THE UNIVERSITY *of* EDINBURGH

This thesis has been submitted in fulfilment of the requirements for a postgraduate degree (e.g. PhD, MPhil, DClinPsychol) at the University of Edinburgh. Please note the following terms and conditions of use:

- This work is protected by copyright and other intellectual property rights, which are retained by the thesis author, unless otherwise stated.
- A copy can be downloaded for personal non-commercial research or study, without prior permission or charge.
- This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author.
- The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.
- When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Machine Assisted Proofs of Recursion Implementation

Avra Jean Cohn

Doctor of Philosophy
University of Edinburgh
1979

Abstract

Three studies in the machine assisted proof of recursion implementation are described. The verification system used is Edinburgh LCF (Logic for Computable Functions). Proofs are generated, in LCF, in a goal-oriented fashion by the application of strategies reflecting informal proof plans. LCF is introduced in Chapter 1.

We present three case studies in which proof strategies are developed and (except in the third) tested in LCF. Chapter 2 contains an account of the machine generated proofs of three program transformations (from recursive to iterative function schemata). Two of the examples are taken from Manna and Waldinger. In each case, the recursion is implemented by the introduction of a new data type, e.g., a stack or counter. Some progress is made towards the development of a general strategy for producing the equivalence proofs of recursive and iterative function schemata by machine.

Chapter 3 is concerned with the machine generated proof of the correctness of a compiling algorithm. The formulation, borrowed from Russell, includes a simple imperative language with a _while_ and conditional construct, and a low level language of labelled statements, including jumps. We have, in LCF, formalised his denotational descriptions of the two languages and performed a proof of the preservation of the semantics under compilation.

In Chapter 4, we express and informally prove the correctness of a compiling algorithm for a language containing declarations and calls of recursive procedures. We present a low level language whose semantics model a standard activation stack implementation. Certain theoretical difficulties (connected with recursively defined relations) are discussed, and a proposed proof in LCF is outlined.

The emphasis in this work is less on proving original theorems, or even automatically finding proofs of known theorems, than on (i) exhibiting and analysing the underlying structure of proofs, and of machine proof attempts, and (ii) investigating the nature of the interaction (between a user and a computer system) required to generate proofs mechanically; that is, the transition from informal proof plans to behaviours which cause formal proofs to be performed.

Contents

Introduction 1

Background 8

Chapter 1: Introduction to Proof in LCF 16

- The Meta Language ML 17
- The Logic PPLAMBDA 21
- The Interface of ML to PPLAMBDA with an Example 24
- Tactical Proof 28

Chapter 2: Proofs of Recursion Removal Schemata 41

- The Accumulator Problem 42
- The List Stack Problem 50
  - The Problem 50
  - The Formalisation 55
  - The List Stack Proof in LCF 58
- The Counter Problem 71
  - The Problem 71
  - The Formalisation 75
  - The Counter Proof in LCF 78
- Conclusions: Towards a General Schema Tactic 83
- Notes 88

Chapter 3: The Russell Compiler 91

- The Problem 93
- The Informal Proof 98
- The Proof in LCF 106
  - Theory Structure for the Proof 106
  - Lemma Structure for the Proof 115

The Machine Proof 124

Conclusions 131

Notes 133

Chapter 4: Implementation of Procedure Declaration 139

The Problem 140

The High Level Language 140

The Low Level Language 142

The Compiler 147

The Equivalence Proofs 148

Standard to Closure Semantics Proof ($\mathcal{S}$ to $\bar{\mathcal{S}}$) 149

Closure to Abstract Stack Semantics Proof ($\bar{\mathcal{S}}$ to $\mathcal{D}$) 153

Abstract Stack to Concrete Stack Semantics Proof ($\mathcal{D}$ to $\mathcal{Run}$) 160

Summary 172

Speculations on Performing the Proofs in LCF 175

Theory Structure for the Proofs 175

Tactics for the Proofs 179

Notes 189

Conclusions 193

Note on Computational and Structural Induction 193

General Conclusions 196

Future Work 207

Appendix: Some Technical Details 210

Details of the List Stack Proof in LCF 211

Implementing Iterated Induction 221

Bibliography 225

Introduction

This work represents several explorations in a methodology and technology for the generation of formal proofs of program correctness. Underlying the work is the belief that it is important to supply complete and correct formal proofs of program correctness, as informal proofs may suffer errors in both logical structure and technical detail. Also implicit is the belief that because the proofs of even simple programs are long and complex, any hope for producing the proofs rests in the design of computer systems which share the task of proof generation with human theorem provers.

The components of our model of user-system cooperation are (i) the effort required on behalf of the user, at <u>least</u> in stating the goals, and possibly, in specifying proof methods, strategies and insights; (ii) the facility for interaction between user and system, enabling the user to communicate goals and possibly strategies to the system, and the system to report the results of its proof attempts to the user, and (iii) the capacity of the system for recognising valid proofs, and possibly, for generating proofs automatically.

Proof systems of various sorts fit this framework, as it is rather general; for example, automatic theorem proving systems, in which (i) the user states the problem to be solved, (ii) the logic in which the problems are stated provides the basis for user-system interaction, and (iii) built-in heuristics endow the system with its capacity for automatic proof. Examples of automatic theorem provers for programs are the Boyer-Moore system for proving theorems about LISP functions [3], and Pratt's system for proving algorithms

written in dynamic logic [40]. The model also includes proof checking systems, in which (i) the user is required to perform the proof, to some degree of refinement, (ii) the specification of a proof step is the basis for user-system interaction, and (iii) the system is able to do proofs automatically to the extent required to perform (and hence check) the specified steps. Proof checkers for programs include Stanford LCF (Logic for Computable Functions) [26,27,50], the Pisa Proof Checker (PPC) [2], and the FOL (First Order Logic) system [9,49]. The framework includes, in addition, standard verification systems based on Floyd's method of inductive assertions [10], and Hoare's proof rules [16]. The Stanford Pascal Verifier [21], and the PL/CV system [7] are two modern examples. In such systems, the user's contribution is, typically, a program in a (fixed) language, annotated at points in the text with assertions which are intended to hold whenever control reaches those points, during evaluation of the program. In the two instances mentioned, the fixed languages are PASCAL and PL/1 subsets, respectively. The Stanford Pascal Verifier relies on a theorem prover, and the PL/CV system on a proof checker, for the proofs that the assertions do in fact hold. Interaction with the system, in the former case, includes a facility for enabling the user to suggest useful facts to the theorem prover; in the latter case, interaction is as in standard proof checking systems.

It is useful to consider two further dimensions along which machine proof systems can be classified, besides the nature of the interaction required to produce proofs. The first is <u>generality</u>. It can be observed that some systems are designed for reasoning in a

particular area, about a particular programming language, or within a certain logical framework, while others are intended to cope with quite general sorts of reasoning. Standard verification systems, for example, are typically built around one particular programming language (and the proof rules for that language) and are tailored for reasoning about programs in that language, within the Floyd-Hoare framework. The FOL system, in contrast, aims at providing an environment in which purely mathematical and 'common sense' arguments can be conducted, as well as arguments about programs of various sorts.

The second dimension is security. Some systems do not ensure that only valid deductions can be performed. This applies to many standard verification systems, in which the absence of an explicit logic means that there is no a priori notion of a valid deduction; hence the security of inferences is left to the user and is not checked by the system. In contrast, systems which rely on explicit logics and which insist on fully checked proofs, relative to those logics, do guarantee security (that is, as long as the logics are consistent). Stanford LCF and the FOL system fall into the second category.

The technology on which we have relied in this work is the Edinburgh LCF system [13,14,15,29,30]. In regard to user-system interaction, LCF is distinguished from conventional automatic theorem proving and proof checking systems by the fact that its interaction facility is a programming language. In this language, goals to be proved and theorems already proved are represented as objects of distinct data types, and strategies for performing proofs

are represented as procedures. A standard set of strategies for performing certain routine proofs steps is provided; beyond that, the extent to which proofs can be performed automatically in LCF is determined by strategies designed and implemented by the user.

In the context of LCF, and of our model of user-system interaction, some more subtle distinctions can be made. We have mentioned automatic theorem proving systems, in which the emphasis is on the system's ability to find proofs, as well as systems in which the emphasis is on the system's ability to check proofs. One can also distinguish proof performing and proof generating systems. A proof which is performed is not necessarily produced as a complete object in the end, but may exist only as a historical sequence of steps which have been evaluated. When we speak of generating proofs, we refer to behaviours on behalf of the user which cause proofs to be performed or produced. In LCF, the user generates proofs, and proofs are performed in the system. The 'extreme' styles of proof finding and proof checking can be accomodated in LCF, but are not necessarily imposed, or even preferred.

As regards the other two dimensions of generality and security, LCF is fairly, but not completely general, and it is completely secure. It is based on a typed lambda calculus logic in which all types correspond to some complete partial ordering (cpo) and is therefore oriented toward reasoning about areas which fall within the framework of Scott-Strachey denotational semantics. Classes of LCF studies have concerned single programming languages (a study of PASCAL and its implementation, in Stanford LCF [1], relations between different semantics for the same language (direct and

continuation semantics) [31], relations between several languages (see Chapters 3 and 4, following, and [6]), recursive functions in general (see Chapter 2, following), and various data types (a study of lists, for example [11]).

It is fundamental to the LCF 'philosophy' that the production of correct, complete formal proofs is vital; in LCF it is ensured that non-valid proofs cannot be produced, even as the result of applying user-defined strategies to goals.

The aims of the work presented herein have been to study, in the context of LCF and of several program correctness proofs, the 'quality' of the interaction required between user and system to perform proofs; to propose methods of organising and structuring large proof efforts; to investigate ways in which informal proof plans can be mirrored by procedures in a programming language; to research the extent to which a user can be isolated from the actual sequence of primitive inference steps which constitute a proof; to test the naturalness and effectiveness of the goal-oriented and strategy-driven style of proof generation; and to isolate patterns of inference for various classes of problems. We feel that these issues are of general interest and applicability, even though the work is intimately tied to the LCF system.

Two remarks pertain to the connection of this work with LCF. Firstly, it is important to be clear about what is original to the author. That includes neither any part of the LCF system, nor the underlying concept of proof generation by the application of tactics which reflect informal inference plans. (It does include the various proofs planned and/or performed in LCF.) Nonetheless, we

have devoted the first chapter to an account of LCF and the methodology of proof generation therein. Although we do not attempt a complete exposition, Chapter 1 enables the subsequent three chapters to be read without continual reference to other documents. None of the material in Chapter 1, at any rate, (excepting the simple example, and some notation) is original to the author in any way.

Secondly, although we report several proof efforts using LCF, we have endeavoured, in this presentation, to concentrate on those aspects of the efforts which address the research aims mentioned, rather than the 'proof engineering' aspects. In this spirit, we have not, in general, included the code of programs, transcripts of interactions with LCF, or statistics about the actual proof performances. (Some material of this sort may be found in the Appendix.)

In Chapters 2, 3 and 4 we give accounts of two actual (and one hypothetical) proof efforts using LCF. The common thread of the problems is the implementation of recursively defined functions. In Chapter 2, we consider the equivalence proof for three pair of recursive and iterative function schemata, and outline a general strategy for proving such equivalences in LCF. In Chapter 3, we verify, in LCF, a compiler for a high level language which includes a while construct. The formulation of the problem is borrowed from Russell [42]. The approach is to supply denotational semantics for the two languages involved, to represent the compiler as a function acting on the abstract syntax of the high level language, and to prove the preservation of the semantics under compilation. In

Chapter 4, we employ a similar approach in stating and informally proving the correctness of a compiler for a block-structured language allowing recursive procedure declaration and invocation. We cope with certain theoretical problems in the proof (to do with recursively defined relations), and outline a proposed machine proof, based on the results of Chapters 2 and 3.

In each chapter, we describe the formalisation of the problem; we present the informal proof (which is usually roundabout, as one is comparing differently structured computations when implementing recursion -- the first section of the Conclusions contains a discussion of the proof methods used); and we give an account of the (actual or proposed) machine proof effort. We conclude with an analysis of the three experiments, and an assessment of our methodology of proof generation.

Background

Since the correctness of programs is clearly relative to their intended meanings, this work rests upon the field of programming language semantics. We have relied, here, on the mathematical and descriptive aspects of denotational semantics, semantics in which programs and the objects from which they are constructed correspond to abstract entities. In an indirect sense, we have used dentotational semantics by using LCF, since, as we have indicated, LCF is designed primarily for reasoning in the setting of denotational semantics. More directly, in Chapters 3 and 4, we use denotational definitions of the source and target languages under consideration. We do not attempt a survey of or introduction to denotational semantics here, but merely acknowledge our debt. Useful references are [12,25,43,44,45,46].

More specifically, we also acknowledge work done on the verification of implementations in a denotational setting by Milne, and Milne and Strachey [24,25], and presented in much simplified form by Stoy [45]. While not claiming mastery of Milne's work, it is clear that the present work deals with some of the same issues, in particular (i) the factoring of the compilation of recursive procedure declaration into stages, includings a closure semantics (`store semantics` in Milne) and a stack semantics, and (ii) the problem of the recursively defined relations that arise naturally in the statements of equivalence of semantics at different levels of abstraction.

Reynolds, too, has studied the problem of recursively defined relations ('directed complete relations') [41]. A sequence of semantics at decreasing levels of abstraction was also proposed by Burstall and Landin [5], in the context of a simple expression compiler and of algebraic proofs.

Other (algebraic) methodologies for formulating and proving compiler correctness have been developed by Morris [34,35] and by the ADJ group [47]. We do not elaborate on these, as they are somewhat outside of the scope of this work.

The work on proving program transformations, in Chapter 2, is based on examples given by Manna and Waldinger [22], although it is perhaps fair to say that the examples are common property. While a great deal of research has concentrated on the problem of discovering and automating program transformations (e. g. by Burstall and Darlington [4], and Darlington and Waldinger [8]) we know of little on formal correctness proofs. The most closely related work is by Huet and Lang [17], in which formal proofs are given for several pair of function schemata similar to the ones we have studied. Although Huet and Lang are rather more concerned with the problems of pattern matching involved in applying program transformations, they do stress the importance of supplying formal proofs, and even suggest LCF as a vehicle for producing the proofs.

The work described in Chapter 3 is based on (and inspired by) a formulation of the problem of compiler correctness by Russell [42]. He proposes a source and target language, gives denotational definitions of both, and specifies a compiling algorithm between them. We have attempted, in formalising the problem and performing

the proof in LCF, to retain as much as possible of his statement of the problem. The informal proof he gives is actually incorrect; evidence, we think, for the need for machine-checked correctness proofs. Nonetheless, we have found his formulation to be useful in isolating the problem of verifying the implementation of the while construct, as well as in avoiding the problem of the generation of new label names (something which complicated many earlier formulations).

The early formulations and proofs of compiler correctness (for schematic compilers in an abstract setting) predate the development of denotational semantics by several years, yet anticipate the role of semantics in the statements of correctness. The paradigm for much subsequent work in compiler correctness was a compiler for arithmetic expressions proposed by McCarthy and Painter in 1967 [23]. The problem consisted in compiling a language of constants and variables, and binary operations on them, into a language of `store`, `load` and `operate` instructions intended to be executed on an abstract, single-address machine with an accumulator. The important features of the formulation included (i) provision of (what is essentially) a denotational semantics for the expression language, based on an abstract state, (ii) an operational semantics for the machine language, based on the state of the machine, specifying how the execution of each instruction affects the state, (iii) reliance on a compiling algorithm rather than a compiler in a particular language, (iv) the use of abstract syntax and the consequent separation of the problem of proving parsers correct from the problem of proving code-generators, (v) the form of the

statement of correctness: if a high and low level state are suitably related, then the outcomes of evaluating a high level program, and of running its compiled image, in the respective states, are also suitably related, and (vi) the proof of the correctness of the compiler by induction on the structure of expressions in the language. The work was intended for eventual machine validation, and in fact, the problem has been used more than once as an exercise in machine proof. One such proof was performed by Milner and Weyhrauch [28], in Stanford LCF, as part of a larger compiler proof (which is discussed below).

Subsequent work on rigorous and machine proofs of compilers has diverged into two trains of research, dealing in turn with compilers for LISP-like and for Algol-like languages. This development is based on the relative natures of applicative and imperative languages. Compiler proofs, either machine-produced, partially machine-produced, or just amenable to machine proof, have been given for LISP subsets by (among others) London [19,20] and Newey [36], and for imperative languages of various sorts by (among others) Kaplan [18], Milner and Weyhrauch [28], and by Milne, and Russell, as mentioned earlier.

The LISP formulations are characterised, in general, by being more realistic; that is, they take real LISP as source and real LAP code as target. This is possible, in part, because of the comparitive simplicity of LISP and its implementations. In the imperative tradition, the languages used have tended to be contrived for the purpose of studying certain features.

The LISP compilers mentioned were taken to be actual programs written in LISP. In the imperative language studies mentioned, McCarthy's use of compiling algorithms and abstract syntax of the source language as starting points has been followed. Using a program rather than an algorithm adds another layer of proof to the problem, namely, a proof of the correctness of the compiler relative to the algorithm it denotes. (Newey gives an account of a proposed proof of this sort [36].)

Because, in some sense, the natural semantics for LISP is an interpretive (operational) semantics (based on the LISP `eval` function), the semantics used in stating the correctness of a LISP compiler is closer in structure to the semantics for LAP code than a denotational semantics for an imperative language would be to an operational semantics for the appropriate machine language. This would appear to make the correctness proofs easier, in the applicative case, and to circumvent problems, discussed by Milne and by Stoy, and encountered in Chapter 4, below, which arise in proving the equivalence of operational and denotational definitions. In addition, LISP's convention of dynamic rather than static binding of variables makes it unnecessary, in an implementation, to preserve declaration time environments of functions.

All of these factors help to explain why the compilation of LISP (or in general, applicative languages) is a rather different problem than the one in which we are interested at present, and we therefore do not go into detail about London's proof, or about Newey's proof (which was partially checked in Stanford LCF). The importance of Newey's work, from the current standpoint, lies in his

conclusions about the feasibility of performing large proofs mechanically, and his recommendations and suggestions about what would have had to be added to Stanford LCF to make the proof effort, in its entirety, feasible.

In relation to Edinburgh LCF, its predecesor, Stanford LCF, was based on a similar but more primitive logic, and did not include a programming language in which to express procedures for manipulating objects in the logic. It had only a few standard facilities for goal-oriented proof generation, and was essentially a proof checker. The Edinburgh LCF system was much influenced by Newey's conclusions about the need for a `high level command language` in which to conduct proofs, for improved abilities to do automatic proof, and for a more organised way of extending the basic logic with new constants and axioms.

