be slow one is less likely to come up against major barriers. In practice a combination of

these two approaches is used. The automatic proof systems are steered through a sequence of theorems chosen by the user, using earlier theorems as stepping stones to later ones. The proof checkers perform some symbolic computation and often use search strategies programmed by the user to fill in automatically some of the more routine steps.

2 Why Work on Computer Proof?

The main motivation for work on computer assisted proof has been the desire to prove that programs do what is intended. This means giving some formal specification of the task to be accomplished, using some logical language adequate to express the mathematical concepts involved, then proving that the result computed by the program is in accord with this specification, *for any input.* The usual method of convincing ourselves that a program is correct, debugging it by trying it on a selection of inputs, can only show this for the finite collection of inputs . Extrapolation to other inputs is the act of faith on which the software industry is built. The mediaeval scholastics might have been pleased to see this demonstration of the inadequacy of reason without faith.

In practice debugging works surprisingly well, but for programs where the cost of error in money or human lives is large one would like to do better. Also in concurrent programs where parts of the program run on different processors proceeding at somewhat unpredictable speed it may not be possible to re-run inputs which gave rise to an error and check that the bug has been corrected, timing differences may give different results even for the same inputs; experiments are not repeatable so empirical testing fails.

However I would like to put forward somewhat speculatively another motivation for working on computer proof systems. The number of mathematicians has not greatly increased since 1950, whilst the number of programmers has increased explosively, many of them largely self taught. To learn to develop mathematical proofs one has to acquire some feeling for what a proof is and when the level of rigour is adequate, otherwise one may be just handwaving. Basically you have to show it to someone who is mathematically skilled and they have to spend an appreciable amount of time looking at it critically. In programming on the other hand the machine, through syntax analysis and debugging, is able to give us a lot of feedback. We still need models to imitate and some advice from our betters, but the computer can point out when we have made a mistake, and provide us with some assurance that we have got something right. This feedback loop is crucial in learning to program. Suppose we had to teach people to program without access to computers, How many programmers would there be in the world, even if we had a real need for programs? Programming would be an arcane speciality.

So it seems to me that it is worth trying to get the same kind of feedback for the activity of producing mathematical proofs. You may say that mathematics is a highly intellectual activity, requiring sophisticated intuitions. But it would be a mistake to dismiss programming as a low level symbol manipulation; it too requires

a high level of skill and intuition. Because a program can be executed mechanically we should not be misled into thinking that there is anything mechanical about the process of *inventing* programs. My contention is that many people, not at the genius level, can exhibit a high level of intuitive skill because the concrete feedback from the computer gives them criticism when they are wrong and confidence when they are right. Our intuition always rests on experience. Maybe if the same level of feedback could be given for mathematical proofs the number of people capable of producing proofs would increase markedly. This might not contribute directly to the discovery of more advanced pieces of mathematics, but if many more people could climb the foothills of mathematics then surely some of them might scale the peaks. If I am right about the possibility of a major scaling up in the number of people able to construct mathematical proofs, then this would surely have an impact on our ability to tackle the program correctness problem in practice.

3 A Formal Language for Constructive Mathematics

We now describe the Calculus of Constructions, a logical language with proof rules suitable for formulating constructive mathematics. Intuitionist (constructive) logic, disallowing the law of excluded middle P **or** *not-*P, seems well adapted to formal proof development on computers, and it has been shown that a considerable portion of mathematics can be expressed constructively [2][3][14]. In particular the parts of mathematics used in computing may be expected to be largely constructive. We will use for our notation that of the **LEGO** implementation of the calculus.

3.1 Type construction operations

The Calculus of Constructions uses the *type construction operators* $\{\_\}\_$ and $<\_>\_$ which are generalisations of the familiar $\rightarrow$ and $\times$. Let us explain them as generalisations of corresponding operations on sets. Suppose S and T are sets.

$f \in S \rightarrow T$	means	$fx \in T$ for all $x \in S$
$(x, y) \in S \times T$	means	$x \in S$ and $y \in T$.

We write fx for the more usual $f(x)$, meaning the value of f for argument x. We write (x, y) for an ordered pair. Now instead of the set T consider a family of sets over S, say F, that is for each $x \in S$, Fx is a set (note that we write Fx rather than F_x).

