Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

78

Michael J. Gordon
Arthur J. Milner
Christopher P. Wadsworth

Edinburgh LCF

A Mechanised Logic of Computation

Springer-Verlag
Berlin Heidelberg New York 1979

Authors

Michael J. Gordon
Arthur J. Milner
Christopher P. Wadsworth
Dept. of Computer Science
University of Edinburgh
James Clerk Maxwell Building
The King's Buildings
Mayfield Road
Edinburgh EH9 3JZ
Great Britain

AMS Subject Classifications (1970): 68C01, 68E15
CR Subject Classifications (1974): 5.21, 5.27

ISBN 3-540-09724-4 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-09724-4 Springer-Verlag New York Heidelberg Berlin

Library of Congress Cataloging in Publication Data.
Gordon, Michael J C 1948-
Edinburgh LCF : a mechanised logic of computation.
(Lecture notes in computer science; 78)
Bibliography: p.
Includes index.
1. Edinburgh LCF (Computer system) 2. Computable functions--Data processing.
I. Milner, Robin, joint author. II. Wadsworth, Christopher P., joint author. III. Title.
IV. Series.
QA9.59.G.67 510'.8s [001.6'4] 79-24745
ISBN 0-387-09724-4

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

Preface

Edinburgh LCF is a computer system for doing formal proofs interactively. This book is both an introduction and a reference manual for the complete system (and its DECsystem-10 implementation). The acronym LCF stands for "Logic for Computable Functions" - a logic due to Dana Scott in which facts about recursively defined functions can be formulated and proved. The original system (developed at Stanford University) was a proof checker for this logic, based on the idea not of proving theorems automatically, but of using a number of commands to generate proofs interactively step by step. The emphasis then was on exploring the class of problems that could conveniently be represented in the logic, and on discovering the kinds of patterns of inference that arose when solving these problems. It was found that, by and large, the original logic was expressive enough, although a few useful extensions were suggested. However, the fixed repertoire of proof-generating commands often required long and very tedious interactions to generate quite simple proofs; furthermore these long interactions often consisted of frequent repetitions of essentially the same sequence of inferences.

From the experience gained, a new system - Edinburgh LCF - has been built. Instead of a fixed set of proof-generating commands, there is a general purpose programming language ML (for "metalanguage"). Among the primitives of this language are ones for performing atomic proof steps; since these are embedded in a programming language, sequences of them can be composed into procedures. Thus, where in Stanford LCF common patterns of inferences would have to be repeated, these now become programmed operations, defined once and then called many times (or even built into yet more complex operations).

ML is a functional language in the tradition of ISWIM and GEDANKEN. Its main features are: first, it is fully higher-order, i.e. procedures are first-class values and may be passed as arguments, returned as results or embedded in data-structures; second, it has a simple, but flexible, mechanism for raising and handling exceptions (or, in our terminology, for "generating and trapping failures"); and third, but perhaps most important, ML has an extensible and completely secure polymorphic type discipline. Imperative features, in particular an ability to introduce assignable storage locations, are also included; in practice, however, we have found these are rarely used, and it is not clear whether they were really necessary.

The inclusion of higher-order procedures stems from a desire to experiment with operations for composing proof strategies. Such strategies are represented by certain types of procedures; if ML were not higher-order, we would not be able to define many natural operations over strategies. Since strategies may fail to be applicable to certain goals, we also needed a mechanism for cleanly escaping from ones inappropriately invoked, and this led to the inclusion of exception handling constructs. These constructs have turned out to be both essential and very convenient.

The reason for adopting a secure type system is best seen by comparing the treatment of proofs in the present system and its predecessor. In Stanford LCF, a proof consisted of a sequence of steps (theorems), indexed by positive integers, each following from previous steps by inference. For example, if 50 steps have been generated, and the 39th step is

```
]- for all x. F
```

(for some logical formula F) then the command

```
SPEC "a+1" 39
```

will generate, by specialization, the 51st step as

```
]- F[a+1/x]
```

(i.e. F with the term "a+1" substituted for x). In Edinburgh LCF, instead of indexing proofs by numbers, theorems are computed values with metalanguage type thm, and may be given metalanguage names. (Other metalanguage types are term and form(ula) - e.g. "a+1" is a term, and "for all X. X+0=X" is a form). Thus if th names the theorem

```
]- for all x. F
```

the specialization rule may be invoked by the ML phrase

```
let th' = SPEC "a+1" th
```

which constructs a new step and names it th'. This change, whilst not profound, is very influential - the identifier SPEC now stands for an ML procedure (representing a basic inference rule) whose metalanguage type is (term->(thm->thm)), and it is a simple matter to define derived inference rules by ordinary programming.