Early work on the verification of compilers for imperative languages was done by Kaplan; he treated (informally) a language containing an assignment statement and a conditional construct, which was compiled into a language of `load` and `store` instructions for an abstract machine. Both languages were given an operational semantics, a compiling algorithm was presented, and a (very long) proof given. The proof was by recursion induction (a precursor of computational induction). The proof was, like McCarthy and Painter's, intended for eventual machine validation.

The work on compiler correctness proofs most relevant to the current work, and on which it is based, was done by Milner and Weyhrauch in the setting (again) of Stanford LCF. There, a high level language containing assignments, conditionals, <u>while</u>

statements and sequencing of statements, (forming a language very close to the one later treated by Russell and used here, in Chapter 3) was considered. A low level stack-manipulating language for an abstract machine was specified, and a denotational and operational semantics (respectively) were given for the high and low level languages. Effort was concentrated on _organising_ the problem for mechanical checking. Concepts from universal algebra were applied, to this end, and a structure of eleven subgoals was formed. Typical subgoals were to establish that the semantic functions and compiling function were homomorphisms. Proofs of seven of the subgoals were successfully checked in Stanford LCF.

The current work has built upon and continued the Milner-Weyhrauch project, both by (i) treating a very similar formulation of the compiler correctness problem, and (ii) making use of a proof generation system which was developed as a result of that research, and Newey's. As regards (i), we have used nearly the same high level language (in Chapter 3), but simplified the problem, following Russell's proposals, by dealing neither with expression compilation, nor with the generation of new label names in the target code. We too have given attention to the effort required to organise and structure the proof, but have chosen to use features of Edinburgh LCF, and other techniques, rather than to appeal to algebraic principles. As regards (ii), we have had the advantage of previous experience in the form of a much more sophisticated proof system, a proof _generation_ rather than a proof _checking_ system, in which strategies for performing proofs can be written and applied. Both Newey, and Milner and Weyhrauch concluded from their

experiments that the generation of formal compiler proofs <u>was</u> a feasible undertaking, but only in the context of the more advanved LCF system which was subsequently designed and implemented by Gordon, Milner, Morris, Newey and Wadsworth. We feel privileged to have had the advantage of all of the previous work on compiler proofs, particularly that done in Stanford LCF, and access to a system which makes proof efforts as desrcibed feasible -- and pleasant.

Chapter 1: Introduction to Proof in LCF

Edinburgh LCF, Logic for Computable Functions, is a system designed to assist in the interactive generation of formal, machine proofs, particularly in the areas of programming language semantics and recursive function theory. It is based on work by Scott and Strachey, [43,44,46], and on its forerunner, Stanford LCF, [26,27,50]. The current system was implemented in 1974-1979 by Gordon, Milner, Morris, Newey and Wadsworth [13,15].

LCF consists of two levels. The first is a logic called PPLAMBDA (for polymorphic predicate lambda calculus) in which properties of recursive functions and semantics can be conveniently stated. PPLAMBDA can be extended by the introduction of new logical types, constants and axioms, to form theories, in the usual logical sense. The terms of PPLAMBDA are as in the typed lambda calculus, and the formulae as in the predicate calculus.

PPLAMBDA is interfaced to a second level, a programming language, ML (for meta language), which is designed for referring to and manipulating objects in the logic. ML is used for programming procedures which generate proofs in PPLAMBDA. It is a general purpose, higher order language with a strict type discipline, a user-defined abstract type facility, and an exception handling mechanism.

In this chapter, we briefly introduce ML and PPLAMBDA, and illustrate, with an example, the concept of tactical proof. Fuller descriptions of ML, PPLAMBDA and tactical proof may be found in the LCF manual [15], and in [6,11,14,29,30,31,32,33].

The Meta Language ML

ML is a general purpose programming language whose type discipline provides the basis of its interface to the logic PPLAMBDA.

An ML expression, e, can take the following (main) forms:

e ::= ce	constant expressions, including the integers 0,1,..., and the truth values true and false
id	variables
e1 e2	application of (function) e1 to (argument) e2
<u>if</u> e <u>then</u> e1 <u>else</u> e2	conditional, where e evaluates to true or false
e1=e2	test for equality of expressions e1 and e2, returning a boolean value
d <u>in</u> e	a local declaration d (see below)
λv1...vn.e	lambda abstraction on the `variable structures` v1,...,vn (see below)
[e1;...;en]	list containing e1,...,en
<u>fail</u>	causes current evaluation to fail

Variable structures, v, may be:

v ::= () | id | v1.v2 | v1,v2 | [v1;...;vn]

for the empty variable structure, a simple variable, a constructed list of variables, a pair of variables, and a list containing the variables v1,...,vn.

As we have mentioned, ML has a type discipline which requires that all ML expressions (and variable structures) have an ML type. The implemenatation of ML includes a (compile-time) type-checker

which infers the types of objects, if a consistent type can be found for them. In addition, the types of expressions and variable structures may be constrained (to a type ty, say) by the following notation:

```
e ::= e:ty
v ::= v:ty
```

ML types are useful for debugging ML procedures, and they are essential in ensuring that ML procedures do not compute non-theorems. (This is discussed in the section after the next.)

ML types are given by:

$$ty ::= cty \mid vty \mid ty1 \times ty2 \mid ty1 \longrightarrow ty2 \mid (ty1,\ldots,tyn)\ id$$

Constant types, cty, include type constants such as int, for integer, and bool, for boolean value. (The ML constants 0, 1,... have type int, and true and false have type bool.) There are also several additional constant types specific to PPLAMBDA, which are discussed in the section after the next.

Types may also be type variables, vty, which we indicate with asterisks (e.g., *, **, etc.).

Compound ML types are built from other types using standard operators such as × and ⟶. ty1 × ty2 denotes the type given by the Cartesian product of ty1 and ty2; ty1 ⟶ ty2 is the type of functions from objects of type ty1 to objects of type ty2.

Finally, types can be built from standard or user-defined abstract type operators. An example of a standard (unary) type operator is list; the type ty list (for some type ty) is the type of

a list of objects of type ty.

Types which are constructed from type variables are called polymorphic types; an object with polymorphic type is said to have each substitution instance of the polymorphic type as its type. (See [15,32] for further discussion of polymorphism.)

ML declarations, d, include the forms:

d ::= let b | letrec b

for non-recursive and recursive declarations, respectively, where bindings, b, can be:

b ::= v=e | id v1 ... vn = e | b1 and b2 ... and bn

(where, in the third case, each bi must be of the first or second form). Bindings of the second form are equivalent to id = λv1...vn.e, so that they are really of the first form. Bindings of the third form effect several bindings at once.

In evaluating an expression containing a non-recursive declaration d, (let d in e, for example), e is evaluated in an environment in which d has been evaluated first. (Environment, here, means an association of identifiers with expression values.) The expression (letrec d in e) gives the recursive interpretation to variables in d. (Only functions may be defined recursively by the letrec construct.) A declaration d is evaluated by first evaluating its binding, b, to produce a new environment. A binding v=e is evaluated by evaluating the expression e, then attempting a pattern match between the value of e and the variable structure v (lists match lists of equal length if corresponding elements match,

identifiers match all expression values, and so on) and finally, extending the environment according to the list of identifier-expression value pairs determined by the match (if the match succeeds). A typical expression, in the following chapters, is

λ[x:int;y:int]. x,y

This denotes a function expecting a list of two integers, and returning an ordered pair of the two elements of the list; that is, the expression denotes a function of type int list$\longrightarrow$(int $\times$ int).

Many features of ML have been omitted or simplified in the exposition above. In particular, there are two additional forms of declaration in ML. The first is an abstract type or type operator definition, introducing a set of types or type operators whose representations are local to the declarations. These, like ordinary ML declarations, may be recursive or non-recursive. We do not provide details here, but refer the reader to [15], especially 2.4.5.

One can also abbreviate types; defined types are identifiers standing for other types. We add to the possibilities for declaration

d ::= <u>lettype</u> db

where defined type bindings db are

db ::= id1=ty1 <u>and</u> ... <u>and</u> idn=tyn

For example, one might write

lettype intpair = int × int

to save writing int × int.

In general, in this presentation, we try to avoid giving explicit ML expressions; instead, we attempt to convey the intention of the ML functions by description or diagram. Where we do list ML expressions, they will generally have the form

let d in e

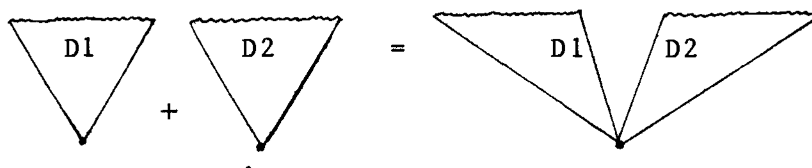
Finally, a variety of standard functions are supplied in ML, some for general list processing, and others for handling PPLAMBDA objects. Typical functions of the first sort are hd:* list ⟶ *, tl:* list ⟶ * list, and null:* list ⟶ tr, to take the head and tail of a list (of arbitrary type), and to test whether a list is empty.

The Logic PPLAMBDA

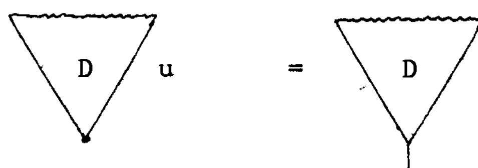
PPLAMBDA is a typed logic in which formulae are built up in the usual ways from terms. Just as all expressions in the programming language ML have ML types, so all terms in PPLAMBDA have PPLAMBDA types. Each PPLAMBDA type is taken to denote a domain (complete partial order, or cpo) with a minimum (least defined) element. PPLAMBDA types, type, are given by:

type ::= c | id | type1 + type2 | type1 × type2
| type1 ⟶ type2 | type u

for type constants (including the type tr for PPLAMBDA truth values), type variables (which, like ML type variables, are written with asterisks), and types which are constructed by the binary type operators +, × , and ⟶, or the unary type operator u. +, × , and ⟶ correspond to the sum, product and function space operators on domains. u corresponds to the 'lifting' operator on domains, which adds to a domain a new minimum element. It should be noted that + means coalesced sum; the corresponding domain operator can be depicted as:



that is, the sum for which the minimum elements of D1 and D2 are identified. The domain operator corresponding to u can be depicted as:



The separated sum, ++, say, can clearly be be expressed in terms of + by use of the lifting operator: D1 ++ D2 = D1 u + D2 u.

The terms t of PPLAMBDA are given by:

t ::= c | id | t1 t2 | λv.t | t⇒t1|t2 | t1,t2 | t:type

for constant terms, variables, application of t1 to t2 (a term which can only be constructed if t1 has functional type *⟶**, say,

relative to which t2 has type *), lambda abstraction to a bound variable v, conditionals (where t must have type tr, and the types of the alternatives t1 and t2 are the same), ordered pairs, and the constraining of the type of a term. The notion of type polymorphism in PPLAMBDA is similar to that in ML.

Constant terms, c, include the following terms, given with their constant or polymorphic types (* and ** are type variables):

c ::=	TT	truth value true	tr
	FF	truth value false	tr
	$\perp$	minimum (undefined) element	*
	FIX	the least fixed point operator	$(* \rightarrow *) \rightarrow *$
	FST	function to select the first element of a pair	$(* \times **) \rightarrow *$
	SND	function to select the second element of a pair	$(* \times **) \rightarrow **$
	INL	functions to inject elements of appropriate type into sum domains	$* \rightarrow (* + **)$
	INR		$** \rightarrow (* + **)$
	OUTL	functions to project elements of appropriate type out of sum domains	$(* + **) \rightarrow *$
	OUTR		$(* + **) \rightarrow **$
	ISL	to test whether an element is in the left summand of a sum domain	$(* + **) \rightarrow$ tr
	UP	to lift and lower domains	$* \rightarrow (*\ u)$
	DOWN		$(*\ u) \rightarrow *$
	DEF	to determine whether an element is defined (returns TT if so and $\perp$ otherwise)	$* \rightarrow$ tr

These constants are axiomatised in LCF by rules of inference; rules of inference are discussed in the next section.

PPLAMBDA formulae, f, are given by:

$$f ::= \mathrm{TRUTH} \mid t \equiv t' \mid t \sqsubseteq t' \mid f \,\&\, f' \mid f \;\mathrm{IMP}\; f' \mid \forall v1 \ldots vn.f$$

That is, the tautology formula TRUTH, equivalences or inequivalences of terms (in the sense of the ordering $\sqsubseteq$ over the domain corresponding to the type of t and t′), and conjunctions, implications, and universal quantifications as in the predicate calculus.

In addition, PPLAMBDA can be extended by the introduction of new types and type operators, new constants having these types, and new axioms (as discussed in the next section) to form LCF <u>theories</u>. The LCF theory facility enables the user to incrementally develop and preserve theories, and to construct hierarchies of theories in which each theory inherits from an ancestor all of the types, constants, axioms and proved facts of that ancestor. In this manner, the objects and theorems needed in the formulation of problems in LCF can be neatly organised and made accessible, rather than being introduced in an *ad hoc* or behind-the-scenes way. We illustrate the use of LCF theories in the following chapters.

The Interface of ML to PPLAMBDA with an Example

The interface is achieved by three additional constant ML types (as well as a parser for concrete PPLAMBDA syntax). The types are term, form and type, to represent PPLAMBDA terms, formulae and

types. These could, in theory, be introduced as abstract types, but they are provided as basic types for convenience and efficiency. Other PPLAMBDA objects are defined in terms of these; for example, theorems, in sequent style, are represented by the type form list × form, that is, a list of hypotheses paired with a conclusion. The type thm, for theorem, admits various rules of inference as operations. <u>Only</u> the rules of inference associated with the type thm can produce results of the type thm; the type-checker for ML expressions ensures this. Thus, modulo the soundness of the rules of inference, only valid theorems can be returned by ML functions.

Among the functions provided in ML for handling objects in PPLAMBDA are the following abstract syntax functions:

mkequiv:(term × term) ⟶ form	for constructing equivalences
destequiv:form⟶ (term × term)	for taking equivalences apart into pairs of terms
rhs:form⟶ term	for selecting the right hand side of an equivalence or an inequivalence
lhs:form⟶ term	for selecting the left hand side of an equivalence or an inequivalence
destcomb:form⟶ (term × term)	for taking applications apart apart into pairs of terms
isbottom:term ⟶ bool	for testing for ⊥ (of any type)

All of these functions fail when inapplicable.

We illustrate some of the ideas presented thus far, and some of the intended uses of LCF, with an example. Following are three PPLAMBDA rules of inference: TRANS, APTHM and MINAP. We write them

below in the natural deduction format of PPLAMBDA inference. The concept of a rule of inference, it should be noted, does not correspond to a particular ML type, since different rules have different types. A rule of inference always takes some number (possibly zero) of theorems, curried or paired, as arguments, and produces a theorem as result. The types of these three are:

TRANS:(thm × thm) ⟶ thm

APTHM:term ⟶ thm ⟶ thm

MINAP:term ⟶ thm

(In fact, the type of APTHM is actually thm ⟶ term ⟶ thm, but for convenience, we have reversed the order of the arguments in this exposition; we prefer to place non-theorem parameters first.)

Here and throughout, we use the following notation for a theorem with a list of hypotheses A and a conclusion w: $A \vdash w$. (Occasionally, though, we do not list the hypotheses.) We denote rules of inference by drawing a line, and writing the theorem returned by the rule below, and the theorem arguments of the rule above. The names of the rules are shown above each diagram, applied to the non-theorem arguments, if there are any, as in the second and third rules below. Some rules have no theorem arguments, e.g., MINAP. Such rules are sometimes referred to as axiom schemes. ($\cup$ denotes union.)

TRANS

$$\frac{A1 \vdash t \sqsubseteq u \qquad\qquad A2 \vdash u \sqsubseteq v}{A1 \cup A2 \vdash t \sqsubseteq v}$$

APTHM t

$$\frac{A \vdash u \sqsubseteq v}{A \vdash u\ t \sqsubseteq v\ t}$$

MINAP t

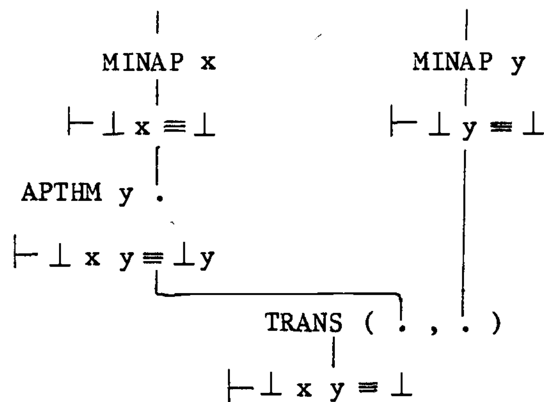
$$\frac{}{\vdash \bot\ t \equiv \bot}$$

These are the rules for the transitivity of $\sqsubseteq$, the monotonicity of application, and the minimality of $\bot$, respectively.

Suppose that we wish to prove that $\bot\ x\ y \equiv \bot$, for all x and y. To do this, we could evaluate

```
TRANS (APTHM y (MINAP x),
       MINAP y)
```

The structure of this proof can be displayed as a tree, in which nodes are theorems and arcs represent the application of rules of inference, as indicated:



A proof done in this fashion is called a <u>forward proof</u>.

Tactical Proof

In contrast, let us consider the following heuristic for proving any formula of the form $t\ x \equiv \perp$:

> When trying to prove that $t\ x \equiv \perp$, try proving as a subgoal that $t \equiv \perp$.

There are three observations to be made about this heuristic, besides the fact that it solves our goal, $\perp x \equiv \perp$ (since $\perp \equiv \perp$ follows by reflexivity). Firstly, the heuristic will not always 'work'; consider a formula $(\lambda x.x)\perp \equiv \perp$, for which the heuristic suggests an _a priori_ unachievable subgoal. Secondly, if the subgoal _can_ be proved, then the original goal, $t\ x \equiv \perp$, can also be proved, by application of the function

λth:thm. TRANS(APTHM x th, MINAP x)

to the theorem corresponding to (achieving) the subgoal:

$\vdash t \equiv \perp$

The application produces a theorem $\vdash t\ x \equiv \perp$. Thirdly, applying the heuristic n-1 times to a formula of the form $\perp$ x1 x2 ... xn $\equiv \perp$ yields a subgoal $\perp$ x1 $\equiv \perp$, which is in turn proved by evaluating MINAP x1. Thus we can use the same heuristic (repeatedly) for solving more complex goals.