$f \in \{x : S\}(Fx)$	means	$fx \in Fx$ for all $x \in S$
$(x, y) \in < x : S > (Fx)$	means	$x \in S$ and $y \in Fx$.

In other words, $\{x : S\}(Fx) = \{f \mid fx \in Fx \text{ for all } x \in S\}$, and $< x : S > (Fx) = \{(x, y) \mid \forall x \in S,\ \forall y \in Fx\}$. Notice that if Fx is a constant set T then these reduce to $S \to T$ and $S \times T$ respectively. $\{x : S\}(Fx)$ is often written $\prod_{x:S} Fx$, and $< x : S > (Fx)$ is often written $\sum_{x:S} Fx$, and they are called the *product* and *sum* respectively.

Example for $\{\_\}\_$

Let $\mathbb{N}$ be the natural numbers with zero, $\mathbb{N} = \{0, 1, \ldots\}$, and $\mathbb{R}^n$ be the n-dimensional space of reals. Then a member of $\{n : \mathbb{N}\}(\mathbb{R}^n \to \mathbb{R})$ gives a function from $\mathbb{R}^n$ to $\mathbb{R}$ for each n, that is it is a function which given a number n produces a particular function from $\mathbb{R}^n$ to $\mathbb{R}$. For example *modulus* m is a member of this set, where $mi(x_1, \ldots, x_i) = (\sqrt{x_i^2 + \ldots + x_i^2})$ (such functions are often called *polymorphic* in computer language theory because they work for a whole family of types, here $\mathbb{R}^n$ for any n).

Example for $< \_ > \_$

Let *String* n be the set of strings of n characters. Then $< n : \mathbb{N} >$ (*String* n) is the set of pairs (n, s) such that n is a number and s is a string of length n.

Other examples:

A matrix transpose function for m by n matrices, any m, n:

$Transpose \in \{m : \mathbb{N}\}\{n : \mathbb{N}\}(Matrix(m, n) \to Matrix(n, m))$.

A finite state automaton, a tuple of three sets and two functions over them:

$$Automaton \in < In : Set >< Out : Set >< State : Set >$$
$$((In \times State \to State) \times (State \to Out) \times State)$$
(a transition function, an output function and a start state)

Now the Calculus of Constructions is a formal calculus in which these two operators are used to build types, reading S and T as types and F as a type valued function. In examples like Automaton we can replace *Set* by *Type*, the type of all types. This of course produces difficulties over paradoxes, so we introduce a hierarchy as follows, $Type_0$, $Type_1, \ldots$. For example, $\mathbb{N}$, $\mathbb{N} \to$ *String* and $\mathbb{N} \times \mathbb{N}$ are in $Type_0$. (In fact we can *declare* basic types like $\mathbb{N}$ and *String* in $Type_0$ with some axioms (see below); then combining them with $\to$ or $\times$ keeps us in $Type_0$.) $Type_0$ and $Type_0 \times Type_0 \to Type_0$, for example, are in $Type_1$ and so on. This enables the calculus to be given a model theoretic semantics. In practice it is rather irritating to have to write the subscripts on the types; **LEGO** is able to deduce the subscripts, thus relieving the user of the obligation to write them.

In the **LEGO** notation we use $->$ for $\to$ and $\#$ for $\times$, since these characters are not available in the standard computer character set. We will also use *nat* for $\mathbb{N}$, for the same reason.

3.2 Expressions and Functions

Expressions are formed from constants, variables and the application of functions to arguments, as we have seen in the examples above. We need some notation for functions and adopt the λ notation of Church (see Hyland's paper in this volume [8]) with a different syntax. If E is an expression involving x we write

$$[x : S]E$$

for "E as a function of x, where x is a variable of type S" (Church wrote $\lambda x : S.E$).

