LUIGIA AIELLO
MARIO AIELLO
GIUSEPPE ATTARDI
GIANFRANCO PRINI

Informal proofs formally checked by machine

Annales scientifiques de l'Université de Clermont-Ferrand 2, tome 60, série *Mathématiques*, nº 13 (1976), p. 31-53
<http://www.numdam.org/item?id=ASCFM_1976__60_13_31_0>

Ann. Sci. Univ. Clermont, Sér. Math., fasc. 13, 1976, pp. 31-54

INFORMAL PROOFS FORMALLY CHECKED BY MACHINE

Luigia AIELLO, Mario AIELLO, Giuseppe ATTARDI, Gianfranco PRINI

Consiglio Nazionale delle Ricerche, PISA, Italy

Universitá di Pisa, PISA, Italy

ABSTRACT. - The paper describes a research in progress. The research consists in the construction of a system for generating proofs of theorems in an interactive way with the help of a computer.

The system is described in some detail. The ideas inspiring it are presented, as well as the underlying logic and a bunch of examples taken from the Mathematical Theory of Computation (i.e: theory of programs, program schemata, programming languages).

Considerations are made on further developments of the system in two main directions :

i) extension of the logic to increase its expressive power and implementation of other logics ;

ii) extension of the proving capability of the system.

The first extension requires an analysis of the fields of application foreseen for the system, while the second one requires an accurate analysis of the most commonly used strategies for building proofs. This is possible only after a long experimentation on the use of the system.

INTRODUCTION

Since a long time, mathematicians formulated the desire of mechanical theorem proving. The advent of electronic computers has renewed such desire and has proposed new fields of application. Besides the mechanical construction of interesting proofs in ordinary mathematics, possible applications are found in the so called «Mathematical Theory of Computation» and «Artificial Intelligence».

In the Sixties, computer scientists devoted a great effort to the implementation on electronic computers of automatic theorem provers. Sets of complete and uniform inference rules have been proposed for such systems - for instance see {1}.

Soon the two following drawbacks of this approach appeared.

i) The combinatorial explosion, which is inherent to uniform inference rules, leads to the practical impossibility of proving even simple interesting theorems. Investigations have been done for finding more efficient strategies. They essentially failed. This failure is confirmed by a recent result {2} stating that every complete proving algorithm for sufficiently powerful theories is inherently inefficient.

ii) Since the steps of the proof are too elementary and the inference rules are far from usual mathematical reasoning, it is extremely difficult to understand the «outline» of the proof by analyzing the behaviour of the automatic theorem prover. This is undesirable in case of both success and failure. In fact, if the analysis of the proving activity is hard to be performed, no experience can be gained by the user to improve the strategies of the system in order to obtain better proofs of theorems successfully proved or find a remedy to previous failures.

To overcome these difficulties, a different approach has been undertaken : the computer is not asked to prove theorems automatically, but to help the user to build proofs.

Earlier works on this line failed because of the constraints imposed by the computers and the programming languages existing at that time. A big improvement in man-machine communication has been introduced by the availability of on-line programming and by the new very high level programming languages.

Recently this approach has been followed at the Artificial Intelligence Laboratory of the Stanford University, where proof checkers are being developed for both typed (LCF - see {3}) and type free (TFL - no description is available at present) λ-calculi and for first order logic (FOL - see {4}).

Presently a proof checker called PPC (Pisa Proof Checker) is being developed at Pisa. The aim of this paper is to present PPC. We intend to explain its underlying ideas, to give some insight into the program which realizes it and to explain to what extent it is an improvement with respect to the already existing proof checkers. After a presentation of the logic presently implemented in PPC, we give some sample proofs. They have been chosen for providing a description of the proving strategies included in PPC and of their advantages. Finally, the applications foreseen for PPC are presented, as well as the kind of experience and insight that is supposed to be gained by using the proof checker. The main applications expected at the moment are in the field of Mathematical Theory of Computation. This because one of our main concern is mechanical program verification. We intend to proceed in the kind of experiments described in {5} : a programming language has been axiomatized in LCF and properties of some programs have been machine checked with very compact and elegant proofs. Besides this application, we think that many others are possible, and not only in the computer science field.

The greatest advantage we hope to have by using a proof checker instead of an automatic theorem prover, besides the feasibility of big meaningful proofs impossible to be done by nowaday automatic theorem provers, is an analysis of the strategies more frequently used for building proofs. This makes it possible to mechanize them and to obtain a greater help by the machine. The help supplied by a proof checker to the user may be increased by adding theorems already proved as auxiliary deduction rules. Even metatheorems of the logic may be proved, as shown in { 6} , and reflected into new deduction rules of the logic itself. How this enriches the proving power of the checker has still to be investigated. What we expect from this experience is to find the boundary between routine and creative proving activity. Only the second one has to be left to the mathematician, leaving to the computer all the routine work. To tell it in other words, the kind of expectation we have is to build a system which formally checks informal proofs.