A _tactic_ is a defined type in ML for representing strategies such as the one above. Tactics are functions which generate subgoals, given goals, _and_ which provide mappings from achievements

of subgoals to achievements of goals. We write, in ML,

<u>lettype</u> tactic = goal→ (goal list × proof)

where we have already defined

<u>lettype</u> proof = thm list→ thm

(the type goal is defined below). We occasionally use tactic to mean tactic scheme, that is, a function from some parameters to a tactic, when this does not cause confusion.

There are many possible ways of defining the type goal in ML, (for example, some are discussed in [15,33]), and in certain respects, the standard definition in LCF is arbitrary. However, because later discussion depends on the particular choice, we explain the actual definition at this point. A goal consists of the formula to be proved, coupled with a list of current assumptions (induction hypotheses, case assumptions, lemmas and the like), and a third component which is extremely useful: a simplification set. A simplification set is (conceptually) a list of theorems intended to be used as left-to-right rewrite rules whenever possible in the course of a proof. In LCF, <u>simpset</u> (for simplification set) is another constant ML type. Simpsets are formed from lists of theorems; we occasionally identify simpsets with lists of theorems, in this presentation, where this does not cause confusion. A standard simpset of simple rewrites, called BASICSS, is provided in ML. It includes the rewrites justified by MINAP, by beta-conversion, by the reflexivity of equivalence, and other routine simplifications. Tools are also provided for the user to

form simpsets. We employ the notation

th + ss

for the simpset resulting from adding a simplification rule corresponding to the theorem th to the simpset ss. (The elements of a simpset are called simplification rules, or simprules.)

In general, the theorems suitable for being included in simplification sets are of the form $\vdash t \equiv t'$. Implications of the form $\vdash$w IMP $t \equiv t'$ are also acceptable, and are used as rewrites only when the antecedent, w, can be proved first by simplification. (Modus Ponens justifies the subsequent use of the simprule.) Rules formed from implications are called conditional simprules. In addition, theorems of the form $\vdash \forall$x1...xn.w are acceptable, when w is acceptable. Theorems of this form are specialised to arbitrary x1′,...,xn′ before being applied as rewrites. (For more detail on simplification in LCF, see [15], especially A8.)

A goal is therefore defined in ML by

<u>lettype</u> goal = form × simpset × form list

that is, it is composed of the formula to be proved, a relevant simplification set, and the current assumptions.

We write simple goals, with formula w, simpset ss, and assumption list A as (w, ss, A), and, in this presentation, more complex ones as

w
ss
A

in order to separate the components.

We say that a theorem, $A \vdash w$, achieves a goal, (w', ss, A') if w is w' (up to alpha-conversion) and if all of the hypotheses, A, of the theorem belong either to the assumption list A′, or are hypotheses of one of the theorems to which an element of the simpset, ss, corresponds.

A set of standard tactics is provided in LCF. Additional tactics are written in ML by the user. We introduce here another informal notation, for tactics, displaying the intended goal above a double line, and the subgoals returned, possibly with an indication of the proof function, below. Most of the time, the details of the proof function can be subordinated, as they are suggested by the specification of the subgoals.

For example, consider the standard inference rule GEN of type term ⟶ thm ⟶ thm:

GEN x

$$\frac{A \vdash w}{A \vdash \forall x.w}$$

where x is not free in A. We can then express a tactic

GENTAC

$\forall x.w$
ss
A

══════════

$w[x'/x]$
ss
A

$(\lambda[th].\ \text{GEN } x'\ th)$

where x' is not free in A, and $w[x'/x]$ means w with all free occurrences of x replaced by x'. GENTAC accepts a goal whose formula is quantified and returns a subgoal whose formula is specialised to an arbitrary variable x'. The proof part uses GEN. A theorem achieving the subgoal, when generalised to x', clearly achieves the goal, since the formula $\forall x'.w[x'/x]$ and $\forall x.w$, are the same up to renaming of variables. Thus GENTAC <u>inverts</u> the inference rule GEN. It implements the following heuristic for proving quantified formulae:

> To prove that w holds for all x, try proving for arbitrary x', that w with x replaced by x' holds.

Two other useful standard tactics are CASESTAC and INDUCTAC. The inference rules which they invert are, naturally, CASES and INDUCT:

CASES: term $\longrightarrow$ (thm $\times$ thm $\times$ thm) $\longrightarrow$ thm

INDUCT: (term $\times$ term) list $\longrightarrow$ form $\longrightarrow$ (thm $\times$ thm) $\longrightarrow$ thm

where

CASES (t:tr)

$$\frac{\begin{array}{l}(t \equiv TT).\ A1 \vdash w \\ (t \equiv FF).\ A2 \vdash w \\ (t \equiv \perp).\ A3 \vdash w\end{array}}{A1 \cup A2 \cup A3 \vdash w}$$

(where an assumption list A `matches` w.A1 if A contains w, up to alpha-conversion, and A's remaining elements match A1). CASES takes three theorems, representing a theorem with the respective

assumptions that some term t is true, false and undefined, and proves the theorem without case assumptions.

We let [funi,fi] denote the list [fun1,f1;...;funn,fn], and w[xi/fi] denote w[x1/f1]...[xn/fn].

INDUCT [funi,fi]

$$\frac{A1 \vdash w[\bot/fi] \qquad w.A2 \vdash w[(funi\ fi)/fi]}{A1 \cup A2 \vdash w[(FIX\ funi)/fi]}$$

where the fi are not free in A2. This expresses the rule of computation induction originally formulated by Park [39]:

$$(w[\bot/fi]\ \&\ \forall fi.\ w \supset w[(funi\ fi)/fi]\ ) \supset w[(FIX\ funi)/fi]$$

INDUCT is the standard rule of induction in LCF; any other desired induction rules must be derived from it. New induction rules are mentioned in Chapters 3 and 4, and discussed in the Conclusions and Appendix.

CASESTAC and INDUCTAC are tactic schemes having types

CASESTAC:term $\rightarrow$ tactic

INDUCTAC:thm list $\rightarrow$ tactic

and are depicted as:

CASESTAC (t:tr)

(w, ss, A)

w
$(t \equiv TT) + ss1$
$(t \equiv TT).A$

w
$(t \equiv FF) + ss2$
$(t \equiv FF).A$

w
$(t \equiv \bot) + ss3$
$(t \equiv \bot).A$

(λ[th1;th2;th3]. CASES t (th1,th2,th3))

where the simpsets ss1, ss2 and ss3 are all ss, with the respective assumptions $\vdash t \equiv TT$, $\vdash t \equiv FF$, and $\vdash t \equiv \bot$ added as simprules.

Another standard tactic, CONDCASESTAC, searches through the formula w to find the first term of boolean type which is the boolean-valued part of a conditional, and performs case analysis on that term. CONDCASESTAC fails if it finds no appropriate term.

INDUCTAC is depicted as follows, where $[\vdash ti \equiv FIX\ ui]$ denotes the theorem list $[\vdash t1 \equiv FIX\ u1;\ldots;\vdash tn \equiv FIX\ un]$:

INDUCTAC $[\vdash ti \equiv FIX\ ui]$

(w, ss, A)

w[⊥/ti]	w[(ui xi)/ti]
ss	ss
A	(w[xi/ti]).A

(λ[basis;step]. INDUCT (ui,ti) w (basis,step))

where the xi are not free in w or A. INDUCTAC, given a list of theorems defining the ti as least fixed points of the functionals ui, returns two subgoals: a basis, with ⊥ substituted for the ti, and a step, with (ui xi) -- xi rather than ti because the inductive step holds for <u>all</u> xi -- with the hypotheses added to the list of assumptions. The proof part expects two theorems, achieving the basis and step, respectively, calls INDUCT to prove w[(FIX ui)/ti], and substitutes according to the definitions of the ti.

If a tactic T when applied to a goal g produces an empty list of subgoals, we say that T <u>solves</u> g. This is not to say that g has been achieved, however, since the proof function might be incorrect. If T on g gives subgoals g1,...,gn and proof p, and if it is the

case that for any theorems th1,...,thn which achieve goals g1,...,gn respectively, p applied to the theorem list [th1;...;thn] achieves g, then we say that T is valid. If, in addition, the goals g1,...,gn are achievable, we say that T is strongly valid. Ideally, one would always use strongly valid tactics, but this is not always possible; the tactic suggested by the heuristic on p. 28, for example, is not strongly valid, but the tactic is nonetheless useful.

In any case, it is important to note that, valid or otherwise, application of a proof function to a theorem list cannot return a non-theorem. At worst, the application fails, or an unexpected theorem results.

To reflect the heurstic, on p. 28, we write a tactic (which we call MINCOMBTAC) depicted as

MINCOMBTAC

$$\frac{(t\ x \equiv \bot,\ ss,\ A)}{(t \equiv \bot,\ ss,\ A) \qquad (\lambda[th].\ \mathrm{TRANS}(\mathrm{APTHM}\ x\ th,\ \mathrm{MINAP}\ x))}$$

A procedure to implement this tactic is easily written in ML. To give the flavour of the process of implementing tactics in ML, we show the procedure below:

```
let (MINCOMBTAC:tactic) (w, ss, A)  =
     let r = rhs w
     in if isbottom r
        then let (t,x) = destcomb(lhs w)
             in (mkequiv(t,r), ss, A), (λ[th].TRANS(APTHM x th,
                                                    MINAP x))
        else fail
```

The procedure examines and takes apart w, and if the right hand side

of the formula is $\perp$, it gives meta-names to w's parts and constructs the appropriate subgoal list and proof. If not, the tactic fails. (For a further discussion of the failure trapping mechanism in ML, see [15], especially 2.1.)

As we observed earlier, the fact that $\perp \equiv \perp$ follows from reflexivity, which is expressed as an inference rule (axiom scheme) in LCF by the rule REFL:term $\longrightarrow$ thm

REFL t

$$\frac{}{\vdash \quad t \equiv t}$$

The tactic we require to complete the proof that $\perp x1...xn \equiv \perp$ could be called BOTREFLTAC:

BOTREFLTAC

$$\frac{(\perp \equiv \perp,\ ss,\ A)}{[\ ],\quad (\lambda[\ ].\ \mathrm{REFL}\ \perp)}$$

The tactic, in trying to prove that $\perp \equiv \perp$, returns an empty list of subgoals (that is, it recognises that the goal can be achieved immediately) and a proof which expects an empty list of theorems and returns the appropriate theorem as result. To implement BOTREFLTAC in ML we would write

```
let (BOTREFLTAC:tactic) (w, ss, A) =
     let (t1,t2) = destequiv w
     in if isbottom t1
        then if isbottom t2
             then ([ ], (λ[ ].REFL t1))
             else fail
        else fail
```

Finally, we indicate how basic and user defined tactics can be combined to form more sophisticated tactics. A control structure for the language of tactics is provided by tacticals. By analogy with functionals, tacticals are functions which take tactics as arguments and/or return tactics as results. The main tacticals provided in LCF are THEN, THENL, ORELSE and REPEAT, with types

THEN: (tactic × tactic) ⟶ tactic

THENL: (tactic × tactic list) ⟶ tactic

ORELSE:(tactic × tactic) ⟶ tactic

REPEAT:tactic ⟶ tactic

As for inference rules, we use tactical to mean tactical scheme, as different tacticals have different ML types. For readability, the first three tacticals listed above are infixed.

As the names suggest, T1 THEN T2 is a tactic which, given a goal, applies T1 to the goal to obtain subgoals, applies T2 to the subgoals to obtain further subgoals, and returns those, along with the correctly composed proof function. T THENL [T1;...;Tn] applies each tactic in the list (respectively) to each subgoal in the list of subgoals produced by applying T to a goal. T1 ORELSE T2 applies T1 to a goal, and if that fails, applies T2. REPEAT T applies T to a goal and to successive subgoals until a failure occurs (if it ever does). (Obviously, the ML failure trapping mechanism is basic to the use of tacticals.)

We distinguish tactics implemented as ML procedures which do not call other tactics from tactics built by the use of tacticals, by calling the two sorts derived and composite tactics,

respectively, throughout this presentation.

To return to the goal with formula part $\perp$ x1...xn $\equiv$ $\perp$, we are now in a position to solve the goal with a composite tactic. We simply apply the tactic

```
(REPEAT MINCOMBTAC) THEN BOTREFLTAC
```

to the goal with the correct formula, an empty simpset, and an empty list of assumptions. This solves the goal, since each application of MINCOMBTAC to the subgoal with formula $\perp$ x1...xi $\equiv$ $\perp$ 'removes' xi to give the subgoal $\perp$ x1...x(i-1) $\equiv$ $\perp$, until the subgoal with formula $\perp \equiv \perp$ is produced; MINCOMBTAC then fails, and the goal is solved by BOTREFLTAC. The proof returned by the application of the whole (composite) tactic to the goal, when applied to the empty list of theorems, returns the theorem $\vdash$ $\perp$x1...xn $\equiv$ $\perp$, which is what we set out to prove.

This example, however, is somewhat contrived, because it operates at a simpler level than that at which one normally works in LCF. Reasoning at this level is generally handled by a standard (rather special) tactic called SIMPTAC. Given a goal with formula part w and simpset ss, SIMPTAC returns a goal with a formula part which is the result of applying the rewrite rules in ss as many times as possible to w. It also returns a proof which justifies the simplifications made. SIMPTAC returns the empty list of subgoals if some subgoal arising in the course of simplification is a (recognised) tautology. In that case, the proof function returned, when applied to the empty list of theorems, returns a theorem achieving the original goal. For example, goals with formulae of

the form $\perp x \equiv \perp$ or $t \equiv t$ are solved immediately by simplification, provided that the standard set of basic simplifications (BASICSS) is included in the simpset of the goals.

SIMPTAC, used with the basic simprules, relieves the user of a great deal of the tedium of generating proofs; it accomplishes much routine work automatically. By using other theorems as simplification rules, still more proof can be relegated to simplification. This is illustrated at numerous points in the following three chapters.

At any rate, we can now see that the goal with the formula part $\perp$ x1...xn $\equiv \perp$ could actually have been solved by a single application of SIMPTAC, assuming that the basic set of simplifications were included in the original goal.

It is important to observe that although the eventual outcome of applying the compound tactic above (or SIMPTAC) to the appropriate goal is simply a theorem, and the sequence of inference rules invoked and intermediate theorems proved is nowhere stored, a complete proof has still been evaluated. That is, each step of the proof has been performed, and the application of the tactic to the goal has generated the proof. Modifications could be made to the type goal in LCF to ensure that the sequence of proof steps were preserved, if that were desired. The type-checking facility of ML guarantees that only rules of inference can return objects of type thm, however, so it is not necessary to store sequences of primitive proof steps.

The style of proof illustrated in this section (in contrast to the forward proof described on p. 27) is called tactical or goal oriented proof. One of the principles of LCF is that the generation of subgoals from goals by the application of tactics reflecting strategies is a natural and convenient style of proof, a style which corresponds to the way in which proofs are planned and abstracted by humans. Tactical proof allows varying degrees of automation; tactics which are inverses of basic inference rules generate subgoals at a basic level, requiring the user to be aware of the detailed course of the proof, while sophisticated tactics may accomplish large proof steps, or whole proofs, sparing the user contact with the details. The end product of a tactical proof is what might be called a 'proof story' or a 'high level proof', rather than a proof in the conventional sense of a sequence of theorems, each following from earlier ones by applications of primitive inference rules. High level proofs are both more intelligible and more revealing (of the structure of the proof effort) than long sequences of this kind; the tactics required to perform a proof provide a better basis for making generalisations and proving other, similar theorems.

The problems considered herein are all experiments in the use of tactical proof. We consider some proofs related to recursion removal and to compilation of simple languages, and study the tactics which generate them, with an eye for useful, general tactics. We then try to assess the difficulty of performing more realistic verifications by this methodology.

Chapter 2: Proofs of Recursion Removal Schemata

The first group of proofs which we discuss are proofs of the equivalence of several recursive function schemata to iterative schemata. Three case studies in LCF are examined. In each case, we present the transformation and give the informal proof in sufficient detail to motivate the tactics which generate the machine proofs. We discuss the formalisations of the problems in PPLAMBDA, and the implementation of the proof strategies in ML. Our aim is to isolate useful and general tactics for these and related proofs, rather than to discover program transformations or to prove their correctness automatically; we concentrate on the more narrow goal of generating proofs once the theorems to be proved have been found and the methods of proof settled. We conclude by outlining a hypothetical general tactic, based on the examples, for proving equivalences of recursive and iterative schemata, illustrating the way in which general strategies can be developed, expressed and applied in LCF.

The machine proofs exercise LCF in its capacity for expressing _general_ properties of recursive functions. We use PPLAMBDA (extended with new logical types) and its implicit semantics to define the functions, rather than give an explicit syntax and semantics for a language of recursive definitions. (When we consider compilation, in Chapters 3 and 4, we _do_ define new languages and give their semantics.) At present, we verify only particular transformations which a compiler would treat in a uniform way. The methods of proof, however, appear to be quite general.

The first two problems are drawn from Manna and Waldinger [22]. The first is very simple, and is considered in some detail chiefly as a way of further introducing the formalisation of problems and the generation of proofs in LCF. We have devised the third problem to show that similar tactics can be used to solve a different goal. Further details of the actual machine proofs are found in the Appendix.

The Accumulator Problem

We consider a recursive function F, defined as follows:

$$F\ x \;=\; P\ x \Rightarrow f\ x \;\mid\; h(x,F(g\ x))$$

where h is taken to be an associative, binary operation with left identity e, strict in its second argument. One can transform F to an iterative [1] function F1 by introducing an accumulator z as an argument [2] :

$$F1\ x\ z \;=\; P\ x \Rightarrow h(z,f\ x) \mid F1(g\ x)(h(z,x))$$

and proving that for all x,

$$F1\ x\ e \;=\; F\ x$$

We prove this by showing something more general, namely

Theorem 2.1

$$\forall x\ z.\ F1\ x\ z \;=\; h(z,F\ x)$$

We formulate the problem by defining F and F1 as the least fixed

points of functionals FUNF and FUNF1 respectively, where

$$\text{FUNF} = \lambda F'\ x.\ P\ x \Rightarrow f\ x \mid h(x, F'(g\ x))$$

$$\text{FUNF1} = \lambda F1'\ x\ z.\ P\ x \Rightarrow h(z, f\ x) \mid F1'(g\ x)(h(z,x))$$

Then F = FIX FUNF and F1 = FIX FUNF1.