Example

$[n : nat](plus(times\ n\ n)\ one)$ - the function f defined by $f(n) = n^2 + 1$ for all n in nat

(The **LEGO** computer input notation currently does not allow infixes like $+$, an infelicity which will be rectified sometime - so I am being honest in the examples in case you want to use the **LEGO** program). We can then regard a definition like

$$f[n : nat] = plus(times\ n\ n)\ one$$

as a syntactic alternative to the basic form

$$f = [n : nat](plus(times\ n\ n)\ one)$$

There are two ways of defining two argument functions. We can define the function to take a pair as argument (pairs are written using a comma and parentheses), or we can define it to take an argument and produce as a result a function which can be applied to the second argument. We use these as follows, assuming $x : s$ and $y : t$,

$f(x, y)$	*a function applied to a pair*, $f : s\#t \text{ -> } u$
$f\,x\,y$	*f applied successively to* x and y, $f : s \text{ -> } t \text{ -> } u$

The second is syntactically simpler and we tend to prefer it. (We used it implicitly for plus and times above). To define such a function we write

$$f[m : nat][n : nat] = \ldots$$

or more briefly

$$f[m, n : nat] = \ldots$$

We can define functions to take a type argument (polymorphic functions)

$id[t : Type][x : t] = x$ *the polymorphic identity function* $\lambda x.x$

It is often convenient to omit this type argument when using the function, and there is a special notation for this:

$id[t|Type][x : t] = x$ the same function but you do not have to give the type argument explicitly (| replaces :)

With the former definition we would have to write *id nat* 3, with the latter we may write simply *id* 3

3.3 Definitions and Declarations

We have seen how to define functions using the equality sign, = . We can define constants similarly. But we can also introduce names of any type by declaration without giving a definition; for this we use [. . . : . . .]. For example to declare n as a natural number and f as a function from natural numbers to natural numbers

$[n : nat]$;

$[f : nat \rightarrow nat]$;

Notice that we use a semicolon after a declaration or definition.

3.4 Propositions and proofs

We wish to develop some logic in our language, in fact a higher order intuitionist logic; for this we will need propositions. We have a type *Prop* of propositions. In fact just as

$$Type_0 : Type_1$$

we have

$$Prop : Type_0$$

Suppose we declare A and B to be propositions, that is values of type *Prop*

$$[A, B : Prop];$$

then we can define further propositions using **and** and **or**, written $\wedge$ and $\vee$ respectively, thus

$$A \vee (B \wedge A)$$

We use $->$ for implication

$$A -> (B \wedge A)$$

and **absurd** for the proposition which has no proof, that is false. Negation is defined by

$$not[A : Prop] = A -> absurd$$

As a matter of fact **and**, **or** and **absurd** are not primitive notions in our language but can themselves be defined, similarly for an equality Q [9]. This shows the power of the basic formalism, but is of less importance to the user who wishes to deal with portions of mathematics, so we omit the definitions here. The use of $->$ for implication is not a notational trick; it really is the same operation as the one used for forming function types. It turns out that the universal quantifier, "for all", is expressed by the dependent function type operation. Thus if we declare predicates P and R over natural numbers and pairs of natural numbers respectively

$$[P : nat -> Prop]; [R : nat \# nat -> Prop];$$

we can form quantified formulas

$\{x : nat\}(Px)$ that is $\forall x : nat.P(x)$

$\{x, y : nat\}(Px -> R(fx, y))$ that is $\forall x, y : nat.Px \supset R(fx, y)$

The existential quantifier, *exists*, can be defined. It takes a function as argument, e.g.

$exists\ ([x : nat](R(x, x))$ that is $\exists x : nat.R(x, x)$

Here the argument of *exists* is $R(x, x)$ as a function of x. Propositions are themselves types and have proofs as their elements.

We can use declarations to introduce assumptions, just as we used them to introduce constants. In fact an assumption is just a declaration of a constant which denotes a proof of the fact being assumed

$$[axiom\ 1 : \{x, y : nat\}(Px -> R(fx, y))]$$

It is possible in **LEGO** to write expressions for proofs, indeed they are formed in just the same way as expressions for other values, but the normal way to create proofs in **LEGO** is by "refinement" (described below), so we will not discuss proof expressions here. It is an important property of the Calculus of Constructions that proofs are values and can be manipulated just like other values, for example they can be arguments and results of functions and elements of pairs.

3.5 Discharging assumptions and declarations

We can discharge an assumption A. In this case any theorem B that we had proved in the context containing the assumption becomes $A \text{ -> } B$ in a new context without the assumption A. In discharging a declaration of a variable x any previous function definitions take x as an extra parameter. (These are really particular cases of the same operation.)