DESCRIPTION OF THE SYSTEM

In this section we describe, by means of an example, the behaviour of PPC. We take a simple theorem and show various proofs interactively built for it, using PPC. A sample dialogue between the user and the system is, in our opinion, the best explanation of its performance. The dialogue is fictitious because we take a theorem from ordinary mathematics, while the only logic implemented so far is LCF (it will be described in the next section). We have chosen such an example since we think that it is closer to common mathematical reasoning. However the kind of proving strategies is almost the same as in LCF. Note that all other examples shown in the paper have actually been performed on the machine. The sample dialogue is also fictitious because we adopt usual mathematical symbols (for more readibility) which are not available on teletypes - for example the formula $\sum_{i=0}^{n} f(i) = \frac{n(n+1)}{2}$ is written on the teletype as SUMMATION (I,0,N,F(I)) = N * (N + 1)/2. This difficulty may be overcome if more sophisticated communication media between man and machine are available, for example a CRT terminal, where the user can define the symbols he wants, even with subscripts and superscripts.

PPC is presently implemented in MAGMA-LISP { 7 } and runs on the IBM 370/168 computer available at the CNUCE of CNR at Pisa. The conversation takes place through a teletype.

Suppose that a mathematician wants to use PPC for generating a proof of the proposition

$$\forall n. \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$$

After a few routine commands for asking the operating system to load PPC he types

```
ENV PEANO ;
```

which tells PPC he wants to build a proof in the environment of Peano arithmetic. PPC replies with the message

```
***** PEANO ARITHMETIC LOADED
```

The mathematician is thus informed that the system is now ready to operate. It knows first order predicate calculus with identity (i.e. its syntax, its axioms and its inference rules), Peano axioms and the définition of arithmetical operations, ordering and many other predicates and functions on integers. In addition it may have some procedural knowledge about Peano arithmetic. For instance it knows how to simplify arithmetic expressions into some standard form. Note that messages written by PPC always begin with «*****».

The user states the theorem he wants to prove by typing

GOAL $\forall n . \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$;

The system gets it and gives it a number.

***** GOAL # 1 $\forall n . \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Many goals can be put into the system at the same time. They will be numbered automatically. Whenever one of them is proved the system gives an appropriate answer.

The mathematician decides that the proof is to be done by induction on n. First he tries to prove the basis of the induction and states it as a lemma, i.e. a new goal.

GOAL $\sum_{i=0}^{0} i \quad \frac{0(0+1)}{2}$;

***** GOAL # 2 $\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$

This goal can be established by simply evaluating the left part and the right part of the equality and checking for their identity. This is done by a simplification algorithm which is available in the system. It can be either a general procedure applicable to any first order theory (for instance using unification, modus ponens and equality), or a specialized one, which knows the rules for the symbolic evaluation of arithmetic expressions. Note that in the first case the simplification algorithm could be also an implementation of some resolution strategy - a technique widely used in automatic theorem provers. In PPC it is the user who decides, for each step of the proof, which axioms and definitions are relevant and must be used by the simplification algorithm.

SIMPL GOAL # 2 ;

***** GOAL # 2 PROVED $\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$

LABEL BASIS ;

The user wants to keep record of the lemma he has just proved. For this reason he gives it a name by the LABEL command. From now on this lemma will be retrieved by its name (i.e. BASIS).

Now the mathematician has to prove the induction step. The induction hypothesis is assumed.

ASSUME $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$;

***** 1 $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ (1)

The system automatically associates a progressive number to each proof step. The first «1» appearing in the message says that the above assumption is the first step of the proof, while the «1» in parentheses records the fact that step 1 is an assumption, so it depends only on itself. If we prove that, on that assumption, the same equality holds for (n+ 1), then the complete proof is got by the induction principle.

GOAL $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$;

***** GOAL # 2 $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$

This goal is again numbered as 2 since the previous goal has already been proved, so it is removed from the list of «open» goals.

To the following command given by the user

SPLIT $\sum_{i=0}^{n+1} i$, n ;

the system answers

***** 2 $\sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + (n+1)$

Step 2 is added to the proof without dependencies (since it only depends on the definition of summation which is part of the basic knowledge of the system).

Then the mathematician asks the system to substitute the left hand side of the induction hypothesis by its right hand side in step 2.

SUBST 1 IN 2 ;

***** 3 $\sum_{i=0}^{n+1} i = \frac{n(n+1)}{2} + (n+1)$ (1)

where (1) means that step 3 depends on step 1.

Goal 2 is proved if the following proposition is proved :

GOAL $\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)((n+1)+1)}{2}$;

***** GOAL # 3 $\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)((n+1)+1)}{2}$

This can be achieved by simplifying it.

SIMPL GOAL # 3 ;

***** GOAL # 3 PROVED $\frac{n(n+1)}{2} + (n+1) = \frac{(n+1)((n+1)+1)}{2}$

LABEL HELP ;

Now by the command

TRANS 3 HELP ;

which applies the transitivity of the equality between step 3 and the lemma named HELP the proof of goal 2 is achieved.

***** 4 $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$ (1)

***** GOAL # 2 PROVED $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$

ASSUME $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

LABEL INDUCTION ;

Now the command

INDUCT BASIS, INDUCTION, n ;

completes the proof.

***** 5 $\forall n. \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

***** GOAL # 1 PROVED $\forall n. \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

Goal 1 is stated. The induction hypothesis has been discarded : the theorem holds without hypotheses.

In order to obtain the above simple theorem more than ten commands have been typed by the mathematician. The amount of details to be communicated to the machine is discouraging. But something better can be done. The mathematician could have the system to divide the theorem into lemmas, instead of doing it by himself. The proof would then have been the following.