The proof of Theorem 2.1 is by parallel computational induction on F and F1. We assume that for all F1′ and F′,

$$\forall x\ z.\ F1'\ x\ z = h(z, F'\ x)$$

and show

$$\forall x\ z.\ P\ x \Rightarrow h(z, f\ x) \mid F1'(g\ x)(h(z,x)) = h(z,\ P\ x \Rightarrow f\ x \mid h(x, F'(g\ x)))$$

proving the step for arbitrary x and z, and arguing by cases on whether P holds of x. (The basis of the induction,

$$\bot\ x\ z = h(z, \bot\ x)$$

is easy to show.) If P x does hold, or if P x is undefinied, the argument is easy. If it does not, we must show that

$$F1'(g\ x)(h(z,x)) = h(z, h(x, F'(g\ x)))$$

This is accomplished by applying associativity, and then using the induction assumption with z instantiated to h(z,x) and x to (g x).

The proof is typical of many proofs about recursively defined functions. The functions are defined as the least fixed points of functionals, so we use computation induction. The defining functionals are conditionals, evaluating some boolean-valued term

and branching to recursive calls. The general form of such proofs is summarised by the following informal strategy:

> Do induction on the recursively defined functions, then prove for arbitrary values of the variables. The basis is easy. Divide into cases according to whether the condition is true or false. Simplify, and use the induction hypothesis where appropriate.

Our aim here is to represent the Accumulator problem in PPLAMBDA and to reflect the plan for the proof in a tactic which generates the formal proof.

We first make sense of the function definitions by assigning types to the variables; * is a type variable.

$F{:}* \longrightarrow *$

$F1{:}* \longrightarrow * \longrightarrow *$

$P{:}* \longrightarrow tr$

$h{:}(* \longrightarrow *) \longrightarrow *$

$g{:}* \longrightarrow *$

$f{:}* \longrightarrow *$

$e{:}*$

We then invoke the PPLAMBDA inference rule (axiom scheme) ASSUME, of type form $\longrightarrow$ thm,

ASSUME w

$$\frac{}{w \vdash w}$$

to introduce the assumptions governing h and e [3] :

$\vdash$ a b c. h(a,h(b,c)) $\equiv$ h(h(a,b),c)

$\vdash$ a. h(a,$\perp$) $\equiv$ $\perp$

$\vdash$ a. h(e, a) $\equiv$ a

(each of which has one hypothesis, namely, the formula assumed), and the definitions of F and F1, to which we give the meta-names thF and thF1:

thF

$\vdash$F $\equiv$ FIX(λF′ x. P x $\Rightarrow$ f x | h(x,F′(g x)))

thF1

$\vdash$F1 $\equiv$ FIX(λF1′ x z. P x $\Rightarrow$ h(z,f x) | F1′(g x)(h(z,x)))

Our aim is to prove the following theorem in LCF, corresponding to Theorem 2.1

thA

$\forall$x z. F1 x z $\equiv$ h(z,F x)

(A for Accumulator) using the five assumptions as rewrite rules. We therefore form a simpset (called SSA), adding to BASICSS the five assumptions, and join SSA with the formula to be proved, and an empty list of assumptions, to form a goal (goalA):

$\forall$x z. F1 x z $\equiv$ h(z,F x)
SSA

We mirror the informal proof plan (up to the use of the induction hypothesis) as a composite tactic:

```
INDUCTAC [thF1;thF] THEN SIMPTAC THEN REPEAT GENTAC
THEN CONDCASESTAC THEN SIMPTAC
```

In the interests of succinctness, we adopt a convention of writing composite tactics in columns, concealing occurrences of the tactical THEN, using T+ to denote T THEN SIMPTAC, and T* to denote REPEAT T. The tactic so far is therefore

```
(INDUCTAC [thFl;thF])+
GENTAC*
CONDCASESTAC+
```

This informal notation could be made rigorous by introducing two new tacticals:

```
SEQ:tactic list⟶ tactic

THENS:tactic⟶ tactic
```

defined in ML by writing

```
let SEQ tacl = if null tacl
               then IDTAC
               else (hd tacl) THEN (SEQ(tl TACL))
```

(where IDTAC is a standard tactic such that for all g:goal, IDTAC g = ([g],hd)) and

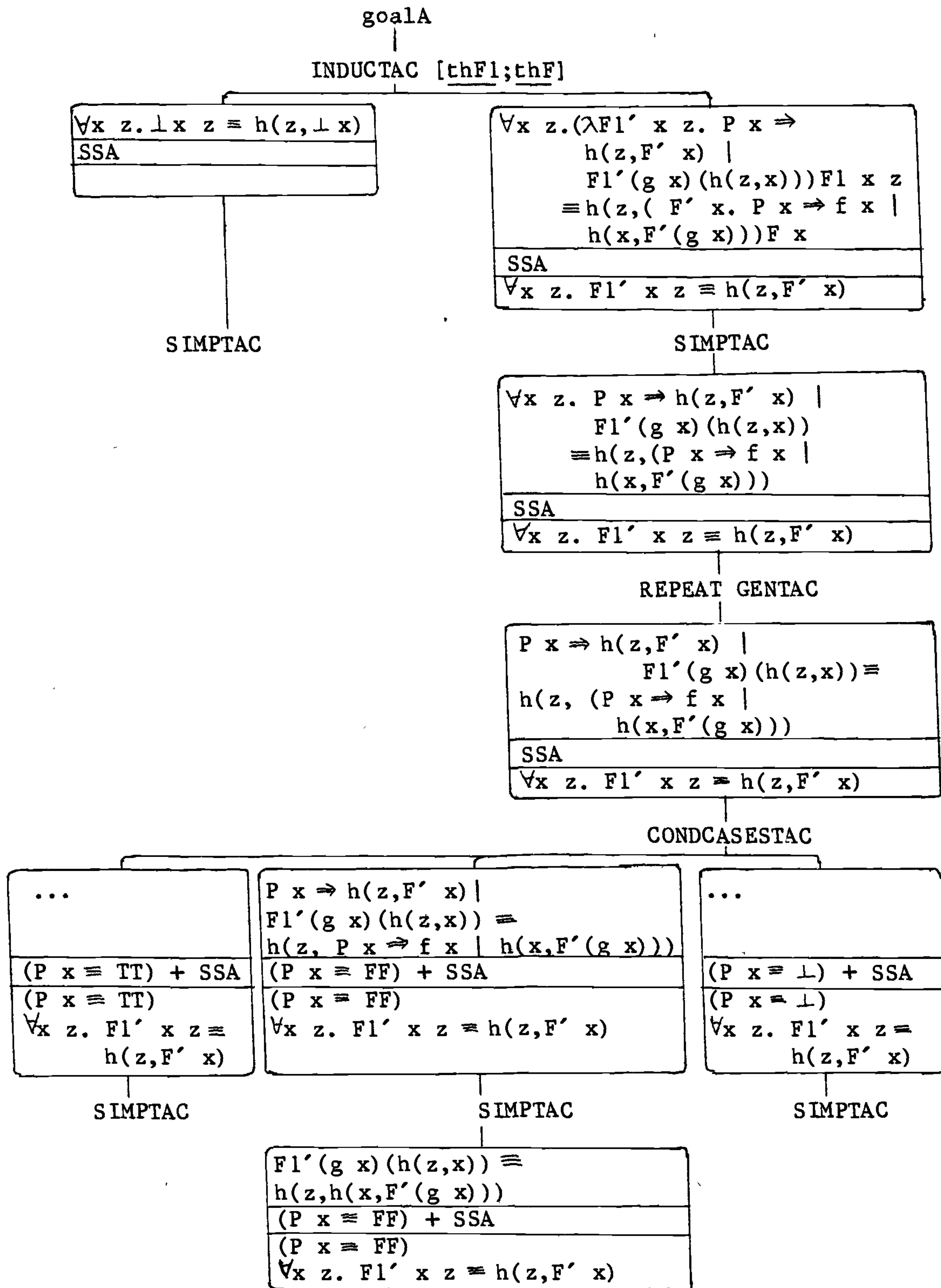
```
let THENS T = T THEN SIMPTAC
```

so that the tactic thus far would then be written

```
SEQ [THENS (INDUCTAC [thFl;thF]);
     REPEAT GENTAC;
     THENS CONDCASESTAC]
```

but we will continue to use the more informal notation.

The tactic generates a tree of successive subgoals:



where the three subgoals produced by CONDCASESTAC all have the same

formula part.

Application of the composite tactic developed thus far to goalA yields exactly one subgoal (the basis subgoal is solved by SIMPTAC using the simplifications in BASICSS for the minimality of $\perp$). The other two cases of P x are solved by SIMPTAC, using simplifications for conditionals, and the strictness of h. The remaining subgoal consists of the formula shown, a simpset supplemented by the assumption that P x $\equiv$ FF, and a list of assumptions including the induction hypothesis and that case assumption.

The tactic sought to complete the proof must use the induction assumption; it must first recognise that the formula to be proved is an instance of (matches the unquantified version of, up to renaming of variables) one of the formulae in the assumption list. We define a tactic called USEASSUMPTAC which accepts a goal, searches through the assumption list for a match as specified above, and, if a match can be found, returns an empty list of subgoals. The proof function assumes the quantified formula and specialises it according to the match. USEASSUMPTAC is programmed in ML in terms of standard procedures to match terms and to test the equivalence of formulae up to alpha-conversion. It is denoted by the diagram below.

USEASSUMPTAC

(w, ss, [...;∀x1...xn.w′;...])

[ ], (λ[ ]. SPECL [x1;...;xn] (ASSUME (∀x1...xn.w′)))

where w′ = w[xi′/xi] and the derived rule of inference SPECL:term list ⟶ thm ⟶ thm specialises a quantified theorem:

SPECL [t1;...;tn]

$$\frac{A \vdash \forall x1\ldots xn.w}{A \vdash w[ti/xi]}$$

(See [15], A5, for a description of the standard rule SPEC, from which SPECL is derived.)

The theorem produced by applying the proof returned by USEASSUMPTAC to an empty list of theorems is

$$\forall x1\ldots xn.w' \vdash w'$$

which obviously achieves the goal.

USEASSUMPTAC completes the tactical proof. The whole tactic which solves goalA, TACA, say, is therefore

TACA

```
(INDUCTAC [thF1;thF])+
GENTAC*
CONDCASESTAC+
USEASSUMPTAC
```

When TACA is applied to goalA we obtain an empty list of subgoals and a proof which when applied to that list applies, in turn, the proof parts of USEASSUMPTAC, SIMPTAC, CONDCASESTAC, and so on, and finally, of INDUCTAC [thF1;thF], to produce thA, corresponding to Theorem 2.1:

$$\vdash \forall x\ z.\ F1\ x\ z \equiv h(z,F\ x)$$

with five hypotheses, corresponding to the five original assumptions.

The proof tree on p. 47 is completed by adjoining to the remaining subgoal the following tree:

```
        |
  USEASSUMPTAC
```

By adding thA to the simpset of the goal

$\forall x.$ F1 x e ≡ F x
BASICSS + SSA

and applying SIMPTAC, we achieve as a corollary the theorem we actually set out to prove ($\forall x.$ F1 x e = F x).

Although not especially interesting in itself, this example illustrates the way in which an informal strategy (which is, in fact, quite general) is mirrored in a tactic and implemented as an ML procedure. The example also suggests the way in which formal proofs (once the main insight is had) can be generated in LCF with a minimum of guidance on behalf of the user.

We go on to consider two more schema problems and proofs, both of which require rather more sophisticated tactics.

The List Stack Problem

The Problem

We begin, this time, with a recursive function F which has parallel recursive calls:

$$F\ x = P\ x \Rightarrow f\ x \mid h(F(g1\ x),F(g2\ x))$$

where h is a binary, associative function with left identity e,and is strict in both arguments. We introduce a 'stack' (list) s, and an accumulator z, to write an iterative function F1:

$$
\begin{aligned}
F1\ x\ z\ s = NULL\ s \Rightarrow z\ | \\
P\ x \Rightarrow F1(HD\ s)(h(z,f\ x))(TL\ s)\ | \\
F1(g1\ x)\ z(CONS(g2\ x)\ s)
\end{aligned}
$$

where NULL, HD, TL and CONS are the usual list operators, (and NIL is the empty list).[4,5]

As before, we define F and F1 to be the least fixed points of functionals FUNF and FUNF1 respectively:

$$FUNF = \lambda F'\ x.\ P\ x \Rightarrow f\ x\ |\ h(F'(g1\ x),F'(g2\ x))$$

$$
\begin{aligned}
FUNF1 = \lambda F1'\ x\ z\ s.\ NULL\ s \Rightarrow z\ | \\
P\ x\quad F1'(HD\ s)(h(z,f\ x))(TL\ s)\ | \\
F1'(g1\ x)\ z(CONS(g2\ x)\ s)
\end{aligned}
$$

We prove[6]:

<u>Theorem</u> <u>2.2</u>

$$\forall x.\ F1\ x\ e\ [NIL] = F\ x$$

Again, we are required to prove something more general. To motivate the theorem we prove, consider the computation of F1 x z s, for some x, z and s, where s = [s1;...;sn]. We would like to compute (F x) and to combine the result, via h, with the accumulated result z, and then combine <u>that</u> with (F s1), and so on. That is,

$$F1\ x\ z\ [s1;...;sn] = h(...h(h(z,F\ x),F\ s1)\ ...,F\ sn)$$

The expression on the right hand side is generated by a function Exp (for Expand) with functional arguments F and g:

$$\text{Exp F h x z s} = \text{NULL s} \Rightarrow \text{z} \mid \text{Exp F h (HD s)(h(z,F x))(TL s)}$$

As Exp is recursive, we define it as the least fixed point of a functional FUNExp (whose definition is obvious).

We prove that Fl x z s = Exp F h x z s, for all x, z and s, and Theorem 2.2 follows easily.

To prove that Fl x z s = Exp F h x z s, we introduce another function which is similar to Exp but does not have functional arguments:

$$\text{G x z s} = \text{NULL s} \Rightarrow \text{z} \mid \text{G(HD s)(h(z, F x))(TL s)}$$

We let G = FIX FUNG, where

$$\text{FUNG} = \lambda G' \text{ x z s. NULL s} \Rightarrow \text{z} \mid G'\text{(HD s)(h(z,F x))(TL s)}$$

and we prove

Theorem 2.3
$G = \text{Exp F h}$

Theorem 2.4
$\text{Fl} \sqsubseteq G$

Theorem 2.5
$G \sqsubseteq \text{Fl}$

We first summarise the proofs [7] :

Plan for Proof of Theorem 2.3
By parallel induction on G and Exp, and case analysis on whether s is empty.

Plan for Proof of Theorem 2.4

By induction on Fl, case analysis on whether s is empty, and unfolding the definitions of Exp and F.

Plan for Proof of Theorem 2.5

By showing that FUNG Fl $\sqsubseteq$ Fl. The proof is by unfolding the definition of Fl, then induction on F. The basis case requires Lemma 2.6, below. Both the basis and step are by case analysis on whether s is empty, and the step is by further cases analysis on whether P holds of x, and by successive uses of the induction hypothesis.

Lemma 2.6

$\forall$x. Fl x $\perp$ s = $\perp$

Plan for Proof of Lemma 2.6

By induction on Fl, case analysis on whether s is empty, and further cases on whether P holds of x.

The rule to which we appeal in the plan for proving Theorem 2.5 is proved by induction (which we do later). We examine the proof of Theorem 2.5 in some detail in order to understand the tactics required to generate the proofs mechanically. The proof is representative of the others.

Proof of Theorem 2.5

It is sufficient to show that FUNG Fl $\sqsubseteq$ Fl (see below), i.e.

$\forall$x z s. (NULL s $\Rightarrow$ z | Fl(HD s)(h(z,F x))(TL s)) $\sqsubseteq$ Fl x z s

Induction is done on F.

Basis

(NULL s $\Rightarrow$ z | Fl(HD s)(h(z,$\perp$x))(TL s) $\sqsubseteq$ Fl x z s

Step

Assume

$\forall$x z s. NULL s $\Rightarrow$ z | Fl(HD s)(h(z,F′ x))(TL s) $\sqsubseteq$ Fl x z s

Show

```
NULL s ⇒ z | F1(HD s)(h(z,(P x ⇒ f x |
                         h(F'(g1 x),F'(g2 x)))))(TL s) ⊑
F1 x z s
```

We unfold the occurrence of F1 on the right hand sides (i.e. the second occurrence) according to the definition; for both the basis and step, the right hand side is

```
NULL s ⇒ z | P x ⇒ F1(HD s)(h(z,f x))(TL s) |
                   F1(g1 x)z(CONS(g2 x)s)
```

We then do case analysis on whether s is empty. Using Lemma 2.6, the basis is easy. If s is empty, the step is also immediate. (Here and elsewhere, the undefined case is obvious, and we omit it.)

Case NULL s = FF

We do further case analysis on P x. If P holds of x, the step is obvious.

Case P x = FF

We must show

```
F1(HD s)(h(z,h(F'(g1 x),F'(g2 x))))(TL s) ⊑
F1(g1 x)z(CONS(g2 x)s)
```

By hypothesis, with (g1 x) for x, z for z, and (CONS(g2 x)s) for s, we know that

```
NULL(CONS(g2 x)s) ⇒ z | F1(HD(CONS(g2 x)s))
                          (h(z,F'(g1 x)))
                          (TL(CONS(g2 x)s)) ⊑
F1(g1 x)z(CONS(g2 x)s)
```

that is,

```
F1(g2 x)(h(z,F'(g1 x))) ⊑ F1(g1 x) z(CONS(g2 x)s)
```

Also, by hypothesis, with (g2 x) for x, h(z,F'(g1 x)) for z, and s for s, we know

```
NULL s ⇒ h(z,F'(g1 x)) | F1(HD s)
                          (h(h(z,F'(g1 x)),F'(g2 x))) (TL s) ⊑
F1(g2 x)(h(z,F'(g1 x)))s
```

which, since s is assumed to be non-empty, implies that

```
F1(HD s)(h(h(z,F'(g1 x)),F'(g2 x)))(TL s) ⊑
F1(g2 x)(h(z,F'(g1 x))) s
```

and the desired result follows by the associativity of h, and by transitivity. To complete the proof, we verify the rule

that we used:

$G = FIX\ FUNG\ \&\ FUNG\ F1 \sqsubseteq F1 \supset G \sqsubseteq F1$

We assume the antecedent, and do induction on G. The basis is easy. We assume $G' \sqsubseteq F1$. Applying FUNG to both sides, we have

$FUNG\ G' \sqsubseteq FUNG\ F1$

and using transitivity with the assumption:

$FUNG\ G' \sqsubseteq F1$

Thus FIX FUNG $\sqsubseteq$ F1, that is, G $\sqsubseteq$ F1. Q.E.D.