4 An Example: Complete Partial Orders

To see how this logical language can be used to describe mathematical systems we consider the definition of a complete partial order. This is an ordering for which every ascending sequence has a least upper bound. For ease of reading I have omitted some brackets and parentheses which LEGO requires.

$t \mid Type$;	assume t is a type (an inferred type since we use vertical bar)
$le : t \text{ -> } t \text{ -> } Prop$;	assume le is a function from t and t to $Prop$, a binary relation - less than or equal
$Refl = \{x : t\}(le\ x\ x)$;	naming a fact about le - reflexivity
$Trans = \{x, y, z : t\}(le\ x\ y \text{ -> } le\ y\ z \text{ -> } le\ x\ z)$;	transitivity
$Antisym = \{x, y : t\}(le\ x\ y \text{ -> } le\ y\ x \text{ -> } Q\ x\ y)$;	antisymmetry
$POrd = Refl \wedge Trans \wedge Antisym$;	Define Partial Order
$seq = nat \text{ -> } t$	new type sequence - function from nat to t
$Chain[s : seq] = \{n : nat\}(le(s\ n)(s\ (succ\ n)))$;	s is a chain iff $s_n \leq s_{n+1}$ for all n
$Ub[s : seq][x : t] = \{n : nat\}(le(s\ n)\ x)$;	x is upper bound of sequence s iff $s_n \leq x$ for all n
$Lub[s : seq][x : t] = Ub\ s\ x \wedge \{x' : t\}(Ub\ s\ x' \text{ -> } le\ x\ x')$;	x is least upper bound of sequence s iff it is an upper bound and less than or equal to all other upper bounds

We have a predicate *Chain*, but we would like to have a type *chain*. This is the type of all sequences satisfying the predicate *Chain*. We define the type *chain* to be the type of ordered pairs consisting of a sequence and a proof that the sequence satisfies the predicate *Chain*. We are making use of the fact that proofs are values and can be components of pairs.

$chain =< s : seq > Chain\ s;$	Type chain is pairs consisting of (i) a sequence s (ii) a proof that the sequence s is a Chain.
$lub :\ chain \rightarrow\ t;$	Least upper bound function.
$Complete = \{c : chain\}(Lub\ c.1(lub\ c));$	Complete means the function *lub* gives the least upper bound (*Lub*) for each chain. $c.1$ is c considered as a sequence (first component of c).
$CPOrd = POrd\ \wedge\ Complete;$	Complete Partial Order - a property of relation *le*.
$Discharge\ t;$	Discharge all assumptions back as far as t. If a definition depends on a discharged variable make it a parameter e.g. t and *le* are parameters of Lub (t is inferred).

$cpord[t : Type] =< le : t \rightarrow\ t \rightarrow\ Prop >< lub : chain\ t \rightarrow\ t > (CPOrd\ lt\ lub);$

Complete Partial Order s over t - a new type, whose elements are triples
(i) a less than or equal function
(ii) a least upper bound function
(iii) a proof that these form a partial order.

$s, t|Type;$

$continuous[C : cpord\ s][D : cpord\ t] = \ldots;$	The type of continuous functions from C to D. These are defined as pairs (i) function from s to t, and (ii) a proof that it is continuous We omit the details.

In this example we have made some simplifying assumptions, for example in using a standard equality, Q, instead of an arbitrary one, but it illustrates the general approach. A convenient representation of set theory in the Calculus of Constructions is still a topic of current research.

5 A Sample Proof

We will illustrate a simple refinement proof in the **LEGO** system (again omitting some brackets and parentheses). We make two assumptions, that the relation R is symmetric and transitive, then show that Rxy implies Rxx. Proof commands typed by the user are in **bold**. The text on the right is commentary. $?n$ is a subgoal derived by machine in response to a proof command. We will explain briefly the two proof commands needed for our example. There are several others not described here.

The **Intros** command can be used in two ways. Applied to a goal with a universal quantifier "for all $x : t$", written $\{x : t\}$ it strips off the quantifier to produce a simpler subgoal, and it adds $x : t$ to the context in which later steps are carried out, as a temporary premise. Applied to an implication, $P \rightarrow Q$, it produces a simpler subgoal Q and adds P as a temporary premise.