GOAL $\forall n.\ \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$;

***** GOAL # 1 $\forall n.\ \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

TRY 1 INDUCT n ;

***** GOAL # 1 # 1 $\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$

***** GOAL # 1 # 2 $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$ ASSUME $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

TRY 1 SIMPL ;

***** GOAL # 1 # 1 PROVED $\sum_{i=0}^{0} i = \frac{0(0+1)}{2}$

TRY 2 SPLIT n ;

***** 1 $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$ (1)

***** GOAL # 1 #2 # 1 $\sum_{i=0}^{n} i + (n+1) = \frac{(n+1)((n+1)+1)}{2}$

TRY 1 SIMPL BY 1 ;

***** GOAL #1 #2 #1 PROVED $\sum_{i=0}^{n} i + (n+1) = \frac{(n+1)((n+1)+1)}{2}$

***** GOAL #1 #2 PROVED $\sum_{i=0}^{n+1} i = \frac{(n+1)((n+1)+1)}{2}$ ASSUME

$\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

***** GOAL # 1 PROVED $\forall n.\ \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

The command SIMPL ... BY 1 means : use equality contained in step 1 to make simplifications.

This gives an idea of the present performance of PPC. It already represents a big improvement on a system implementing an unadorned logic. What we are trying to achieve is something like

GOAL $\forall n.\ \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$;

***** GOAL # 1 $\forall n.\ \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

TRY 1 INDUCT ;

***** GOAL # 1 PROVED $\forall n.\ \sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

If we look at the above three proofs of the theorem we see that the main difference between the first and the second one consists in the ability of the system to manage the proof tree. The user is freed from the tiresome work of keeping track of lemmas and putting them together for deducing new steps. The gap between the second and the third proofs is a bit more difficult to be filled, since an accurate analysis of the commonly used proof strategies has to be done. They have to be automatized into a sort of proving algorithm which might also use some heuristics. For instance, when looking for the variable to induct on. The third proof is very similar to the one that may be produced by an automatic theorem prover. The underlying philosophy is, however, quite different. In PPC it is the user who decides the strategy and splits the proof into elementary steps. The complexity of them depends on the available proof generation technology. As already noted, improvements in this technology may be obtained from the experience gained in the proof generation activity.

In designing PPC we took advantage of the experience we gained in the massive use of the proof checker available at Stanford, while proving properties of the programming language PASCAL and of PASCAL programs.

Our aim is to design a system which makes the teletype (or the display) an «intelligent» tool which really helps the matematician in carrying out proofs. For this reason we think that a very flexible subgoaling mechanism is very useful. In fact the user must have the possibility of trying a strategy, abandoning it, trying a new one. Then, for instance, having got a new insight into the problem, he may return again to a previous trial with new ideas of how to go on with that proof. The system is responsible of managing all incomplete proofs and putting together the «right steps» of a proof that had success. This feature required a completely new goal structure for the proof checker.

We consider PPC a system in evolution : we think that a proof checker must grow with the experience of its users. For this reason we have implemented it in a modular way, so that additions and modifications can be made quite easily and independently of other parts of the program. In particular other logics may be easily added since all may rely on the common management level of the system. Several logics can operate simultaneously, thus allowing the user to switch to the right one for his specific problem.

In this paper we do not get into many details about the program which realizes PPC. A user's manual and a description of the program are in preparation. To give just a rough idea we may say that PPC is controlled by a looping routine which reads a command, executes it and then prints a message. First, each command typed by the user is analyzed. Then, if the syntax is correct, checks are done to see if the inference we are trying to do is a legal one. In such a case a new step is added to the internal structure which represents the proof. Messages from PPC consist in displaying pieces of the internal structure of the proof, by means of printing routines.

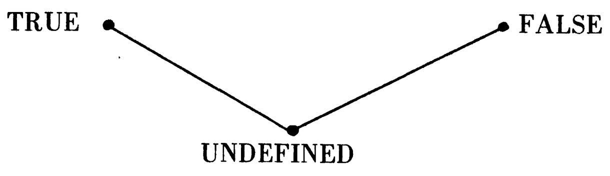
We hope we have given an idea of how the system works. Certainly we have not given any idea of how fast it works. We may say that, if no big simplifications are involved, it takes more time to the user to type the command than to the system to answer.

DESCRITPION OF LCF

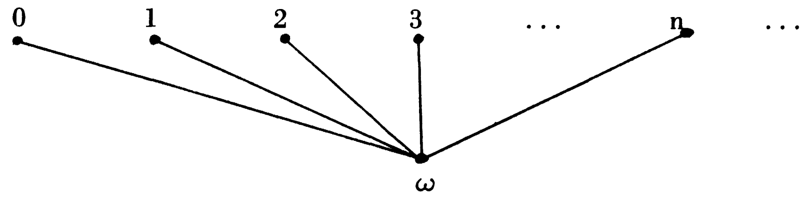
As we have already specified, the logic implemented so far in PPC is the Logic for Computable Functions (LCF). It has been proposed by Dana Scott in 1969 in an unpublished paper {8} and is described in detail in { 9 }. LCF is a typed version of Church's λ -calculus { 10}, augmented by some constructs to increase its expressive power. LCF was proposed by Dana Scott before he found a model for the type-free λ -calculus {11} .

Let us informally describe LCF. Our aim is to speak about a hierarchy of functional spaces, each of them ordered by a suitable ordering relation formalizing the concept «less defined than» which is recurrent in Mathematical Theory of Computation.