The Formalisation

To perform the proof of Theorem 2.2 in LCF, we work in a theory of lists (of arbitrary type) in which a unary type operator (* list) is available, and various new constants, with the usual meanings, have been introduced:

HD: * list→ *

TL: * list→ * list

CONS:*→* list→ * list

NIL: * list

NULL:* list→ tr

LIST:*→* list

For purposes of presentation it does not matter how lists are axiomatised, as long as the facts below are axioms or proved theorems. (For more details on possible list theories, see the Appendix, or [15], especially A1.) We take CONS to be non-strict.

$\vdash \forall x\ s.\ NULL(CONS\ x\ s) \equiv FF$

$\vdash NULL\ NIL \equiv TT$

$\vdash \forall x\ s.\ HD(CONS\ x\ s) \equiv x$

$\vdash \forall x\ s.\ TL(CONS\ x\ s) \equiv s$

$\vdash \forall x.\ LIST\ x \equiv CONS\ x\ NIL$

$\vdash \forall s.\ HD(LIST\ s) \equiv s$

$\vdash \forall s.\ TL(LIST\ s) \equiv NIL$

We introduce four assumptions defining the functions F, F1, Exp and G:

thF

$$F \equiv FIX(\lambda F'\ x.\ P\ x \Rightarrow f\ x \mid h(F'(g1\ x), F'(g2\ x)))$$

and similarly for thF1, thExp and thG, and four assumptions about h and e :

$\vdash \forall x.\ h(e,x) \equiv x$

$\vdash \forall x.\ h(x,\perp) \equiv \perp$

$\vdash \forall x.\ h(\perp,x) \equiv \perp$

$\vdash \forall a\ b\ c.\ h(h(a,b),c) \equiv h(a,h(b,c))$

All of these theorems are put into a simpset (along with the basic simplification rules), which we call SSL (for simpset for List Stack).

Our main goal is to prove the theorem corresponding to Theorem 2.2, , which we call thL0. We use the new constants LIST and NIL to construct the list containing exactly one element.

thL0

$$\forall x.\ Fl\ x\ e\ (LIST\ NIL) \equiv F\ x$$

To prove thL0, we specify a goal, goalL0:

goalL0

∀x. Fl x e (LIST NIL) ≡ F x
thL4 + thL5 + SSL

where thL4 is the theorem which achieves goalL4:

goalL4

Fl ⊑ Exp F h
LemmaL2 + SSL

and thL5 is the theorem (needed for proving our main goal from thL4) which achieves goalL5:

goalL5

Exp F h x e (LIST NIL) ≡ F x
SSL

and LemmaL2 is the theorem (corresponding to Lemma 2.6) which achieves goallemL2:

goallemL2

∀x. Fl x ⊥ s ≡ ⊥
SSL

thL5 is easy to prove; we concentrate, in the following on proving goalL4. To achieve goalL4, we must prove thL1, thL2 and thL3, which are, respectively, the theorems which achieve goalL1, goalL2 and goalL3 (corresponding to Theorem 2.3, Theorem 2.4 and Theorem 2.5):

goalL1

G ≡ Exp F h
SSL

goalL2

F1 ⊆ G
SSL

goalL3

G ⊆ F1
LemmaL2 + SSL

The List Stack Proof in LCF

With the goals thus set out, we are now able to discuss the generation of the proof in LCF. As mentioned earlier, the main aim of this work is not to do automatic theorem proving; we are not interested in writing tactics, say, to generalise the main goal (goalL0) to goalL5, or to inspect goalL3 and decide that it is sufficient and convenient to prove a goal with the formula FUNG F1 ⊆ F1, instead. Our aim is to design tactics which mirror the informal proof once these insights have been found. We begin, though, by applying to goalL4 a tactic which 'discovers' goalL1, goalL2 and goalL3, and combines them for us. The tactic is motivated by observing the relation between G and Exp. The equivalence of G to Exp F h is an instance of the 'By-law', so called because of its combinatory form:

B Y = B Y S

where $Y = FIX$, $B = \lambda x\ y\ z.\ \ x(y(z))$, and $S = \lambda x\ y\ z.\ \ x\ z\ \ (y\ \ z)$. For our purposes, the By-law can be stated as:

$$\lambda F1...Fn.\ FIX(\Phi F1...Fn) = \\ FIX(\lambda E\ F1'...Fn'.\ \Phi F1'...Fn'\ (E\ F1'...Fn'))$$

Intuitively [8], the fixed point can be taken inside our outside of the abstraction. In the present case we take Φ to be

$$\lambda F'\ h'\ G'\ x\ z\ s.\ NULL\ s \Rightarrow z \mid G'(HD\ s)(h'(z,F'\ x))(TL\ s)$$

and n to be 2, F1 to be F, and F2 to be h. Then the By-law tells us that

$$FIX(\lambda G'\ x\ z\ s.\ NULL\ s \Rightarrow z \mid G'(HD\ s)(h(z,F\ x))(TL\ s)) \equiv$$
$$(FIX(\lambda Exp'\ F'\ h'\ x\ z\ s.\ NULL\ s \Rightarrow z \mid \\ Exp'\ F'\ h'\ (HD\ s)(h'(z,F'\ x))(TL\ s))\ F\ h$$

that is, by definition, $G \equiv Exp\ F\ h$.

We write an ML procedure called BYLAW [9] to express the By-law as a rule of inference scheme:

BYLAW [F1;...;Fn]

$$\frac{\vdash G \equiv FIX(\Phi\ F1...Fn)}{\vdash G \equiv FIX(\lambda E'\ F1'...Fn'.\Phi F1'...Fn'(E\ F1'...Fn'))\ F1...Fn}$$

BYLAW takes a list of the functional arguments to be made explicit in the function definition, coins a new variable E′ of appropriate type, and returns the new function definition. The proof is by induction on x and y in the formula

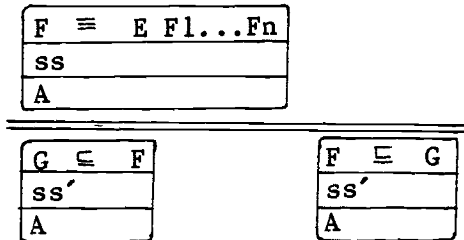
$$x \equiv y\ F1...Fn$$

with the functionals Φ F1...Fn and λ E F1'...Fn'. F1'..Fn' (E F1'...Fn') for x and y, respectively.[10]

We arrange for BYLAW to do the induction for us. (Details of the procedure are given in the Appendix.)

The tactic wanted for goalL4 (one which inverts BYLAW) is denoted by

BYTAC ($\vdash$ E $\equiv$ FIX(λE' F1'...Fn'.ΦF1'...Fn'(E F1'...Fn')))

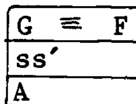


BYTAC invents the function G without functional arguments, and produces the two subgoals shown, where ss' is ss with the definition of G added, that is, with

G $\equiv$ FIX(Φ F1...Fn)

added. The proof part of BYTAC expects two theorems achieving the two subgoals, combines them to prove that G $\equiv$ F, then proves that G $\equiv$ E F1...Fn by using BYLAW and the definition of E. It conclude s that F $\equiv$ E F1...Fn, the theorem desired.

We note that a more basic BYTAC would return one subgoal, namely



and would be composed, to have the effect of the BYTAC described,

via THEN with a simple tactic called SYNTHTAC, inverting the standard inference rule SYNTH.

SYNTHTAC

$F \equiv G$
ss
A

$$\overline{\overline{\qquad\qquad\qquad\qquad\qquad}}$$

$F \subseteq G$
ss
A

$G \subseteq F$
ss
A

where

SYNTH

$$\frac{A1 \vdash t \subseteq u \qquad\qquad A2 \vdash u \subseteq t}{A1 \cup A2 \vdash t \equiv u}$$

In our proof, BYTAC does some of the top-level work for us by 'inventing' the function G, inventing and achieving goalL1 internally, and inventing as subgoals goalL2 and goalL3.

We examine the proof of goalL3. We concentrate, in doing this, on finding useful and general tactics for generating the proof, which reflect the reasoning done in the informal proof. Our methodology is to design tactics and to employ standard tactics for the main proof steps, and to combine them using tacticals (primarily, the sequencing tactical THEN) to form composite tactics which solve the goals in a single application.

As indicated, we appeal to the following rule, which we shall call MINFIX, to prove goalL3 [11] :

MINFIX

$$\frac{\vdash \text{FUNG F1} \sqsubseteq \text{F1} \qquad \vdash \text{G} \equiv \text{FIX FUNG}}{\vdash \text{G} \sqsubseteq \text{F1}}$$

The ML procedure which implements this rule performs induction, as in the informal proof of the rule. The tactic which inverts the rule, MINFIXTAC, is given by:

MINFIXTAC ($\vdash$G $\equiv$ FIX FUNG)

G $\sqsubseteq$ F1
ss
A

FUNG F1 $\sqsubseteq$ F1
ss
A

It generates a proof function which calls MINFIX.

We begin the tactical proof of goalL3 by applying MINFIXTAC to obtain a subgoal whose formula is (after simplification)

λx z s. (NULL s $\Rightarrow$ z | F1(HD s)(h(z,F x))(TL s)) $\sqsubseteq$ F1

We then require a tactic which applies both sides of an inequivalence (or equivalence) to an arbitrary variable of the correct type, and generalises to that variable. As it uses extensionality (for which the standard PPLAMBDA inference rule is EXT), we call the tactic EXTTAC:

EXTTAC

$F \sqsubseteq G$
ss
A

$\forall x.\ F\ x \sqsubseteq G\ x$
ss
A

Clearly, F must have a functional type, $* \longrightarrow **$, say, relative to which x has the type *. The proof part of EXTTAC uses EXT:

EXT

$$\frac{A \vdash \forall x.\ u\ x \sqsubseteq v\ x}{A \vdash u \sqsubseteq v}$$

We may also wish to do similar reasoning about a formula whose goal is quantified already, so we include as a special case of EXTTAC the following:

EXTTAC

$\forall x.\ F\ x \sqsubseteq G\ x$
ss
A

$\forall x\ y.\ F\ x\ y \sqsubseteq G\ x\ y$
ss
A

EXTTAC fails on goals whose formulae are not of one of the two forms indicated.

Applying EXTTAC repeatedly to the current subgoal, we obtain a subgoal whose formula is:

$$\forall x\ z\ s.\ \text{NULL}\ s \Rightarrow z \mid \text{F1(HD s)(h(z,F x))(TL s)} \sqsubseteq \text{F1}\ x\ z\ s$$

We then apply INDUCTAC [thF] THEN SIMPTAC THEN (REPEAT GENTAC). This yields a basis subgoal and a step subgoal, where the latter is

(NULL s ⇒ z | F1(HD s)(h(z,(P x ⇒ f x | h(F′(g1 x),F′(g2 x))))(TL s)) ⊑ F1 x z s
SSL
∀x z s. (NULL s ⇒ z | F1(HD s)(h(z,F′ x))(TL s)) ⊑ F1 x z s

with the induction hypothesis added to the set of assumptions.

Next, for both subgoals, we wish to unfold the occurrence of F1 on the right hand side of the formula according to its definition. We write a tactic in ML to accomplish this reasoning, called UNFOLDTAC:

UNFOLDTAC (⊢F ≡ FIX FUNF)

w[F/t]
ss
A

══════════

w[(FUNF F)/t]
ss
A

The proof part uses the standard inference rule FIX:

FIX

$$\frac{A \vdash t \equiv \text{FIX FUN}}{A \vdash t \equiv \text{FUN } t}$$

It is also useful to write UNFOLDOCCSTAC:int list ⟶ thm ⟶ thm, which takes a list of occurrence numbers as a parameter and substitutes only for the corresponding occurrences.

After applying UNFOLDTAC thF1 and simplifying, the basis and step subgoals are, respectively:

```
(NULL s ⇒ z | ⊥ ) ⊆ (NULL s ⇒ z | P x ⇒
                          F1(HD s)(h(z,f x))(TL s) |
                          F1(g1 x)z(CONS(g2 x)s))
------------------------------------------------------
SSL
------------------------------------------------------

```

```
NULL s ⇒ z | F1(HD s)(h(z,(P x ⇒ f x |
                        (h(F'(g1 x), F'(g2 x)))))))(TL s) ⊆
NULL s ⇒ z | P x ⇒ F1(HD s)(h(z,f x))(TL s) |
                        F1(G1 x)z(CONS(g2 x)s)
------------------------------------------------------
SSL
------------------------------------------------------
∀x z s(NULL s ⇒ z | F1(HD s)(h(z,F' x))(TL s) ⊆  F1 x z s
```

We now wish to do case analysis on whether s is empty, so we apply the standard tactic CONDCASESTAC (which finds the first boolean-valued term, i.e. NULL s). The three subgoals derived from the basis are solved directly by simplification; their formulae are

```
z ⊆ z

⊥ ⊆ ⊥

⊥ ⊆  P x ⇒ F1(HD s)(h(z,f x))(TL s) |
           F1(g1 x)z(CONS (g2 x)s)
```

The true and undefined cases, for the step, are also solved by simplification. The remaining subgoal is

```
F1(HD s)(h(z, P x    f x | h(F'(g1 x),F'(g2 x))))(TL s) ⊆
P x ⇒ F1(HD s)(h(z,f x))(TL s) | F1(g1 x)z(CONS(g2 x)s)
------------------------------------------------------
(NULL s ≡ FF) + SSL
------------------------------------------------------
NULL s ≡ FF
∀x z s. (NULL s ⇒ z | F1(HD s)(h(z,F' x))(TL s)) ⊆  F1 x z s
```

We apply CONDCASESTAC again (to do cases analysis on whether P x is true) and simplify; the true and undefined cases are immediately solved, and the simplification based on the associativity of h is

used to produce the one remaining subgoal:

F1(HD s)(h(h(z,F′(g1 x),F′(g2 x))))(TL s) ⊑ F1(g1 x)z(CONS(g2 x)s)
(NULL s ≡ FF) + (P x ≡ FF) + SSL
NULL s ≡ FF P x ≡ FF ∀x z s. (NULL s ⇒ z | F1(HD s)(h(z,F′ x))(TL s)) ⊑ F1 x z s

We would like, at this point, to use the induction hypothesis, by matching the right hand side of its conclusion to the right hand side of the formula of the current subgoal, letting s be (CONS(g2 x)s), z be z, and x be (g1 x). That is, the assumption implies that

```
NULL(CONS(g2 x)s) ⇒ z | F1(HD(CONS(g2 x) s))
                          (h(z,F′(g1 x)))
                          (TL(CONS(g2 x)s)) ⊑
F1(g1 x)z(CONS(g2 x)s)
```

which, after simplifications based on the facts of list theory, is

```
F1(g2 x)(h(z,F′(g1 x)) s  ⊑  F1(g1 x)z(CONS(g2 x))s
```

If we could now prove a subgoal with the formula

```
F1(HD s)(h(h(z,F′(g1 x), F′(g2 x))))(TL s)  ⊑
F1(g2 x)(h(z F′(g1 x)))s
```

we would be finished, by transitivity. This, however, is another instance of matching the right hand side of the conclusion of the induction hypothesis to the right hand side of the formula to be proved, letting s be s, x be (g2 x) and z be (h(x,F′(g1 x))), and simplifying.

To complete the generation of the proof of goalL3 we write a tactic, to be applied twice in succession, in this case, which uses an assumption exactly as we have just done informally. It is more complicated than USEASSUMPTAC, since the formula of the goal is not necessarily an instance of one of the assumptions. Here, we generate an intermediate subgoal to be combined later with the assumption, using transitivity. USEASSUMPRHSTAC (for matching to the right hand side of an assumption) captures the reasoning above:

USEASSUMPRHSTAC

$t1' \subseteq t3'$
ss
... $\forall x1 \ldots xn.\ t2 \subseteq t3$ ...

$$\overline{}$$

$t1' \subseteq t2'$
ss
... $\forall x1 \ldots xn.\ t2 \subseteq t3$ ...

where t2′ is t2 with the substitutions for x1,...,xn determined by matching t3 to t3′ (that is, instantiating for the variables in t3). If no formula in the list of assumptions matches, the tactic fails. The proof function uses the standard rule TRANS.

We occasionally match assumptions to the left hand side of formulae, so we write the dual tactic USEASSUMPLHSTAC:

USEASSUMPLHSTAC

$t1' \sqsubseteq t3'$
ss
... $\forall x1...xn.\ t1 \sqsubseteq t2$ ...

$$\overline{\overline{\qquad\qquad\qquad}}$$

$t2' \sqsubseteq t3'$
ss
... $\forall x1...xn.\ t1 \sqsubseteq t2$ ...

where t2′ is t2 modulo the substitutions for x1,...,xn determined by matching t1 to t1′. This pair of tactics does not address the general issue of reasoning about inequivalences, but it does faithfully reflect the reasoning used in <u>this</u> proof, a very common chain of reasoning in proofs of inequivalences by induction.

We observe that <u>if</u> exactly one application of USEASSUMPRHSTAC or USEASSUMPLHSTAC solves a goal, then (i) either tactic will suffice, and (ii) either tactic returns the trivially easy subgoal with formula $t2' \sqsubseteq t2'$; that is, the assumption <u>a priori</u> achieves the goal. In the latter case, USEASSUMPTAC is an adequate and more direct way of solving the goal (more direct because it does not involve a subsequent call of SIMPTAC).

The composite tactic, TACL3, which solves <u>goalL3</u> is now complete:

```
TACL3

    (MINFIXTAC thG)+
    EXTTAC+
    (INDUCTAC [thF])+
    GENTAC*
    (UNFOLDOCCSTAC 2 thF1)+
    (CONDCASESTAC+)*
    (USEASSUMPRHSTAC+)*
```

TACL3 applied to goalL3 yields an empty list of subgoals and a proof which when applied to the empty list of theorems returns thL3. Although in explaining TACL3 we have applied the component tactics one by one, the composite tactic TACL3 solves the goal in a single application, and the proof function produces the theorem in one application. The work of doing the proof is in formalising the problem and in designing and implementing the derived tactics. One would hope that the tactics developed for this proof are useful in other, similar proofs. This is shown to be so in the next section, and in later chapters. We examine another recursion removal problem, and go on to generalise the tactics developed so far.