The **Refine** command uses an assumption to simplify a goal. If the goal is Q and the assumption is $P \rightarrow Q$ then it produces the subgoal P. If the goal is $Q\ a$ and the assumption is $\{x : t\}(Px \rightarrow Qx)$ then it produces the subgoal $P\ a$. If the goal is $?n$ then **Refine** a replaces $?n$ by the term a. **Refine** also deals with more general cases involving several premises and several variables, but we need not discuss the general case here. (Technically speaking the **Refine** command performs first order unification.)

These commands work on the current top subgoal.

$t : Type$; - assumption
$Sym : \{x, y : t\}(R\ x\ y \rightarrow R\ y\ x)$; - assumption
$Trans : \{x, y, z : t\}(R\ x\ y \rightarrow R\ y\ z \rightarrow R\ x\ z)$; - assumption

Goal $\{x, y : t\}(R\ x\ y \rightarrow R\ x\ x)$; - to be proved
Intros $x\ y$; - strip quantifier off goal

$x : t$
$y : t$
$?1 : R\ x\ y \rightarrow R\ x\ x$ - new (sub)goal
Intros $h1$; - strip premise off goal?1

$h1 : R\ x\ y$
$?2 : R\ x\ x$
Refine Trans; - use fact Trans for goal?2

?3 : t
?4 : $R\ x$?3
?5 : R ?3 x
Refine y; -instantiate ?3 to y

?6 : $R\ x\ y$
?7 : $R\ y\ x$
Refine $h1$; - use assumption h1 for goal ?6
Refine *Sym*; - use fact Sym for goal ?7

?8 : $R\ x\ y$
Refine $h1$; - use assumption h1 for goal ?8

QED - LEGO says that the proof is complete.

It only remains to save the resulting proof under an appropriate name for use in proving later theorems.

Acknowledgements
I am very grateful to Randy Pollack for the opportunity of using his **LEGO** proof system and to various other users of the system who have helped me to understand it, notably to Paul Taylor and to Claire Jones. The **LEGO** users manual, largely written by Zhaohui Luo, has been very helpful. I am grateful to the Science and Engineering Research Council and the European Community BRA for supporting our research in this area. I would like to thank the referees and editors for their helpful comments.

References

[1] Backhouse R., Chisholm P., Grant M. and Saaman E., 'Do-it-Yourself Type Theory', *Formal Aspects of Computing*, 1,1, Springer International ,1989.

[2] Beeson, M. J., *Foundations of Constructive Mathematics*, Springer, 1989.

[3] Bishop, E., *Foundations of Constructive Analysis*, McGraw Hill, 1967.

[4] Bundy, A., *The Computer Modelling of Mathematical Reasoning*, Academic Press, 1983.

[5] Coquand Th. and Huet G., 'The Calculus of Constructions.' *Information and Control*, 76, 1988.

[6] Constable R.L. *et al. Implementing Mathematics in the NuPrl Proof System*, Prentice Hall, 1986.

[7] Harper R. and Pollack R., 'Typechecking, Universe Polymorphism and Typical Ambiguity in the Calculus of Constructions', *Proc TAPSOFT Conference*, Barcelona, Springer LNCS 352, 1989.

[8] Hyland, J. M. E., 'Computing and Foundations', in *The Mathematical Revolution Inspired by Computing*, J H Johnson & M J Loomes (eds), Oxford University Press, 1991.

[9] Luo Z., Pollack R. and Taylor P., 'How to use **LEGO**' (A preliminary Users' Manual), unpublished draft, LFCS, Dept of Computer Science, Edinburgh University, 1989.

[10] Martin-Lof P., *Intuitionistic Type Theory*, Bibliopolis, Naples, 1984.

[11] Nordstrom B., Peterson K., and Smith J., *Programming in Martin-Lof's Type Theory*, Clarendon Press (Oxford), 1990.

[12] Projet Formel, 'The Calculus of Constructions: Documentation and User's Guide', INRIA, Rocquencourt BP105, France, Technical Report 110, 1989.

[13] Robinson, J. A., 'A machine oriented language based on the Resolution Principle', *Journal of the ACM*, 23-41, 1965.

[14] Troelstra, A. S., van Dalen, D., *Constructivism in Mathematics: An Introduction (Vols 1 & 2)*, North Holland, 1988.