ON PROVING LAWS OF THE ALGEBRA OF FP-SYSTEMS IN EDINBURGH LCF

Jacek Leszczyłowski
Polish Academy of Sciences
Institute of Computer Science
P.O.BOX 22, 00-901 Warszawa PKiN, POLAND

I INTRODUCTION

J.Backus, in CACM 21/8, defined a class of applicative programming systems called FP /functional programming/ systems in which a user has: 1.objects built recursively from atoms, UU /an undefined element/ and objects by a strict /i.e. a UU-preserving/ "list" operator, 2. elementary functions over objects, 3. tools for building functions out of already defined functions.

One can think of a machine support while working with FP systems and proving facts about FP systems as well as facts concerning the functions being defined. The choice of EDINBURGH LCF is rather natural because it is an interactive computer system /implemented in LISP/ for reasoning about functions /see [6]/. It consists of two parts. The first part is a family of calculi each of which is characterized by four factors: 1. type operators /representing domains in the sense of Scott's theory; see [11]/, 2. constants /representing continuous functions/, 3. axioms, 4. inference rules. One of them, PPLAMBDA, is given as the "initial" calculus, and other calculi may be built by users as extensions of existing calculi. The second part is a high level programming language ML which is fully higher order and is strongly typed. Its polymorphic types make it as convenient as typeless languages.

This paper is a short report on the application of EDINBURGH LCF to proving the laws of the algebra of FP systems listed by Backus in [1]. Actually, we generalized FP-systems and the laws are formulated in stronger form than it was done by Backus. We briefly describe /sec.II/ the style of proving with the system, then /sec.III/ comment the strtegies used in the proofs giving only their specifications. The summing up remarks will be given in sec.IV. More detailed report on the project is given in [9].

II STYLE OF PROVING

As mentioned there are inference rules associated with each of the calculi of the system; the inferece rules of PPLAMBDA are primitive, and derived rules may be programmed by users. The inference rules are represented as ML-functions taking theorems /being a data type in ML/ as arguments and giving values which are theorems as well; an example is the computational induction rule INDUCT /it is the Scott induction rule; for more detailes see [2]/. We could prove our theorems applying the inference rules in appropriate order but it is not a convenient style of proving.

We base our proofs on partial subgoaling methods, called tactics; these mean that given a formula to be proved we want to transform it into "simpler" formulae /which in turn have to be proved/ and the proof justifying the "transformation". The system can support this kind of proving via predefined types: goal, tactic and proof, defined as follows:

```
goal = form # simpset # form list
proof = thm list -> thm
tactic = goal -> goal list # proof
```

The first element of the Cartesian product defining the type goal is for stating the formula which is going to be proved, the third one for listing assumptions, the second one is /from the user's point of view/ an abstract type consisting of simplification rules; these are /possibly conditional/ equivalences of terms to be used as left-to-right rewriting rules.

We shall explain now the use of the subgoaling methods. Let us define when a theorem $A' \vdash f'$ /A'- hypotheses, f'- conclusion/ achieves a goal f,ss,A. This is the case when, up to renaming of bound variables, f' is identical with f and each member of A' is either in A or is in the hypotheses of a member of the simplification set ss . Then, we say, a theorem list achieves a goal list if the first element of the theorem list achieves the first element of the goal list, etc. Thus, a tactic T will work "properly" if for any goal g and any goal list gl and any proof p such that

$$T(g) = gl,p$$

if we have a theorem list thml which achieves the goal list gl , then p(thml) will achieve the goal g . An important special case is when gl is empty for then we need only apply p to the empty theorem list - i.e. evaluate p(nil) to obtain a theorem achieving our original goal g .

We shall use two of the standard tactics. The first is SIMPTAC; applied to any goal (w,ss,A) , SIMPTAC produces a singleton list of goals

$$[(w',ss,A)]$$

and proof p , where w' is the simplification of w by rewriting rules in ss and p justifies all the simplifications made. In the case where w' is a tautology the goal list is null. The second one is CONDCASESTAC ; for any goal (w,ss,wl) it

finds a term t of the type tr /truth values domain/ which is free in w and occurs as the conditional expression and then produces three subgoals with formula w as the their first element and the simplification sets extended by the new assumptions /cases when t equal UU , truc and false respectively/.

Now we introduce the last tool for desigining proofs: tacticals - mechaninsms for combining tactics to form larger ones. We shall use only one of the standard tacticals of the system called THEN For any tactic T1 and T2 , the composed tactic T1 THEN T2 applies T1 to the goal and then applies T2 to all resulting subgoals produced by T1 ; the proof function returned is the composition of the proof functions produced by T1 and T2 .

III COMMENTS ON THE PROOFS

There are tree groups of the proofs of the laws of the FP-system algebra listed by Backus in [1]. The first one is based on SIMPTAC ; for example, to prove II.1 of [1] we used the tactic: APTAC THEN SIMPTAC where APTAC is one of the pro grammed tactic and is specified as follows:

```
APTAC ("u =<v",ss,wl) =
      ["!X. u X =  v X",ss,wl] , p
```

where: ! - universal quantification
= < stands for equality or inequality of terms
X - a new variable

The second group of proofs is based on CONDCASESTAC . The composed tactic used in the proofs was:

APTAC THEN CONDCASESTAC THEN SIMPTAC .

For example in proofs of the laws: II.2 and II.3.1 presented in [1] .