We do not describe the development of the compound tactics which solve goalL2 and goallemL2, but simply state them below. The correspondence to the informal proofs is obvious.

TACL2, to solve goalL2

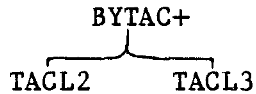
```
EXTTAC+
(INDUCTAC [thF1])+
GENTAC*
(UNFOLDTAC thG)+
(CONDCASESTAC+)*
(UNFOLDTAC thF)+
USEASSUMPLHSTAC+
((UNFOLDOCCSTAC 1 thG)+)*
```

TAClemL2, to solve goallemL2

```
(INDUCTAC [thF1])+
GENTAC*
(CONDCASESTAC+)*
USEASSUMPTAC
```

With LemmaL2 added to the simpset of goalL4, the following tactic solves goalL4 (we extend our informal notation to allow branching

into columns to abbreviate a use of the tactical THENL):



so that TACL2 and TACL3 are applied, respectively, to the two subgoals (goalL2 and goalL3) produced by the application of BYTAC to goalL4. goalL5 is solved by

((UNFOLDTAC thExp)+)*

and the main goal, goalL0, is solved by SIMPTAC.

The Counter Problem

The Problem

The third schema problem, in which recursion is implemented by use of an integer counter, shares many of the same patterns of inference with the List Stack proof, and therefore, its machine proof is achieved by similar tactics. We recount the problem and solution more briefly than before.

The function F has a nested recursive call:

$$F\ x = P\ x \Rightarrow f\ x \mid F(h(F(g\ x)))$$

where h and P are assumed strict. The recursion is implemented by using a counter n:

$$F1\ x\ n = P\ x \Rightarrow (n{=}0 \Rightarrow f\ x \mid F1(h(f\ x))(n-1)) \mid F1(g\ x)(n+1)$$

We show[12,13]

<u>Theorem 2.7</u>
$F1\ x\ 0 = F\ x$

Again, F and F1 are defined as the least fixed points of functionals FUNF and FUNF1 in the obvious way. The more general relation one has to prove is

$$F1\ x\ n \quad = \quad F((h \circ F)^{n}\ x)$$

that is, F1 on x with counter n is equal to the result of applying (h o F) to x, n times. We define a function Expo to do the exponentiation suggested by the above notation:

$$\text{Expo}\ F\ h\ x\ n = (n{=}0) \Rightarrow F\ x \mid \text{Expo}\ F\ h\ (h(F\ x))(n-1)$$

where Expo is formally defined as the least fixed point of a functional FUNExpo, again in the obvious way. The relation to be proved is

<u>Theorem 2.8</u>
$\forall$x n. Fl x n $\equiv$ Expo F h x n

Expo is analogous to Exp in the List Stack Problem. As before, we introduce a new function (H) to `freeze' the functional arguments:

H x n = (n=0) $\Rightarrow$ F x | H(h(F x))(n-1)

where H is defined as FIX FUNH. Also as before, we prove

<u>Theorem 2.10</u>
Fl $\sqsubseteq$ H

<u>Theorem 2.11</u>
H $\sqsubseteq$ Fl

By application of the By-law, we prove

<u>Theorem 2.9</u>
H = Expo F h

and <u>Theorem 2.8</u> follows.

We summarise the proofs first.

<u>Plan for Proof of Theorem 2.10</u>
By induction on Fl. For both the basis and the step, we do case analysis on P x, and further cases, in the step, on n=0. <u>Lemma 2.13</u> (below) is needed.

<u>Plan for Proof of Theorem 2.11</u>
By proving that FUNH Fl $\sqsubseteq$ Fl. Then by either

<u>Method (i)</u>
By induction on F, then cases on n=0, followed by cases on P x. <u>Lemma 2.12</u> and <u>Lemma 2.13</u> (below) are required.

Method (ii)

By cases on n=0, appealing to Lemma 2.14 (below) in which most of the work is done.

Lemma 2.12

$\forall n.Fl \perp n \equiv \perp$

Lemma 2.13

$\forall x.Fl\ x \perp \equiv \perp$

Lemma 2.14

$\forall x\ n.\ F\ x \sqsubseteq Fl\ x\ 0 \quad \& \quad Fl(h(F\ x))n \sqsubseteq Fl\ x\ (n+1)$

Plan for Proof of Lemma 2.12

By unfolding the definition of Fl, and using the strictness of P.

Plan for Proof of Lemma 2.13

By induction on Fl and cases on P x.

Plan for Proof of Lemma 2.14

By induction on both occurrences of F, and cases on P x. The true case is by further cases on n=0. The false case is by two uses of the induction hypothesis (one use of each conjunct) for the first part, and two uses (both of the second conjunct) for the second part. Lemma 2.12 and Lemma 2.13 are needed.

The proof for Lemma 2.14, Theorem 2.10 and Theorem 2.11 Method (i) are very similar to the proofs for Lemma 2.6, Theorem 2.4 and Theorem 2.5 from the List Stack problem. We concentrate, therefore, on the proof of Theorem 2.11, Method (ii), and the accompanying Lemma 2.14.

Proof of Lemma 2.14

We prove both formulae together, by induction on F.

Basis
Easy, given the strictness of h, and Lemma 2.12.

Step

Assume
$\forall x.\ F'\ x \sqsubseteq Fl\ x\ 0 \quad \& \quad Fl(h(F'\ x))\ n \sqsubseteq Fl\ x\ (n+1)$

Show
$(P\ x \Rightarrow f\ x \mid F'(h(F'(g\ x)))) \sqsubseteq Fl\ x\ 0 \quad \&$
$Fl(h(P\ x \Rightarrow f\ x \mid F'(h(F'(g\ x)))))n \sqsubseteq Fl\ x\ (n+1)$

We consider cases on whether P x is true.

Case P x = TT
The first part is easy, by unfolding Fl.

Second Part
We must show

$Fl(h(f\ x))n \sqsubseteq Fl\ x\ (n+1)$

RHS = $Fl(h(f\ x))(n+1)$ by unfolding Fl.

Case P x = FF

First Part
We must show

$F'(h(F'(g\ x))) \sqsubseteq Fl\ x\ 0$

RHS $= Fl(g\ x)(0+1)$ by unfolding Fl
LHS $\sqsubseteq Fl(h(F'(g\ x)))0$ by hypothesis, first part

$\sqsubseteq Fl(g\ x)(0+1)$ by hypothesis, second part

Second Part
We must show

$Fl(h(F'(h(F'(g\ x)))))n \sqsubseteq Fl\ x\ (n+1)$

$= Fl(g\ x)(n+1+1)$
by unfolding Fl

$\sqsubseteq Fl(h(F'(g\ x)))(n+1)$
by hypothesis, second part

$\sqsubseteq Fl(h(F'(h(F'(g\ x)))))n$
by hypothesis, second part

Q.E.D.

Once this lemma has been proved, the proof of Theorem 2.10 is not difficult.

Proof of Theorem 2.10

It is sufficient to show

$$((n=0) \Rightarrow F\ x \mid F1(h(F\ x))(n-1)) \sqsubseteq F1\ x\ n$$

We consider cases on whether n=0, and use the two parts of Lemma 2.13 in the two cases, respectively.

The Formalisation

We work in a theory of integers. The theory has a new type (nat, for natural number) and new constants, including

ZERO:nat

SUCC:nat $\rightarrow$ nat

PRED:nat $\rightarrow$ nat

ISZERO:nat $\rightarrow$ tr

We assume that the following axioms and/or theorems are available:

$\vdash$ SUCC $\perp \equiv \perp$

$\vdash$ ISZERO $\perp \equiv \perp$

$\vdash$ $\forall$n.ISZERO n $\equiv$ TT IMP ISZERO(SUCC n) $\equiv$ FF

$\vdash$ $\forall$n.ISZEROn $\equiv$ FF IMP ISZERO (SUCC n) $\equiv$ FF

$\vdash$ $\forall$n.ISZERO n $\equiv$ TT IMP PRED(SUCC n) $\equiv$ n

$\vdash$ $\forall$n.ISZERO n $\equiv$ FF IMP PRED(SUCC n) $\equiv$ n

$\vdash$ $\forall$n.ISZERO n $\equiv \perp$ IMP n $\equiv \perp$

$\vdash$ ISZERO ZERO $\equiv$ TT

In addition, we introduce assumptions for the strictness of P and h:

$\vdash h \perp \equiv \perp$

$\vdash P \perp \equiv \perp$

and to define the functions F, F1, H and Expo:

thF

$\vdash F \equiv \quad FIX(\lambda F'\ x.\ P\ x \Rightarrow f\ x \mid F'(h(F'(g\ x))))$

thF1

$$\vdash F1 \equiv FIX(\lambda F1'\ x\ n.\ P\ x \Rightarrow (ISZERO\ n \Rightarrow f\ x \mid F1'(h(f\ x))(PRED\ n)) \mid F1'(g\ x)(SUCC\ n))$$

and similarly for thExpo and thH. We add all of the facts, except

$\vdash ISZERO\ n \equiv \perp\ IMP\ \ n \equiv \perp$

which would cause an infinite cycle of simplifications, to BASICSS, to form a simpset called SSC (for simpset for Counter). (Any rule of the form w IMP $t1 \equiv t2$ will `loop` as a simplification rule if t1 occurs in w, because of the way simplification works in LCF. The reason is that the simplifier can replace occurrences of t1 by t2 in a formula being simplified *if* it can first prove w by simplification. But since w contains t1, the simplifier will try to replace *that* t1 by t2 by first showing w, and so on *ad infinitum*.)

At any rate, we can now define the goals for this problem.

The main goal is to prove thC0, corresponding to Theorem2.7:

thC0

$\vdash \forall x.\ F1\ x\ ZERO \equiv F\ x$

where thC0 achieves the main goal, goalC0:

goalC0

$\forall x.\ F1\ x\ ZERO \equiv F\ x$
thC4 + SSC

This requires proving a theorem, thC4, achieving the goal goalC4:

goalC4

$F1 \equiv Expo\ F\ h$
SSC

As before, application of BYTAC to goalC4 generates the subgoals goalC2 and goalC3:

goalC2

$F1 \sqsubseteq H$
LemmaC2 + SSC

goalC3

$H \sqsubseteq F1$
LemmaC1 + LemmaC2 + SSC
LemmaC3

The theorems which achieve these two goals, thC2 and thC3 respectively, correspond to Theorem 2.10 and Theorem 2.11. The two subgoals need the lemmas indicated in their simpsets. LemmaC1, LemmaC2 and LemmaC3, corresponding, respectively, to Theorem 2.12, Theorem 2.13 and Theorem 2.14, achieve the goals goallemC1, goallemC2 and goallemC3:

goallemC1

$\forall$n. F1 $\perp$ n $\equiv$ $\perp$
SSC

goallemC2

$\forall$x. F1 x $\perp$ $\equiv$ $\perp$
SSC

goallemC3

$\forall$x n. F x $\sqsubseteq$ F1 x ZERO & F1(h(F x))n $\sqsubseteq$ F1 x (SUCC n)
LemmaC1 + LemmaC2 + SSC

With minor modifications (changes of parameter, etc.), the tactics TAClemL2, TACL2 and TACL3, from the List Stack proof, solve goals goallemC2, goalC2 and goalC3, respectively (the latter by Method (i) and without LemmaC3 as an assumption). We examine the proofs of goallemC3 and goalC3, by Method (ii) only, in LCF.

The Counter Proof in LCF

We commence the tactical proof of LemmaC3 by applying

```
(INDUCTAC [thF])+
GENTAC*
(UNFOLDOCCSTAC [1;3] thF1)+
```

to mirror the informal proof. (Since in both subsequent cases we unfold the right hand side's occurrence of F1, we unfold it here at the outset.) We then apply CONDCASESTAC to do case analysis on P x. In the true case, the first part of the conjunctive subgoal is solved directly by simplification, and we are left with the subgoal

Fl(h(f x))n ⊑ Fl(h(f x))(PRED(SUCC n))
(P x ≡ TT) + LemmaCl + LemmaC2 + SSC
P x ≡ TT ∀x n. F′ x ⊑ Fl x ZERO & Fl(h(F′ x))n ⊑ Fl x (SUCC n)

This is straightforward if (ISZERO n) is defined, for we have rules to simplify (PRED(SUCC n)) to n to be applied conditionally on whether (ISZERO n) is true or false. If (ISZERO n) is undefined, we know that n is undefined, and therefore, by LemmaC2, the whole left hand side of the formula is undefined. The rest can be managed by simplification.

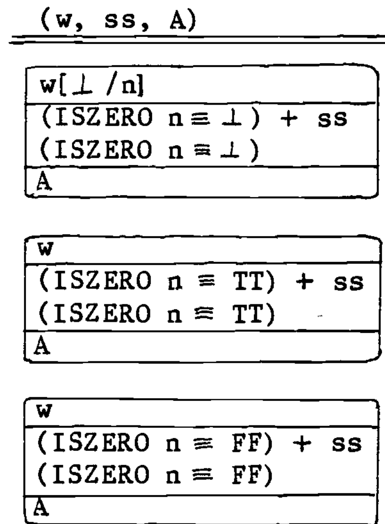
CONDCASESTAC does not suffice, as the term (ISZERO n) does not actually occur in the formula to be proved. CASESTAC (ISZERO n) does not quite do either, because we cannot use the theorem

$$\forall n.\text{ISZERO } n \equiv \bot \text{ IMP } n \equiv \bot$$

as a simplification rule, yet we do have to make this simplification in the case that ISZERO n ≡ ⊥. Our solution is to write a tactic called NATCASESTAC, similar to CASESTAC (ISZERO n), except that it *finds* a term n of type nat, does case analysis on (ISZERO n), and, for the subgoal corresponding to the assumption that ISZERO n ≡ ⊥, makes a direct substitution of ⊥ for n. This avoids having to use the 'dangerous' theorem as a simprule. The proof part of the tactic justifies this substitution by the theorem in question. Used in place of CONDCASESTAC+, NATCASESTAC+ has the same effect *if* (ISZERO n) is the boolean-valued term found by CONDCASESTAC and n is the term found by NATCASESTAC. (CASESTAC (ISZERO n))+ has the same effect as NATCASESTAC+ *if* the term (ISZERO n) actually occurs in the formula, and n is the term found by NATCASESTAC. [14]

Obviously, NATCASESTAC is meaningful only in the theory of numbers, or a descendent of such a theory, as it refers to the type nat and the constant ISZERO. It is depicted by

NATCASESTAC

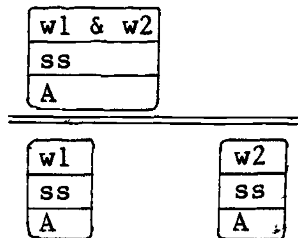


After applying NATCASESTAC to the goal and simplifying, we are left with the subgoal for the false case of P x, which has the formula

$$F'(h(F'(g\ x))) \sqsubseteq F1(g\ x)(SUCC\ 0) \quad \& \quad F1(h(F'(h(F'(g\ x)))))n \sqsubseteq F1(g\ x)(SUCC(SUCC\ n))$$

We employ a tactic to divide conjunctive sugoals into two subgoals:

CONJTAC



CONJTAC inverts the basic inference rule CONJ:

CONJ

$$\frac{A1 \vdash w1 \qquad\qquad A2 \vdash w2}{A1 \cup A2 \vdash w1 \;\&\; w2}$$

Next, for each of the subgoals returned by the application of CONJTAC, we use the induction hypothesis, each conjunct. We write a tactic which enables this by noting any assumptions in the assumption list of a goal which are of the form

$\forall$xl...xn. wl & w2

and adding to the assumptions list two further assumptions:

$\forall$xl...xn. wl

$\forall$xl...xn. w2

We call this tactic CONJASSUMPTAC. Clearly, a theorem, th, achieving the subgoal with the supplemented assumption list also achieves the original goal, as the extra hypotheses of th (if it has them) have only to be specialised, conjoined, and generalised again. The proof function of CONJASSUMPTAC does this.[15]

To complete the proof, we apply USEASSUMPRHSTAC repeatedly. The separate conjuncts of the induction hypothesis will thus be used as in the informal proof, and goallemC3 is solved.

The composite tactic, TAClemC3, which solves goallemC3 in one stroke is:

TAClemC3

```
(INDUCTAC [thF])+
GENTAC*
CONDCASESTAC+
(UNFOLDOCCSTAC [1;3] thF1)+
        ┌──────────┴──────────┐
NATCASESTAC+              CONJTAC
                          CONJASSUMPTAC
                          (USEASSUMPRHSTAC+)*
```

Once LemmaC3 has been proved and placed in the assumption list of goalC3, the proof of goalC3 is quite short. We process the assumption list so that both conjuncts of the conclusion of LemmaC3 (generalised) appear, by applying CONJASSUMPTAC. We then apply CONDCASESTAC (or NATCASESTAC, which is equivalent in this instance) and simplify. To solve goalC3, we have

TACC3

```
CONJASSUMPTAC
CONDCASESTAC+
```

Finally, the original goalC0, with thC4 in its simpset, is solved by unfolding F1 and simplifying, i.e. by

```
(UNFOLDTAC thF1)+
```

Conclusions: Towards a General Schema Tactic

We speculate briefly, in this section, on some generalisations based on the Accumulator, List Stack, and Counter proofs. We aim at writing a uniform, general tactic to include as instances most of the composite tactics discussed in this chapter. Although we have not implemented the general tactic in ML, we sketch its design.

It would be possible, for example, to write a tactic called INDUCTCHOOSETAC to take as a parameter a list of function definitions, and choose, according to the list and to the formula of a goal, the variables on which to induct. For proving an equivalence of the form $F \equiv G$, where the list includes $\vdash F \equiv$ FIX FUNF and $\vdash G \equiv$ FIX FUNG, F and G should be chosen. For an inequivalence $F \sqsubseteq G$, F should be chosen. After doing induction, INDUCTCHOOSETAC would do simplification.