There is nothing new in representing inference rules as procedures (for example PLANNER does it); what is perhaps new is that the metalanguage type discipline is used to rigorously distinguish the types thm, term and form, so that - whatever complex procedures are defined - all values of type thm must be theorems, as only inferences can compute such values (for example, since the type system is secure, the value "1=0" of type form can never aquire type thm). This security releases us from the need to preserve whole proofs (though it does not preclude this) - an important practical gain since large proofs tended to clog up the working space of Stanford LCF.

The emphasis of the present project has been on discovering how to exploit the flexibility of the metalanguage to organise and structure the performance of proofs. The separation of the logic from its metalanguage is a crucial feature of this; different methodologies for performing proofs in the logic correspond to different programming styles in the metalanguage. Since our current research concerns experiments with proof methodologies - for example, forward proof versus goal-directed proof - it is essential that the system does not commit us to any fixed style.

Much of our work on proof methodologies is independent of the logic in which the proofs are done, and so the acronym LCF is perhaps inappropriate for the complete system. However since the present logic is quite similar to the original one - though it incorporates extensions suggested by the experiments at Stanford - we have felt justified in continuing to call the system "LCF". One important extension, both conceptual and practical, is that the logic itself is now no longer a fixed calculus but a family of deductive calculi (called PPLAMBDA), with facilities for introducing and axiomatizing new types and new constants. Collections of types, constants and proved theorems are called theories, and these can be organised into a hierarchical data-base. A typical theory contains the axiomatization of a particular problem area, and is in general built on other theories.

Several case studies have been done (see Bibliography), including a fairly substantial proof of a compiler. This proof was based upon a published informal proof, which was found incorrect when its formalization was attempted within our system. Reports of current and future experiments will also be published. We hope that the complete system description given here, in conjunction with reports of particular studies, will guide the design of future proof generating systems.

How to read this document.

If you are reading for general interest, and not intending to use the system, then Chapter 1 and Sections 2.1 and 3.1 give a quick overview, and Appendix 1 shows the creation of a simple theory. Section 3.1 has enough pointers to other parts of the text to enable you to discover our intended style of proof, in which three important ingredients are theories (3.4), simplification (Appendix 8) and goal-directed proof (1.1 and 2.5).

If you are only interested in ML as a programming language, then it is worth noting that Section 2 is completely independent of PPLAMBDA, and is supported by Appendices 3, 6 and 10.

If you wish to use Edinburgh LCF, first get the overview as suggested above; next, read Appendix 11 to see how to run the system, then perform the examples in 3.1 with any variations that you can think of; then study at least Appendix 5 (inference rules), 3.4 (theories), 2.5 with Appendix 9 (goals and tactics) and Appendix 1 (an example of theories) before going on to your own proofs; at this point the whole document should serve as a reference manual.

Acknowledgements

Many people have helped us in this work. First and foremost, Lockwood Morris and Malcolm Newey worked intensively on the project in its early stages and are responsible for much of what we are now describing; if it had been feasible, they should have been co-authors.

We are indebted to Dana Scott for providing a large part of the theoretical basis for our work: to John McCarthy for encouraging the forerunner of this project at Stanford; to Richard Weyhrauch who with Newey contributed greatly to that project; to Tony Hoare for his probing criticism of a draft of this document; to both him and Jerry Schwarz for help with the notion of abstract types in ML; to Avra Cohn for help in the final design stages through her experiments; to Jacek Leszczylowski for his helpful criticisms of the final draft; and to Rod Burstall, Gordon Plotkin and many colleagues in Edinburgh for illuminating discussions. Finally, we are indebted to the Science Research Council for supporting, and to our department for tolerating, a project which has taken longer to mature than we hoped.

Table of Contents