We start with two complete partial orderings (i.e. partially ordered sets having a least element such that every monotonically increasing sequence contained into them has a least upper bound) : the complete partial ordering (c.p.o.) of truth values $D_{\underline{tr}}$ and the c.p.o. of individuals $D_{\underline{ind}}$. The first one contains only three elements : TRUE, FALSE and UNDEFINED which are ordered as it is shown in the figure below (the order relation «goes upwards»).



The second one is completely arbitrary. For instance it can be the following c.p.o. of natural numbers augmented by an «undefined» natural number ω



or the set of all computable functions with the following order relation : f < g iff for all x such that f(x) is defined also g(x) is defined and f(x) = g(x).

The c.p.o.'s $D_{\underline{tr}}$ and $D_{\underline{ind}}$ are the basic spaces in the hierarchy. Higher order spaces can be built using the following rule : if D_β and $D_{\beta'}$ are spaces already present in the hierarchy, then $D_{(\beta \rightarrow \beta')} = \{D_\beta \rightarrow D_{\beta'}\}$, which is the space of all continuous functions (i.e. monotonically increasing functions preserving least upper bounds of monotonically increasing sequences) mapping D_β into $D_{\beta'}$, is also in the hierarchy. It is easy to show that if D_β and $D_{\beta'}$ are c.p.o.'s then $\{D_\beta \rightarrow D_{\beta'}\}$ is also a c.p.o. if the order relation and least upper bounds are taken pointwise.

The above hierarchy of spaces is described by LCF. Let $D_{\underline{ind}}$ be given (note that the choice of $D_{\underline{ind}}$ completely determines the hierarchy).

First of all we introduce <u>types</u>.

i) <u>tr</u> and <u>ind</u> are types, denoting the type of $D_{\underline{tr}}$ and $D_{\underline{ind}}$ respectively.

ii) If β and β' are types, then $(\beta \rightarrow \beta')$ is a type denoting the type of $D_{(\beta \rightarrow \beta')} = \{D_\beta \rightarrow D_{\beta'}\}$.

We shall write $\beta_1 \rightarrow \beta_2 \rightarrow \ldots \beta_{n-1} \rightarrow \beta_n$ as a shorthand for $(\beta_1 \rightarrow (\beta_2 \rightarrow \ldots \rightarrow (\beta_{n-1} \rightarrow \beta_n)\ldots))$.

We now introduce typed <u>terms</u> (we write $t : \beta$ for denoting a term of type β).

i) The following non logical constants are terms :

- $T : \underline{tr}$ which denotes TRUE $\varepsilon D_{\underline{tr}}$;

- $F : \underline{tr}$ which denotes FALSE $\varepsilon D_{\underline{tr}}$;

- $U_\beta : \beta$, for every type β , which denotes the least element of D_β (in particular, $U_{\underline{tr}}$ denotes UNDEFINED $\varepsilon D_{\underline{tr}}$) ;

- $C_{\underline{tr} \rightarrow \beta \rightarrow \beta \rightarrow \beta} : \underline{tr} \rightarrow \beta \rightarrow \beta \rightarrow \beta$, for every type β , which denotes the <u>conditional function</u> $\xi_\beta \ \varepsilon \{D_{\underline{tr}} \rightarrow \{D_\beta \rightarrow \{D_\beta \rightarrow D_\beta\}\}\}$ defined as follows :

$$\xi_\beta\,(x)(y)(z) = \begin{cases} y & \text{if } x = \text{TRUE} \\ z & \text{if } x = \text{FALSE} \\ \text{the least element of } D_\beta & \text{if } x = \text{UNDEFINED} \end{cases}$$

(note that ξ_β is continuous for every β) ;

- $Y_{(\beta \rightarrow \beta) \rightarrow \beta} : (\beta \rightarrow \beta) \rightarrow \beta$, for every ($\underline{tr} \neq \beta \neq \underline{ind}$), denotes the <u>least fixpoint operator</u> $FIX_\beta \ \varepsilon \ \{\{D_\beta \rightarrow D_\beta\} \rightarrow D_\beta\}$ defined as follows :

$$FIX_\beta\,(f) = \text{the least } x \text{ such that } x = f(x)$$

(the existence of a fixpoint operator FIX_β and its continuity are guaranteed by a theorem due to Knaster and Tarski).

ii) For every type β there exists a countable set of <u>variables</u> of type β : all of these variables are terms. They are intended to denote arbitrary objets in D_β . We do not give these variables explicitely : we shall refer to them by the metavariables x, y and z.

iii) If $s : \beta_1 \to \beta_2$ and $t : \beta_1$ are terms then $s(t) : \beta_2$ is a term denoting the value of the function denoted by s when applied to the argument denoted by t. Terms of this kind are called applications.

iv) If $x : \beta_1$ is a variable and $s : \beta_2$ is a term, then $(\lambda\ x.s) : \beta_1 \to \beta_2$ is a term denoting the function generated by the object denoted by s when all possible denotations in D_{β_1} are assigned to the variable x. Terms of this kind will be called λ-abstractions. The special constant λ binds variables in a way similar to the quantifiers of the predicate calculus.