It would be equally simple to write a tactic to use an assumption in the appropriate manner (USEASSUMPCHOOSETAC, say). As long as the component tactics were written to fail where inapplicable,

(USEASSUMPTAC ORELSE USEASSUMPLHSTAC ORELSE USEASSUMPRHSTAC)+

is a good definition for USEASSUMPCHOOSETAC. This could be refined by including the heuristic that when neither USEASSUMLHSTAC nor USEASSUMPRHSTAC fails, but neither, after simplification, solves the goal, the preferred one is the one which does not produce a subgoal whose formula includes the current induction variable on both of its sides. For example, in the proof of thL2 (which we have not shown in this presentation), at the point at which it is appropriate to

use the induction hypothesis, the subgoal is

```
F1′(g1 x)z(CONS(g2 x)s) ⊑
G(HD s)(h(z,h(F(g1 x),F(g2 x))))(TL s)
-----------------------------------------
...
-----------------------------------------
...
∀x z s. F1′ x z s ⊑ G x z s
...
```

(F1 is the induction variable.) Application of USEASSUMPRHSTAC produces a subgoal:

```
F1′(g1 x)z(CONS(g2 x)s) ⊑
F1′(HD s)(h(z,(F(g1 x),F(g2 x))))(TL s)
-----------------------------------------
...
-----------------------------------------
...
```

which does not advance the proof (whereas use of USEASSUMPLHSTAC does).

We can generalise and define an UNFOLDCHOOSETAC, which, like INDUCTCHOOSETAC, would take a list of function definitions as a parameter and select appropriate functions (and occurrences of the functions) to unfold. After unfolding, again, it would simplify. This tactic requires rather more thought than the others, as the criteria for deciding whether and where to unfold are quite heuristic. One does not, for example, wish to unfold a function variable some of whose arguments are not present, such as F in the expression Exp F h x z s. However, even for the instances of unfolding in the proofs discussed in this chapter, we have found no very simple set of heuristics which is adequate. Some of the choice, though, can be avoided by carefully including theorems to be used as simplifications. If we prove, for example,

$\vdash$ Exp F h x z (CONS x′ s′) ≡ Exp F h x′ (h(z,F x)) s′

and use the theorem as a rewrite rule (in the proof of Theorem 2.4) then only those occurrences of Exp which have (CONS ...) as their fifth argument will be unfolded. Occurrences of this sort unfold to become conditionals of the form

(NULL(CONS ...)) ⇒ ... | ...

for which we have further simplification rules. Similarly, if we prove

$\vdash$NULL s ≡ FF IMP
Exp F h x z s ≡ Exp F h (HD s)(h(z,F x))(TL s)

we can use the theorem as a conditional simplification; the simplification will only be made in case NULL s is false, as is appropriate.

Again, in the proof of Theorem 2.5, we could prove, and include as simprules

$\vdash$NULL s ≡ TT IMP F1 x z s ≡ z

$\vdash$NULL s ≡ FF IMP F1 x z s ≡ P x ⇒ F1(HD s)(h(z,f x))(TL s) |
F1(g1 x)z(CONS(g2 x)s)

$\vdash$NULL s ≡ ⊥ IMP F1 x z s ≡ ⊥

so that F1, in the expression F1 x z s, is unfolded according to its definition once case analysis has been done on the term (NULL s), since the three new simprules then apply to the three cases. However, in the expression

```
F1(HD s)(h(z, (P x ⇒ f x | h(F′(g1 x),F′(g2 x)))))(TL s)
```

(which also occurs in the proof) F1 is not unfolded -- as long as case analysis is not done on (NULL(HD s)), which it is not. Again, this is the desired effect.

In this fashion, we could arrange for many (in the proofs in this chapter, <u>all</u>) of the choices about unfolding to be made in the course of simplification. This methodology proves to be of great use in Chapter 3. In any case, we assume for the moment that some adequate UNFOLDCHOOSETAC can be designed (or combination of SIMPTAC and carefully chosen simprules). We can then state a general tactic, SCHEMATAC, of which TACA, TACL2, TAClemC3 and TACL3 are instances:

```
SCHEMATAC

     (EXTTAC*)+
     INDUCTCHOOSETAC list
     GENTAC*
     (CHOOSECASESTAC* ORELSE UNFOLDCHOOSETAC* ORELSE
      USEASSUMPCHOOSETAC*)*
```

where list is the list of all relevant function definitions, typically, for some boolean-valued term B, of the form

$$\vdash F \equiv FIX(\lambda F'\ x1...xn.\ B \Rightarrow t1 \mid t2)$$

After induction and stripping of variables, the general tactic tries case analysis and unfolding of function variables until the induction hypothesis is applicable. To generalise further, one could add other tactics (e.g. CONJTAC) to the last line to cope with other `shapes' of formulae or other proof situations. (By

adding CONJTAC, TAClemC3 would become an instance of SCHEMATAC.) This hypothetical tactic naturally reflects our reasoning in the informal proofs. In addition, it would appear to be useful in many other proofs about recursively defined functions.

In conclusion, we have illustrated, in three case studies, the generation of formal machine proofs by the design of tactics which (i) represent informal proof plans, and (ii) abstract formal proofs to provide high level proof outlines. We have speculated about a general tactic for the proofs considered, and possibly for other proofs about recursively defined functions.

We go on to consider more difficult problems for which many of the tactics derived in this chapter prove of use.

Notes for Chapter 2

1. F1 is, of course, recursively defined, but all recursive calls of it are `outermost`. `Iterative-recursive` would perhaps be more appropriate.

2. As an instance of this schema, we take x to have integer type, g to be $(\lambda n.n-1)$, f to be $(\lambda n.1)$, P to be a test-if-zero predicate ISZERO, and h to be multiplication (TIMES) with identity 1. Then F is the familiar factorial function, and F1 is

F1 x z = ISZERO x ⇒ z | F1(n-1)(TIMES(z,n))

and it is true that F1 x 1 = F n. However, we leave the non-logical constants uninterpreted here.

3. One could, alternatively, introduce an LCF theory in which P, h, e, etc., were new constants and the assumptions discussed were axioms. As this is not a very interesting theory we leave the variables free, and content ourselves with assumptions. The choice is immaterial to the proof.

4. Intuitively, F1 implements F by using depth-first search and a stack. Viewing the computation of F as a binary tree, z denotes the value of the left subtree computed so far. In the second call of F1, the `stacking` of the value (g2 x) corresponds to the second recursive call of F, and the call of F1 with the argument (g1 x) corresponds to the first recursive call of F. In the first call of F1, on x, z and s, the first deferred element on the `stack` is taken off, and (f x) is combined with the result accumulated so far, i.e. z.

5. To give an instance of this schema, let l range over LISP-style (i.e. non-flat) lists, g1 be Car, g2 be Cdr, h be Append, P be a function AtomOrNil, to test for atomic or empty lists, IsNull a function to test for empty lists, and f be the function $(\lambda l.$IsNull l ⇒ Nil | List l). (List is the usual list function.) Then F is a flattening function for lists:

```
F l = AtomOrNil l ⇒ (IsNull l ⇒ Nil | List l) |
                     Append(F(Car l),F(Cdr l))
```

and F1, with accumulator z and stack s, is an iterative version:

```
F1 l z s = IsNull l ⇒ z |
           AtomOrNil l ⇒ F1(HD s)(Append(z,(IsNull l ⇒
                                 Nil | List l)))(TL s) |
                       F1(Car l)z(CONS(Cdr l)s)
```

and it is the case that F1 l Nil [NIL] = F x.

6. <u>Any</u> one-element list would do; [NIL] is just convenient.

$\forall$x y. F1 x e [y] = F x

would perhaps have been a better theorem to prove.

7. G is not, in fact, necessary. We could instead take a fixed point with the functional arguments outside:

$$\text{Exp F h} = \lambda \text{x z s. NULL s} \Rightarrow \text{z} \mid \text{Exp F h (HD s)(h(z,F x))(TL s)}$$

$$= \text{FIX}(\lambda \text{ExpFh x z s. NULL s} \Rightarrow \text{z} \mid \text{ExpFh(HD s)(h(z,F x))(TL s))}$$

where the function ExpFh is of appropriate type. If we let ExpFh = FIX FUNExpFh, we can then show that FUNExpFh F1 $\subseteq$ F1 by induction on F. Introducing G to abbreviate ExpFh simply makes the proof look neater. See later discussion of the By-law, p. 58-61.

8. To motivate this, we call the left hand side and the right hand side, respectively, of the formula on p. 59 $\mathcal{L}$ and $\mathcal{R}$. Then

$$\mathcal{L}\text{F1...Fn} = \text{FIX}(\Phi\,\text{F1...Fn})$$

$$= \Phi\ \text{F1...Fn}\ (\mathcal{L}\text{F1...Fn})$$

On the other side,

$$\mathcal{R}\text{F1...Fn} = \text{FIX}(\lambda \text{E F1}'\text{...Fn}'.\Phi\text{F1}'\text{...Fn}'(\text{E F1}'\text{...Fn}'))\text{F1...Fn}$$

which we call FIX α F1...Fn. Then

$$\text{FIX}\alpha\text{F1...Fn}\ (\alpha\,\text{FIX}\,\alpha)\ \text{F1...Fn}$$

$$= (\lambda \text{E F1}'\text{...Fn}'.\Phi\text{F1}'\text{...Fn}'\ (\text{E F1}'\text{...Fn}'))\ (\text{FIX}\alpha)\ \text{F1...Fn}$$

$$= \Phi\ \text{F1...Fn}\ ((\text{FIX}\,\alpha)\ \text{F1...Fn})$$

$$= \Phi\ \text{F1...Fn}\ (\mathcal{R}\text{F1...Fn})$$

so that $\mathcal{L}$ and $\mathcal{R}$ can be seen to satisfy the same equations.

9. The procedure which implements this rule proves the By-law at each invocation, but we could instead have proved the theorem as a fact in some LCF theory, just once, and saved it for later use. It is convenient to have it avalable in procedural form, however, to circumvent having to instantiate it when using it, and to avoid having to supply a theorem as a parameter to BYTAC. We also choose a procedural representation in several other places. Rules such as INDUCT, of course, _must_ be represented as rules and cannot be proved as theorems. This point is discussed in the Conclusions.

10. That is,

Basis
$\perp = \perp$ F1...Fn

Assume
x′ = y′ F1...Fn

Show
(Φ F1...Fn) x′ =
(λE F1′...Fn′.Φ F1′...Fn′ (E F1′...Fn′))y′ F1...Fn

That is, ($\bar{\Phi}$ F1...Fn)x′ = $\bar{\Phi}$F1...Fn (y′ F1...Fn); but the right hand side is $\bar{\Phi}$ F1...Fn x′ by hypothesis.

11. The same comment as (9.) applies to MINFIX.

12. Intuitively, F can be evaluated by working on the inner call of F whenever P is false, and keeping count of the (F o h)′s waiting to be applied. F1 simulates F by testing whether P holds, and if it does, testing whether n=0, that is, whether there are any (F o h)′s pending. If so, F1 is called again, to simulate the outer call of F, on (h(f x)) (where (h(f x)) corresponds to h and the inner call of F), and the counter is decremented; if n is 0, (f x) is returned. If P does not hold, F1 is called on (g x) (to simulate the inner call of F) and the counter is incremented (corresponding to the outer call of F).

13. There are not very many natural examples of this schema. A related example, in that it has nested recursive calls, is Ackermann′s function, A:

A(x,y) = ISZERO x $\Rightarrow$ y+1 | ISZERO y $\Rightarrow$ A(x-1,1) |
A(x-1,A(x,y-1))

The example is from Manna and Waldinger [22].

14. A related tactic could be written to use the `dangerous` simplification rule DEF x$\equiv\perp$ IMP x $\equiv\perp$ where DEF is the definedness predicate which is $\perp$ on $\perp$ and TT otherwise. In the same manner, it would make the substitution of $\perp$ for x immediately, in the undefined case.

15. It is possible to describe a whole class of tactics which process goals simply by changing the assumption lists. Another useful tactic might scan the assumption list of a goal for a formula of the form

$\forall$x1...xn. w1 IMP w2

and another formula, w1′, matching w1 with certain instantiations, and add to the assumption list of the goal the formula w2′, where w2′ is w2 with the instantiations used in matching w1′ to w1. This class of tactics forms a sort of simplification facility at the formula level.

Chapter 3: The Russell Compiler

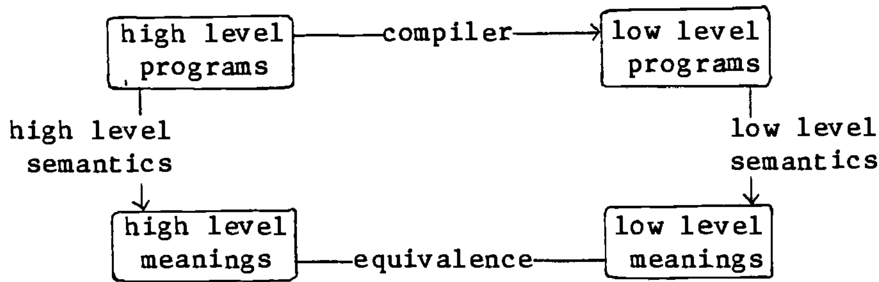
In the next two chapters, we study the informal and machine generated proofs of correctness of compilers for two simple high level languages. The ultimate aim is to verify a standard implementation of a realistic programming language. As a step toward this goal, we have partially factored the process of compilation into stages, so that each stage concerns itself with the implementation of one (or of several related) high level construct(s). The factorisation is intended to (i) make the proofs easier and more modifiable, (ii) focus attention on difficulties raised by particular featues, and (iii) be conceptually coherent, as far as possible.

The transformation described in this chapter maps a high level language whose features include while loops and conditionals to a low level language whose (labelled) statements include go-to's and conditional jumps. The formulation is based closely on one given by Russell [42]. In this chapter, we describe the problem, present an informal proof, and give an account of the successful generation of the proof in LCF. The presentation of the machine proof is somewhat idealised, but we mention the idealisations where relevant.

The second transformation concerns the implementation of procedure declarations and calls in a block-structured high level language. In Chapter 4 we describe the problem and the informal proof, speculating on the generation of the proof in LCF.

In both cases, our approach has been to supply denotational semantics of the languages in question, to represent the compiler as a function from high level to low level programs, and to prove the

preservation of the semantics under compilation. The following diagram illustrates these relations:



In both cases, we have defined high and low level languages which isolate the major difficulties raised by the compilation of the relevant constructs, and have given semantics which are convenient and natural for the proofs. For studying while loops and conditionals, we use Russell's pair of languages. The high level language contains essentially only the two constructs of interest. We follow Russell in giving a standard direct and a continuation semantics, repectively, for the two languages. For coping with procedure declaration and call, we define a high level language containing just declarations and calls, and in which all declarations are of (parameterless) procedures; we give a low level language whose operational semantics reflect an activation stack implementation.

The current studies differ from the schema studies described in Chapter 2 in being 'longitudinal'; here we relate two different languages, rather than studying properties of one language. We explicitly define the semantics of both. The machine proofs rely more heavily than before on LCF's theory-building facility for their formulation and organisation. The 'proof engineering' aspects of

the generation of the proofs, particularly in the current chapter, occupy rather more of our attention than in Chapter 2, as the proofs are long and complex, and require careful planning and management. We conclude, nonetheless, that the proofs in Chapter 3, and the proofs outlined in Chapter 4, call for many of the tactics derived in Chapter 2, and that the composite tactics used in Chapters 3 and 4 have much the same shape as the tactics we have already seen.

The Problem

The high level language given by Russell is shown below. We let p, p1 and p2 range over a domain HPROGRAM of high level programs, I over a domain ID of identifiers, and exp over a domain EXP of expressions.

```
p ::= assign(I,exp) |
      if exp then p1 else p2 |
      while exp p1 |
      p1;p2
```

A program can be an assignment (this case is present just to provide an atomic case), a conditional, a while loop, or a sequence of two programs.

For the low level language, I and exp are as above, q ranges over a domain LPROGRAM of low level programs, t over a domain STATEMENT of statements, and L over a five-element domain of labels, called LABEL.

```
q ::= <L1:t1,...,Ln:tn, L>

t ::= assign(I,exp) |
      ifnot(exp,L4) |
      goto L |
      q

L ::= L1 | L2 | L3 | L4 | L5
```

Low level programs are sequences of labelled statements followed by a terminating label; statements can be if-not jumps, jumps to labels, assignments as at the high level, or whole programs again. (Thus there is block structuring of a sort in the low level language; we do not allow jumps out of blocks. By this technique, we can limit ourselves to the use of a finite number of labels and so separate the problem of compiling while and conditional statements from the problem of generating unique new label names.)

The compiling algorithm, C:HPROGRAM ⟶ LPROGRAM, is defined for the various high level constructs by clauses. We use Quine corners to map concrete to abstract syntax.

```
C⌜assign(I,exp)⌝            =     L1: assign(I,exp)
                                  L2:

C⌜if exp then p1 else p2⌝   =     L1: ifnot(exp,L4)
                                  L2: C(p1)
                                  L3: goto L5
                                  L4: C(p2)
                                  L5:

C⌜while exp p1⌝             =     L1: ifnot(exp,L4)
                                  L2: C(p1)
                                  L3: goto L1
                                  L4:

C⌜p1;p2⌝                    =     L1: C(p1)
                                  L2: C(p2)
                                  L3:
```

The five labels, therefore, have fixed functions, in the compiled images of the four high level constructs [1].

The high level semantic function maps high level programs to store transformations, where stores hold the current values of identifiers. We let s range over a domain STORE = ID $\longrightarrow$ VALUE, where VALUE is an (unspecified) domain of values (OTHERVALUES) plus the truth values -- VALUE = OTHERVALUES + tr. We need a function, eval, to evaluate expressions in stores to produce truth-valued elements of VALUE; we introduce eval:(EXP $\times$ STORE) $\longrightarrow$ tr.

The high level semantic function, hsem, has type

hsem:HPROGRAM $\longrightarrow$ STORE $\longrightarrow$ STORE

We use the usual notation for extending functions; f[x/y] means $\lambda y'. \ y'=y \Rightarrow x \mid f\ y'$. We define hsem by clauses:

hsem ⟦<u>assign</u>(I,exp)⟧ s = s[eval(exp,s)/I]

hsem ⟦<u>if</u> exp <u>then</u> p1 <u>else</u> p2⟧ s = eval(exp,s) ⇒ hsem ⟦p1⟧ s |
hsem ⟦p2⟧ s

hsem ⟦<u>while</u> exp p1⟧ s = eval(exp,s) ⇒
(hsem ⟦<u>while</u> exp p1⟧)
(hsem ⟦p1⟧ s) |
s

hsem ⟦p1;p2⟧ s = (hsem ⟦p2⟧)(hsem ⟦p1⟧ s)

We define hsem as the least fixed point of a functional HSEM, in the obvious way.

For the low level semantics, we define a domain of continuations and of label environments (mapping labels to continuations). (For more on continuations, see [46] and [45].)