The third group of proofs involves the use of the computational induction rule INDUCT in an inderect way. The proofs are based on the programmed tactic INDTAC which is an "inverse" of the structural induction rule over the type of lists which in turn is derived from the computational induction rule INDUCT . The specification of INDATC is the following:

```
INDTAC ("!L.w[L]",ss,wl) =
       [("w[UU]",ss,wl);
        ("w[nil]",ss,wl);
        ("w[cons X L]",ss´,wl)]      p
```

where:
X - a new variable
ss´ is ss extended by the assumption w simplified by ss
p uses the structural induction rule on lists derived form INDUCT
nil and cons are the constants over lists.

Let us present one of the proofs using INDTAC Suppose we want to prove:

"ToALL(G o F) = (ToALL G) o (ToALL F) "

where:
o - the composition of functions
ToALL is takes two arguments: H - being a function and a list and produces a list of the results and the following axioms are satisfied:

```
"!H. ToALL H UU  =  UU "
"!H. ToALL H nil =  nil "
"!H.!X.!L. ToALL H (cons X L) =
           cons (H X) (ToALL H L)"   .
```

From the shape of subgoals produced by INDTAC we know that we need to simplify the formulae by the above axioms as well as the definition of the composition operator. Suppose we created the simplification set ss including the desired rewriting rules; now we can specify our goal in the following way:

```
g = "ToALL (G o F) = (ToALL G) o (ToALL F)",ss,nil
```

Let T = APTAC THEN INDTAC THEN SIMPTAC be a tactic we want to use to tackle our problem. If we apply our tactic T to our goal g we get a pair as result. Let us store the value on the variables p and gl ; we can do it in ML in the following way:

```
gl,p  :=  T g
```

It terns out that our tactic was fully successful and the system respond that gl is the empty list. Thus , see sec.II, we can apply p to the empty theorem list and the produced value is the theorem we wanted to prove.

In general cases when we are not so clever, the subgoal list is not empty and we have to apply another tacic to solve the subgoals.

IV FINAL REMARKS

As we mentioned in sec.I the aim of the paper was to present an application of EDINBURGH LCF But the aim of any application itself is to find general tactics which can be used in totally different examples. Why? Because EDINBURGH LCF is essentially a tool placed somewhere between a theorem prover and a proof checker; this is why we cannot rely on the built in strategies which is the case with the theorem provers but we are interested in looking for general purpose ones which when found can be programmed in ML and an be used to tackle other goals. The tactic used to prove the laws of the FP-system algebra are of a ganeral use and were inwolved in proving properties of the functions defined in FP-systems; see [9]. The generalization

of FP systems is briefly described in [12] and presented in more detailed version in [9] . Examples of more powerful and complex tactics can be found in [8] .

Let us compare EDINBURGH LCF with other implemented systems. On one hand the explicite presence of the logic of the system makes EDINBURGH LCF "human-oriented" and easy to extend. On the other hand, for example, the Boyer-Moore theorem prover /see [3]/ is very efficient in its use of built in strategies, but difficult to extend ; by contrast the need to conduct all inferences through the basic inference rules /as ML-procedures/ , which appears necessary if we wish to allow users to extend the system reliably by programming leads to some inefficiency in LCF. This we have found tolerable and indeed it can be reduced significantly by direct implementation of ML /which at present is compiled into LISP / . Another way of making the system quick is by running it on multiprocessor machines which is done for example at Royal Technical University , Stockholm, Sweden . For a nice general comparison of these two system see [5] .

The EDINBURGH LCF style of proving which consists in solving the problems by means of programmed proof strategies seems to be natural. It took the author 2 months to be able to work with the system. This style fits pretty well to doing large proofs with machine assistance. By this we mean neither that a large proof is submitted step by step and merely checked by the machine /see [7] /, nor that the system discovers the large proof by itself, but that the problem may be split into smaler parts, each of which is tackled semiautomatically by a subgoaling method. A nice example of such application of EDINBURGH LCF is the compiler correctness proof presented in [4] .

ACKNOWLEDGEMENTS

I wish to thank Avra Cohn, Mike Gordon, Robin Milner and Chris Wadsworth for their friendly help during my stay in Edinburgh and especially Robin Milner for his support while preparing the draft version of the paper.

REFERENCES

[1] Backus J. "Can programming be liberated from the von Neumann style? A functional programming style and its algebra of programs", Comm ACM 21,8 , 1978 .

[2] Bird R., "Programs and Machines; an introduction to the theory of computation", Wiley 1976

[3] Boyer R.S., Moore J S., "A computational Logic" Academic Press, New York 1979 .

[4] Cohn A., "High level proof in LCF", Proc. 4th Workshop on Automated Deduction , Austin , Texas, 1979 .

[5] Cohn A., "Remarks on Machine Proof", manuscript, 1980 .

[6] Gordon M., Milner R., Wadsworth C., "EDINBURGH LCF", Springer Verlag , 1979 .

[7] van Benthem Jutting L.S., "Checking Landau's 'Grundlagen' in the AUTOMATH system", Tech. Hoghschule, Eidhoven, The Netherlands, 1977 .

[8] Leszczyłowski J., "An experiment with EDINBURGH LCF" , Proc. CADE-5 , Les Arsc, France , 1980 .

[9] Leszczyłowski J., "Theory of FP systems in EDINBURGH LCF", Internal Report, Comp. Sci. Dept., Edinburgh University, Edinburgh, Scotland, 1980 .

[10] Milner R., "LCF: a way of doing proofs with a machine", Proc. MFCS-8 Symposium , Olomouc, Chechoslowakia, 1979 .

[11] Milner R., "Implementation and application of Scott's logic for computable functions", Proc. ACM Conference on Proving Assertions about Programs, SIGPLAN Notices , 1972 .

[12] Leszczyłowski J., "EDINBURGH LCF supporting FP systems", Proc. Annual Conference of the Geselschaft fur Informatik, Universitat des Saarlandes, 1980 .