The following shorthands will be used through the paper (s, t and u will be used as metavariables for terms) :

i) $s(t_1, ..., t_n)$ for $s(t_1(t_2 ... (t_{n-1}(t_n)) ...))$

ii) $\lambda\ x_1 ... x_n.s$ for $(\lambda\ x_1.(\lambda\ x_2. ... (\lambda\ x_n.s) ...))$

iii) $\lambda\ x.s(t)$ for $(\lambda\ x.s(t))$

iv) $(s \to t,u)$ for $C_{\underline{tr} \to \beta \to \beta \to \beta}(s \to t,u)$

v) $(\alpha\ f.s)$ for $Y_{(\beta\to\beta)\to\beta}(\lambda\ f.s)$

vi) U for U_β

Formulas of LCF are defined as follows.

i) If $s : \beta$ and $t : \beta$ are terms then $s < t$ is an atomic formula which is true if the object denoted by s is less than (in D_β) the object denoted by t, false otherwise.

ii) If $P_1 ... P_n$ are atomic formulas then $\{P_1, ..., P_n\}$ is a formula which is true if all P_i are true, false otherwise.

Note that the above notions of truth and falsity of formulas depend on the choice of $D_{\underline{ind}}$.

If P and Q are formulas then $P \vdash Q$ is a sentence which is valid if Q is true in all hierarchies in which P is true.

The following shorthands will be used for formulas and sentences (P and Q will be used as metavariables for formulas) :

i) $\forall x_1 ... x_n.s < t$ for $\lambda\ x_1 ... x_n.s < \lambda x_1 ... x_n.t$

(this notation is consistent with the pointwise definition of the order relation for higher-order spaces)

ii) $s::t < u$ for $(s \to t, U) < (s \to u, U)$

(the symbol :: plays the role of a rudimentary implication : the above formula can be read «if s is TRUE then $t < u$ else nothing can be said»)

iii) $s \equiv t$ for $\{s < t, t < s\}$

iv) $\vdash Q$ for $P \vdash Q$ when P is empty

v) $P, s < t \vdash Q, s' < t'$ for $\{ P_1, ..., P_n, s < t \} \vdash \{ Q_1, ..., Q_m, s' < t' \}$
if $P = \{ P_1, ..., P_n \}$ and $Q = \{ Q_1, ..., Q_m \}$

The axioms and inference rules of the logic are now presented (axioms are given as rules with no premises).

INCL $$\frac{P \vdash \{ Q_1, ..., Q_n \}}{P \vdash \{ Q_i \}} \qquad i = 1, ..., n$$

CONJ $$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 \cup P_2 \vdash Q_1 \cup Q_2}$$

CUT $$\frac{P_1 \vdash Q_1 \qquad P_2 \vdash Q_2}{P_1 \vdash Q_2}$$ if every atomic formula in P_2 is also in Q_1

APPL $$\frac{P \vdash t < u}{P \vdash s(t) < s(u)}$$ (the dual rule $$\frac{P \vdash t < u}{P \vdash t(s) < u(s)}$$ is easily derivable in LCF)

REFL $$\frac{}{\vdash s < s}$$

TRANS $$\frac{P_1 \vdash s < t \qquad P_2 \vdash t < u}{P_1 \cup P_2 \vdash s < u}$$

MIN1 $$\frac{}{\vdash U < s}$$

MIN2 $$\frac{}{\vdash U(s) < U}$$

CONDT $$\frac{}{\vdash (T \to s, t) \equiv s}$$

CONDF $$\frac{}{\vdash (F \to s, t) \equiv s}$$

CONDU $$\frac{}{\vdash (U \rightarrow s,t) \equiv U}$$

ABSTR $$\frac{P \vdash s < t}{P \vdash \lambda x.s < \lambda x.t}$$

ALPHACONV $$\frac{}{\vdash \lambda x.s \equiv \lambda y.\{y/x\}s}$$

$\{y/x\}$ s denotes the result of substituting y for all free occurrences of x in s, provided that no free occurrence of x in s is transformed into a bound occurrence of y in s

BETACONV $$\frac{}{\vdash (\lambda x.s)(t) \equiv \{t/x\}\ s}$$

$\{t/x\}$ s denotes the result of substituting t for all free occurrences of x in s : possible conflicts between free variables of t and bound variables of s are resolved by suitable renamings (alpha-conversions)

ETACONV $$\frac{}{\vdash \lambda x.s(x) \equiv s}$$ if x is not free in s

CASES $$\frac{P_1, s \equiv T \vdash Q \qquad P_2, s \equiv F \vdash Q \qquad P_3, s \equiv U \vdash Q}{P_1 \cup P_2 \cup P_3 \vdash Q}$$

FIXP $$\frac{}{\vdash Y(x) \equiv x(Y(x))}$$

INDUCT $$\frac{P_1 \vdash \{U/x\}\ Q \qquad P_2 \cup Q \vdash \{s(x)/x\}\ Q}{P_1 \cup P_2 \vdash \{Y(s)/x\}\ Q}$$

To informally describe the INDUCT rule, let us suppose $P_1 = P_2 = P$. If we want to prove that Q holds (depending on P) for the least fixpoint (denoted by Y(s)) of a function (denoted by s) in D_β , it is sufficient to prove that Q holds (depending on P) for the least element (denoted by U) of D_β and that, whenever Q holds for some object (denoted by x), it also holds (depending on P) for the object denoted by s(x). The soudness of the INDUCT rule is assured by the following observations.

i) U, s(U), s(s(U)), ..., s(s(s ... (s(U)) ...)), ...
denotes a monotonically increasing sequence.

ii) Y(s) denotes the least upper bound of the above sequence (this easily follows from the Knaster-Tarski theorem cited above).

iii) The terms contained in Q denote continuous functions, hence least upper bounds are preserved.

It is evident that LCF is strongly inspired by Church's λ-calculus {10}. Since in the λ-calculus with types it is not possible to define all computable functions, a fixpoint operator has been added to allow recursive definitions. The FIXP and INDUCT rules allow proofs about them. Definitions by cases are often required, so conditional terms are added - they play the role of the construct «if ... then ... else ...» which is present in most programming languages. The CASES, CONDT, CONF and CONDU rules allow proofs to be done by performing a case analysis on the truth value of predicates.

Extensions of LCF to include some predicate calculus and an enrichment of the type structure may be found in {12}.

EXAMPLES

In this section, we present some sample proofs actually run on the available system. They are taken from the Mathematical Theory of Computation.

We first put into the system the definitions of some functions which denote the semantics of the statements of a simple but complete programming language : i.e. «<u>if</u> ... <u>then</u> ... <u>else</u>», sequencing of statements and «<u>while</u>» statement (see for instance {5}).

$$\text{COND} \equiv \lambda p\ f\ g\ s.p(s) \rightarrow f(s), g(s)$$

$$\times \equiv \lambda f\ g\ s.g(f(s))$$

$$\text{WHILE} \equiv \lambda p\ f.(\alpha\ g.\text{COND}(p, f \times g, \text{ID}))$$

where

$$\text{ID} \equiv \lambda\ x.x$$

The above functions define the semantics of programming language constructs denotationally, as a mapping from stores into stores. For example, COND applied to a test p and a couple of functions f and g is a map from stores into stores. The operator × denotes the composition of functions. Note that the order of composition respects the order of sequencing of statements in programming languages, i.e.

$$f \times g \equiv \lambda\ s.g(f(s))$$

Note also that binary operators may be declared to be infix. If × is declared to be infix, we may write f × g instead of × (f,g), while × (f) is still a legal term, which may be proved to be equal to λ g s.f(g(s)).

Besides the above functions, the semantics of the programming language is defined by an axiomatization of the store and by appropriate functions for storing values into it and evaluating boolean and arithmetic expressions.

Let F, G, H denote statements and F;G the compund statement obtained by concatenating F and G. In the first example, we prove the following : if, whenever the execution of G terminates also the execution of H terminates yielding the same result, then the same holds for the compound statements F; G and F ; H. Let f, g, h be the functions denoting the meaning of F, G, H. Then what we should prove is

$$\text{if } g < h \text{ then } \forall f.f \times g < f \times h$$

The command we give is :

```
GOAL  ∀f.f× g< f × h        ASSUME  g< h ;

  ***** GOAL  # 1    ∀f.f × g < f × h      ASSUME  g< h
```

This goal can be achieved by the tactic ABSTR, which is the converse of the abstraction rule, i.e. an instantiation. The proof is performed as follows :

```
TRY 1 ABSTR ;
***** 1 g < h   (1)
***** GOAL #1 #1   f× g < f × h
APPL ×(f) 1 ;
***** 2 f×g < f ×h    (1)
***** GOAL #1 #1 PROVED f× g < f× h
***** GOAL #1 PROVED ∀f.f× g < f× h  ASSUME g < h
```

which completes the proof.

The second example is a simple property about conditional statements, namely :

if P then if P then A else B else C

is equivalent to

if P then A else C

Let p, a, b, c denote the meaning functions associated to P, A, B, C. Since COND is the meaning function of the statement «if ... then ... else», the goal to be proved is :

GOAL $\forall$p a b c.COND(p, COND(p,a,b),c) $\equiv$ COND(p,a,c) ;

The calculus is extensional, so the above formula is a mapping from stores into stores, even if the variable s does not appear explicitly in it. In fact, the above formula and

$\forall$p a b c s.COND(p, COND(p,a,b), c)(s) $\equiv$ COND(p,a,c)(s)

are interderivable in LCF. The system gets the goal and answers

```
***** GOAL #1 ∀p a b c.COND(p, COND(p,a,b),c) ≡ COND(p,a,c)
```

By using the tactic SIMPL it is possible to save some tedious steps of the proof such as :

i) performing all possible β-reductions ;

ii) applying the rules of inference of the SIMPSET, until possible ;

iii) substituting terms for terms according to the equivalence formulas of the SIMPSET, until possible.

SIMPSET is a set of inference rules. Usually it contains CONDT, CONDF, CONDU, MIN1 and MIN2, but it can be modified by the user and may change during a proof. Also the equivalence formulas of SIMPSET are set by the user. In particular, function definitions are often put into the SIMPSET in order to have names substituted by the corresponding definitions.

Suppose we have put the definition of COND into the SIMPSET. Then by the command

TRY SIMPL ;

we get

```
***** GOAL  # 1  # 1  λ p a  b  c  s.(p(s)→(p(s)→ a(s), b(s)), c(s))≡
                      λ p a  b  c  s.(p(s)→ a(s), c(s))
```

This goal can be achieved by the tactic ABSTR, and then by performing a case analysis on the predicate p(s).

```
TRY ABSTR ;
*****  GOAL #1 # 1# 1  (p(s)→ (p(s) → a(s), b(s)), c(s)) ≡
                       (p(s)→ a(s), c(s))
TRY CASES p(s) ;
*****  GOAL  # 1 # 1 #1 #1  (p(s)→ (p(s) →a(s), b(s)), c(s)) ≡
                            (p(s)→a(s), c(s))        ASSUME  p(s) ≡ T
*****  GOAL  # 1 #1  # 1 #2  (p(s) →(p(s)→  a(s),b(s)), c(s)) ≡
                             (p(s) → a(s), c(s))     ASSUME  p(s)  ≡ U
*****  GOAL  # 1 # 1 # 1 #3  (p(s)→  (p(s)→ a(s), b(s)), c(s)) ≡
                             (p(s)→  a(s), c(s))     ASSUME  p(s) ≡  F
```

The above three goals are now proved by simplifying them. The assumption about the truth value of p(s) is automatically put into the SIMPSET associated with the corresponding goal. In this way we have

```
TRY 1 SIMPL ;
*****  1 p(s) ≡ T    (1)     ASSUME
*****  GOAL  # 1 # 1 # 1 # 1   PROVED  (p(s)→ (p(s)→  a(s), b(s)), c(s)) ≡
                                       (p(s)→  a(s), c(s))
                               ASSUME  p(s) ≡ T
TRY 2 SIMPL ;
*****  2   p(s)  ≡ U    (2)
*****  GOAL   # 1  # 1 # 1 # 2  PROVED  (p(s)  → (p(s)  → a(s), b(s)), c(s)) ≡
                                        (p(s)  →a(s), c(s))
                                ASSUME  p(s)  ≡ U
TRY 3 SIMPL ;
*****  3 p(s) ≡ F     (3)
*****  GOAL   # 1 #1 # 1 # 3  PROVED  (p(s) →(p(s)  → a(s), b(s)), c(s)) ≡
                                      (p(s) →a(s), c(s))
                              ASSUME  p(s)  ≡ F
```

The system now backs up on the goal tree : every subgoal has been proved, hence the theorem is achieved.

```
***** GOAL # 1 # 1 # 1  PROVED  (p(s)→ (p(s)→a(s),b(s)), c(s))≡
                                 (p(s)→ a(s), c(s))

***** GOAL # 1 # 1   PROVED  λ p a b c s.(p(s)→(p(s)→ a(s),b(s)),c(s))≡
                             λ p a b c s.(p(s)→a(s), c(s))

***** GOAL # 1  PROVED  ∀ p a b c.COND(p,COND(p,a,b),c) ≡ COND(p,a,c)
```

We now present an example showing how an already proved assertion may be transformed into a theorem which may then be used as a tactic.

The theorem we prove first is :

F < G   if   F ≡ α g.FUN(g)   and   G ≡FUN(G)

The two hypotheses of the theorem (steps 1 and 2) are assumed. Then the goal is tackled by induction. Two subgoals are generated. The first one (basis of the induction) is established by simplification. The second one (induction step) is proved by using the monotonicity of the functions we are dealing with and the transitivity of the relation < .

```
ASSUME  F ≡ α g.FUN(g),  G ≡ FUN(G) ;
  ***** 1  F ≡ αg.FUN(g)    (1)
  ***** 2  G ≡FUN(G)   (2)
GOAL  F < G ;
 ***** GOAL  # 1   F< G
TRY 1  INDUCT 1 ;
 *****  GOAL  # 1  # 1    U< G
 ***** GOAL  # 1 # 2   FUN(g)< G      ASSUME  g< G
TRY 1  SIMPL ;
 ***** GOAL  # 1  # 1   PROVED  U < G
TRY 2 ;
 ***** 3  g < G     (3)
APPL FUN 3 ;
 ***** 4   FUN(g) < FUN(G)    (3)
SYM 2 ;
 *****  5   FUN(G) ≡ G   (2)
TRANS 4 5 ;
 *****  6  FUN(g) < G     (2  3)
```

```
***** GOAL  # 1 # 2   PROVED   FUN(g) < G     ASSUME  g< G
***** GOAL  # 1   PROVED  F < G
```

The above result can be established as a theorem by giving it a name.

```
THEOREM MINIMALITY ;
*****  THEOREM MINIMALITY  F < G   ASSUME   F ≡α g.FUN(g), G ≡ FUN(G)
```

We now use this theorem in a proof of equivalence of two different terms denoting the semantics of the <u>while</u> statement.

The first two commands are given to assume the two definitions of the semantics of the <u>while</u> statement. Then the goal is put into the system. It is first split into two subgoals.

```
ASSUME  WHILE1 ≡ αg.( λ q  f.COND(q,f × g(q,f),ID)) ;
*****  1 WHILE1  ≡α g.(λ  q  f.COND(q,f  ×g(q,f),ID))
ASSUME  WHILE2 ≡ λ q  f.(α  g.COND(q,f × g,ID)) ;
*****  2  WHILE2 ≡ λ q   f.( α g.COND(q,f × g,ID))
GOAL  WHILE1≡  WHILE2 ;
*****  GOAL   # 1   WHILE1 ≡ WHILE2
TRY  1  HALF ;
*****  GOAL  # 1 # 1   WHILE1 < WHILE2
*****  GOAL  # 1 # 2   WHILE2 < WHILE1
```

Only the first goal is now proved. The proof of the second one is analogous.

```
TRY  1  USE MINIMALITY   FUN ← λ h  q   f.COND(q,f× h(q,f),ID) ;
```

This command tells the system to use the theorem MINIMALITY to prove the goal by instantiating the metavariable FUN appearing in the theorem to be the term λ h q f.COND(q,f ×h(q,f), ID). Two subgoals are generated, corresponding to the two assumptions of the theorem.

```
*****   GOAL   # 1 #1# 1  WHILE1 ≡  αg.( λ h  q  f.COND(q,f × h(q,f),ID))(g)
*****   GOAL   # 1 #1 #2  WHILE2 ≡ (λ h  q  f.COND(q,f× h(q,f),ID)) (WHILE2)
```

The two subgoals are now proved in a fairly standard way. The first one is got by simplification.

```
TRY  1  SIMPL ;
*****   GOAL  # 1  #1  # 1  PROVED   WHILE1 ≡ α g.(λ h  f  g.COND(q,f× h(q,f),ID))(g)
```

The second goal is got by simplifying it and then by a number of instantiations and an application of the rule FIXP.

```
TRY  2  SIMPL ;
*****  GOAL   # 1 # 1 # 2 # 1   WHILE2 ≡ λ q   f.COND(q,f ×WHILE2(q,f),ID)
```

```
TRY 1 ABSTR ;
***** GOAL  # 1 # 1 # 2 # 1 # 1  WHILE2(q,f) ≡ COND(q,f × WHILE2(q,f),ID)
SAPPL 2 q f ;
***** 3  WHILE2(q,f) ≡α g.COND(q,f × g,ID)
FIXP 3 ;
***** 4  WHILE2(q,f) ≡ COND(q,f ×WHILE2(q,f),ID)
***** GOAL  # 1 # 1 # 2 # 1 # 1  PROVED  WHILE2(q,f) ≡ COND(q,f× WHILE2(q,f),ID)
***** GOAL  # 1 # 1 # 2 # 1  PROVED  WHILE2 ≡ λq f.COND(q,f× WHILE2(q,f),ID)
***** GOAL  # 1 # 1 # 2  PROVED  WHILE2 ≡ (λ h q f.COND(q,f × h(q,f),ID)(WHILE2)
***** GOAL  # 1 # 1  PROVED  WHILE1 < WHILE2
```

The command SAPPL is equivalent to a command APPL followed by a SIMPL.

CONCLUSIONS

In the section devoted to examples we have chosen short meaningful proofs using different strategies. Certainly we have not given an idea of the dimensions of proofs that can be performed. We have not yet built big proofs in PPC, but we may make reference to the experiments reported in {13} and {5} about the construction of proofs on the LCF available at Stanford. We think that PPC will eventually allow to build even larger proofs.

Even though the whole project originated by an experience on proving properties of programs, we do not want PPC to be just a tool for providing programs with a warranty of correctness. We want it to be a formal framework in which to design programming languages by defining their semantics and then discussing their properties in the form of theorems with a machine checked proof.

Besides the insight we hope to have into the theory of programming, we expect to learn which patterns of proofs are more often used and to be able to add them easily to the system as automatic strategies. In this way it is possible to have a system that actually helps the mathematician since it may grow as the experience of the mathematician grows.

REFERENCES

1 - J.A. ROBINSON - *A machine oriented logic based on the resolution principle* - Journal of the ACM - Vol. 12 - 1965.

2 - R. EHRENFEUCHT, M. RABIN - *There is no perfect proof procedure* - Unpublished paper - Underground Jerusalem - 1972.

3 - R. MILNER - *Logic for computable functions : description of a machine implementation* - Artificial Intelligence Memo No 169 - Stanford University - 1972.

4 - R. WEYHRAUCH, A. THOMAS - *FOL : a proof checker for first order logic* - Artificial Intelligence Memo No 235 - Stanford University - 1974.

5 - L. AIELLO, M. AIELLO, R. WEYHRAUCH - *The semantics of PASCAL in LCF* - Artificial Intelligence Memo No 221 - Stanford University - 1974.

6 - M. AIELLO, R. WEYHRAUCH - *Checking proofs in the metamathematics of first order logic* - Artificial Intelligence Memo No 222 - Stanford University - 1974 - Also in Proc. of the Fourth International Joint Conference on Artificial Intelligence - Tbilisi - 1975.

7 - C. MONTANGERO, G. PACINI, F. TURINI - *MAGMA-LISP : a «machine language» for artificial intelligence* - Proceedings of the Fourth International Joint Conference on Artificial Intelligence - Tbilisi - 1975.

8 - D. SCOTT - *An alternative approach to CUCH, ISWIM, OWHY* - Unpublished paper - Underground Princeton - 1969.

9 - R. MILNER - *Models of LCF* - Artificial Intelligence Memo No 186 - Stanford University - 1973.

10 - A. CHURCH - *The calculi of λ-conversion* - Annals of Mathematical Studies No 6 - Princeton - 1941.

11 - D. SCOTT - *Continuous lattices* - Technical Monograph PRG-7 - Oxford University - 1971.

12 - R. MILNER, L. MORRIS, M. NEWEY - *A logic for computable functions with reflexive and polymorphic types* - Proceedings of the International Symposium of Proving and Improving Programs - Arc et Senans - 1975.

13 - M. NEWEY- *Formal semantics of LISP with applications to program correctness* - Thesis - Artificial Intelligence Memo No 257 - Stanford University - 1975.