$c \in \text{CONTINUATION} = \text{STORE} \longrightarrow \text{STORE}$

$e \in \text{LABELENV} = \text{LABEL} \longrightarrow \text{CONTINUATION}$

We define a low level semantic function, lsem, mapping low level programs to label environments in which labels denote continuations.

$\text{lsem}:\text{LPROGRAM} \longrightarrow \text{LABELENV}$

Each label is associated, in the label environment returned by lsem, with the continuation representing the meaning of the program from that label to the end of the program. The terminating label is associated with the identity continuation. We need another semantic function, lsemst, to compute the meaning of individual statements in a label environment with a continuation:

$\text{lsemst}:\text{STATEMENT} \longrightarrow \text{LABELENV} \longrightarrow \text{CONTINUATION} \longrightarrow \text{CONTINUATION}$

lsemst is defined by the clauses:

$$\text{lsemst } [\![\underline{\text{assign}}(I,exp)]\!] \; e \; c = \lambda s.\ c(s[eval(exp,s)/I])$$

$$\text{lsemst } [\![\underline{\text{ifnot}}(exp,L)]\!] \; e \; c = \lambda s.\ eval(exp,s) \Rightarrow c\ s \mid e\ L\ s$$

$$\text{lsemst } [\![\underline{\text{goto}}\ L]\!] \; e \; c = e\ L$$

$$\text{lsemst } [\![q]\!] \; e \; c = \lambda s.\ c(\text{lsem } [\![q]\!] \; L1\ s)$$

This is straightforward in all but the last case. If a statement <u>is</u> a program, the label environment for the whole (outer) program is disregarded, and the continuation provided is applied to the meaning (found by applying lsem) of the program which constitutes the statement. This isolates the inner labels as desired. We let q abbreviate a low level program as shown below; then lsem assigns meanings to whole low level programs by constructing label environments as follows, where q = <L1:t1, L2:t2,...,Ln:tn, L(n+1)>.

$$
\begin{aligned}
&\text{lsem}[\![q]\!]= \\
&\bot\,[\text{lsemst}\,[\![t1]\!]\;(\text{lsem}\,[\![q]\!])(\text{lsem}\,[\![q]\!]\;L2)\;/\;L1] \\
&\quad[\text{lsemst}\,[\![t2]\!]\;(\text{lsem}\,[\![q]\!])(\text{lsem}\,[\![q]\!]\;L3)\;/\;L2] \\
&\qquad\vdots \\
&\quad[\text{lsemst}\,[\![tn]\!]\;(\text{lsem}\,[\![q]\!])(\text{lsem}\,[\![q]\!]\;L(n+1))\;/\;Ln] \\
&\quad[(\lambda s.s)\;/\;L(n+1)]
\end{aligned}
$$

The idea is that to each label Li in the program q is bound the meaning (continuation) of the corresponding statement, taken in the label environment of the <u>whole</u> program and with the continuation attached to the <u>next</u> label, beginning with the completely undefined label environment. Since lsem and lsemst are mutually recursive, we use the device, justified by Bekic's theorem [2] of passing along a functional argument to lsemst. (We feel that this is neater than taking a simultaneous fixed point, though that is perhaps the more obvious solution.) Thus, to be correct, lsemst has the type

$$
\begin{aligned}
&\text{lsemst}:\text{STATEMENT}\longrightarrow\text{LABELENV}\longrightarrow\text{CONTINUATION}\longrightarrow \\
&\qquad(\text{LPROGRAM}\longrightarrow\text{LABELENV})\longrightarrow\text{CONTINUATION}
\end{aligned}
$$

and the functional argument, lsem′, say, is the one that is applied to the subprogram in the case of a statement which is a program:

$$\text{lsemst} [\![q]\!] \text{ e c lsem}' = \lambda s.\ c(\text{lsem}' [\![q]\!] \text{ L1 s})$$

lsem passes itself as a functional argument in each of its calls of lsemst. We define lsem as the least fixed point of a functional LSEM, whose definition is obvious.

The statement which expresses the correctness of the compiler is:

Theorem 3.1

$$\forall p.\ \text{hsem} [\![p]\!] \equiv \text{lsem} [\![C(p)]\!] \text{ L1}$$

That is, the meaning of any high level program p is equivalent to the meaning of the compiled version of p (the meaning is a label environment) applied to L1 (L1 is necessarily the first label of any compiled program). Intuitively, (lsem $[\![C(p)]\!]$ L1) is the meaning of the first statement of the compiled program, in the label environment for the whole program, with the continuation for the rest of the program.

The Informal Proof

The proof of Theorem 3.1 which Russell gives, in fact, is incorrect. He attempts to do computation induction on hsem and C, proving the theorem as an equivalence, and unfolding the induction variable for C in the process. The easiest proof we have found is of a pair of inequivalences, by computation induction on the two semantic functions, in turn. The proof can also be done, although it is slightly more complicated, by induction on the structure of high level programs. Although structural induction is more natural,

the proof by it requires an inner computation induction in the while case. In the proof we have generated in LCF and described here, we adhere to Russell's original proof plan as far as possible, and use computation induction. This point is discussed further in Note 3 and in the Conclusions.

The pair of theorems we prove are:

Theorem 3.1a
$\forall p.$ hsem $[\![p]\!] \sqsubseteq$ lsem $[\![C(p)]\!]$ L1

Theorem 3.1b
$\forall p.$ lsem $[\![C(p)]\!]$ L1 $\sqsubseteq$ hsem $[\![p]\!]$

The proof of Theorem 3.1a is by computation induction on hsem. That of Theorem 3.1b is somewhat more complicated; we wish to do induction on lsem, but we also wish to unfold lsem several times in the course of evaluating the left hand side of the formula (to reflect the fact that low level programs have several statements for each high level construct). For example, consider the label environment corresponding to the compiled image of ⌜while exp p1⌝, that is, to the low level program

```
L1: ifnot(exp,L4)
L2: C(p1)
L3: goto L1
L4:
```

which we call q. The label environment constructed by lsem is:

```
⊥[lsemst [[ifnot(exp,L4)]] (lsem [[q]])(lsem [[q]] L2) lsem / L1]
 [lsemst [[C(p1)]]         (lsem [[q]])(lsem [[q]] L3) lsem / L2]
 [lsemst [[goto]]L1        (lsem [[q]])(lsem [[q]] L4) lsem / L3]
 [(λs.s) / L4]
```

To evaluate the application of this whole label environment to L1

(in the expression lsem ⟦q⟧ L1) we have to evaluate lsem ⟦q⟧ L2, for which we need lsem ⟦q⟧ L3; for that we take lsemst ⟦goto L1⟧ (lsem ⟦q⟧)(lsem ⟦q⟧ L4) lsem, which is lsem ⟦q⟧ L1 again. The point is that we have to be able to unfold lsem three times. As a solution, we have formulated a rule of _iterated_ computation induction which unfolds the induction variable a given number of times. This rule is generally useful for proofs by simultaneous induction on functions with different rates of recursion, a situation which naturally arises in compiler proofs. (For other uses of iterated induction, see Note 3, and Chapter 4, p. 167.) To unfold n times, the rule is

$$\begin{array}{l}(w[\bot/f] \ \& \ w[fun \bot/f] \ \& \ \ldots \ \& \ w[fun^{n-1}\bot/f] \ \& \\ \forall f'. \ (w[f'/f] \ \& \ w[fun \ f'/f] \ \& \ \ldots \ \& \ w[fun^{n-1}f'/f]) \supset \\ w[fun^{n}f'/f]) \quad \supset \\ w[FIX \ fun/f]\end{array}$$

This rule is valid because it is just an ordinary induction on f' in the formula

$$w \ \& \ w[fun \ f'/f] \ \& \ \ldots \ \& \ w[fun^{n-1}f'/f]$$

The basis is the basis shown, and the step follows easily from the step shown. The first conjunct is selected from the conclusion, after induction.

To prove _Theorem 3.1b_ using this rule, we let n = 4, and prove four bases and a step with four hypotheses:

BASIS1

$$\perp \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

BASIS2

$$LSEM \; \perp \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

BASIS3

$$LSEM^{2} \; \perp \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

BASIS4

$$LSEM^{3} \; \perp \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

IH1

$$lsem' \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

IH2

$$LSEM \; lsem' \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

IH3

$$LSEM^{2} \; lsem' \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

IH4

$$LSEM^{3} \; lsem' \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

STEP

$$LSEM^{4} \; lsem' \; [\![C(p)]\!] \; L1 \subseteq hsem \; [\![p]\!]$$

The proofs of Theorem 3.1a, the STEP of Theorem 3.1b and the latter three basis cases of Theorem 3.1b (the first basis case is easy) follow similar lines. We therefore present just the proof of the STEP.

We first compute, once for all, the label environments for the various high level constructs. These computations are given in four lemmas:

Lemma 3.2

```
     p = ⌜assign(I,exp)⌝ ⊃
     LSEM^n lsem' ⟦C(p)⟧ =
     ⊥[lsemst  ⟦assign(I,exp)⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L2)
               (LSEM^n-1 lsem')                      / L1]
      [(λs.s)                                        / L2]
```

Lemma 3.3

```
     p = ⌜if exp then p1 else p2⌝ ⊃
     LSEM^n lsem' ⟦C(p)⟧ =
     ⊥[lsemst  ⟦ifnot(exp,L4)⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L2)
               (LSEM^n-1 lsem')                      / L1]
      [lsemst  ⟦C(p)⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L3)
               (LSEM^n-1 lsem')                      / L2]
      [lsemst  ⟦goto L5⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L4)
               (LSEM^n-1 lsem')                      / L3]
      [lsemst  ⟦C(p2)⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L5)
               (LSEM^n-1 lsem')                      / L4]
      [(λs.s)                                        / L5]
```

Lemma 3.4

```
     p = ⌜while exp p1⌝ ⊃
     LSEM^n lsem' ⟦C(p)⟧ =
     ⊥[lsemst  ⟦ifnot(exp,L4)⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L2)
               (LSEM^n-1 lsem')                      / L1]
      [lsemst  ⟦C(p1)⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L3)
               (LSEM^n-1 lsem')                      / L2]
      [lsemst  ⟦goto L1⟧
               (LSEM^n-1 lsem' ⟦C(p)⟧)
               (LSEM^n-1 lsem' ⟦C(p)⟧  L4)
               (LSEM^n-1 lsem')                      / L3]
      [(λs.s)                                        / L4]
```

Lemma 3.5

$$
\begin{array}{ll}
p = \ulcorner p1;p2 \urcorner \supset & \\
LSEM^{n}\ lsem'\ [\![C(p)]\!] = & \\
\bot\ [lsemst\ \ [\![C(p1)]\!] & \\
\quad (LSEM^{n-1} lsem' [\![C(p)]\!]) & \\
\quad (LSEM^{n-1} lsem' [\![C(p)]\!]\ \ L2) & \\
\quad (LSEM^{n-1} lsem') & /\ L1] \\
\ [lsemst\ \ [\![C(p2)]\!] & \\
\quad (LSEM^{n-1} lsem' [\![C(p)]\!]) & \\
\quad (LSEM^{n-1} lsem' [\![C(p)]\!]\ \ L3) & \\
\quad (LSEM^{n-1} lsem') & /\ L2] \\
\ [(\lambda s.s) & /\ L3]
\end{array}
$$

Proof of the STEP of Theorem 3.1b

We assume IH1, IH2, IH3 and IH4.

Show

$$LSEM^{4}\ lsem'\ [\![C(p)]\!]\ \ L1 \subseteq hsem\ [\![p]\!]$$

We consider the cases for the four high level constructs.

Case p = $\ulcorner$assign(I,exp)$\urcorner$

$$RHS = \lambda s.s[eval(exp,s)/I]$$

$$
\begin{array}{ll}
LHS = & lsemst\ \ [\![assign(I,exp)]\!] \\
& (LSEM^{3}\ lsem'\ [\![C(p)]\!]) \\
& (LSEM^{3}\ lsem'\ [\![C(p)]\!]\ \ L2) \\
& (LSEM^{3}\ lsem')
\end{array}
$$

by Lemma 3.2

$$= \lambda s.\ (LSEM^{3}\ lsem'\ [\![C(p)]\!]\ L2)\ (s[eval(exp,s)/I])$$

by unfolding lsemst according to its definition

$$= \lambda s.\ (\lambda s.s)\ (s[eval(exp,s)/I])$$

by applying Lemma 3.2 again

$$= \lambda s.\ s[eval(exp,s)/I]$$

The proofs for the other cases are similar; we give only the while case in detail.

Case p = if exp then p1 else p2

RHS unfolds to

$$\lambda s.\ eval(exp,s) \Rightarrow hsem\ [\![p1]\!]\ s \mid hsem\ [\![p2]\!]\ s$$

After the sequence of unfoldings using Lemma 3.3 and the definition of lsemst, we arrive at

$$
\begin{array}{l}
\lambda s.\ eval(exp,\ s) \Rightarrow LSEM^{2} lsem'\ [\![C(p1)]\!]\ \ L1\ s\ \mid \\
\qquad\qquad LSEM^{2} lsem'\ [\![C(p2)]\!]\ \ L1\ s
\end{array}
$$

for the LHS, and we then use IH3.

Case p = ⌜while exp p1⌝

```
RHS = eval(exp,s) ⇒ hsem ⟦p⟧ (hsem ⟦p1⟧ s) | s
        by unfolding hsem according to its definition.

LHS = lsemst ⟦ ifnot(exp,L4)⟧
             (LSEM ³lsem′ ⟦C(p)⟧)
             (LSEM ³lsem′ ⟦C(p)⟧  L2)
             (LSEM ³lsem′)
        by Lemma 3.4

    = λs. eval(exp,s) ⇒ LSEM ³lsem′ ⟦C(p)⟧  L2 s |
                        LSEM ³lsem′ ⟦C(p)⟧  L4 s
        by unfolding lsemst according to its definition

    = λs. eval(exp,s) ⇒ lsemst  ⟦C(p1)⟧
                                (LSEM ²lsem′ ⟦C(p)⟧)
                                (LSEM ²lsem′ ⟦C(p)⟧  L3)
                                (LSEM ²lsem′) s |
                        s
        by applying Lemma 3.4 twice

    = λs. eval(exp,s) ⇒ (LSEM ²lsem′ ⟦C(p)⟧ L3)
                        (LSEM ²lsem′ ⟦C(p)⟧ L1 s) |
                        s
        by unfolding lsemst

    = λs. eval(exp,s) ⇒ (lsemst ⟦goto L1⟧
                                (LSEM lsem′ ⟦C(p)⟧)
                                (LSEM lsem′ ⟦C(p)⟧  L4)
                                (LSEM lsen′))
                        (LSEM ²lsem′ ⟦C(p1)⟧  L1 s) |
                        s
        by applying Lemma 3.4

    = λs. eval(exp,s) ⇒ (LSEM lsem′ ⟦C(p)⟧  L1)
                        (LSEM ²lsem′ ⟦C(p1)⟧  L1 s) |
                        s
        by unfolding lsemst again.
```

At this point, finally, we can use IH2 and IH3 to complete the proof.

Case p = ⌜p1;p2⌝

```
RHS =  λs. (hsem ⟦p2⟧ )(hsem ⟦p1⟧ s)
```

LHS eventually unfolds, using Lemma 3.5 and the definition of lsemst, to

```
λs. (LSEM ²lsem′ ⟦C(p2)⟧ L1) (LSEM ³lsem′ ⟦C(p1)⟧  L1 s)
```

for which we can use IH3 and IH4. Q.E.D.

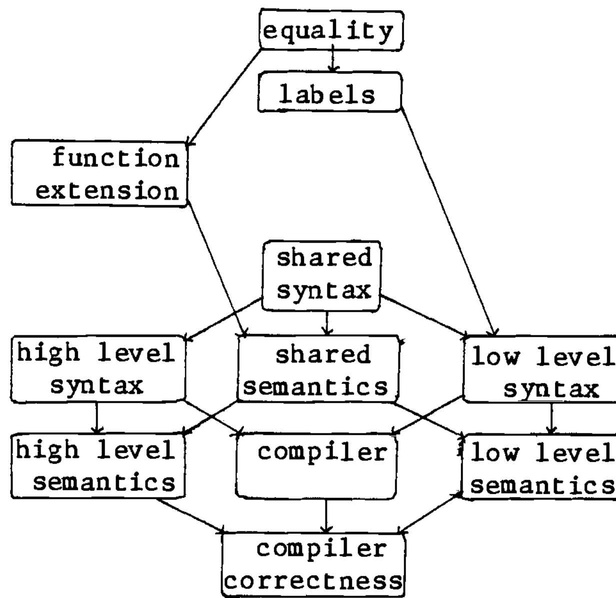
It is clear that the bulk of the proof consists in repeated invocations of the lemmas, and unfoldings of lsemst, in which continuations are used in the specified ways. This applies equally to the proofs of Theorem 3.1a, and BASIS2, BASIS3, and BASIS4 of Theorem 3.1b. Thus, a strategy for generating the proofs is:

> Do induction or iterated induction, and prove for arbitrary p. Divide into cases for the four high level constructs. In each case, unfold the appropriate occurrences of LSEM or lsem by using the lemmas; then unwind lsemst. Do these unfoldings repeatedly until the left hand side equals the right hand side, or until one of the induction hypotheses is applicable.[7]

The Proof in LCF

Theory Structure for the Proof

To organise the new objects and facts required in formalising this problem in LCF, we work within a network of LCF theories. We build a theory of the semantics of both languages, each in turn based on a theory of syntax, since semantic functions operate on syntactic entities. We factor out a theory of the shared syntax and a (daughter) theory of the shared semantics, as the high and low level languages share such types as assignment statements, and the high and low level semantics share objects such as stores. The compiler theory requires both syntax theories as parents, since the compiler theory maps high level to low level programs. The theory in which the correctness of the compiler is stated and proved depends on both semantics theories, as well as the compiler theory, since correctness is the preservation of the semantics under compilation. We factor out the trivial theory of labels as a separate theory (a parent of the low level syntax theory) and give a general polymorphic theory of function extension as a parent of the shared semantics theory, so that we can deal with extensions to stores and to label environments in a uniform way. A polymorphic theory of equality is a parent of both function extension and label theory. [4] The structure of theories for this proof effort is shown below. T1 $\rightarrow$ T2 means that theory T1 is a parent of theory T2.



We outline the main theories below. For brevity, not all new types, constants and axioms are shown, and not all definitions are fully expanded (we resort to ellipsis).

The theory of labels is quite simple. We introduce a new type, LABEL, and five new constants having that type: L1, L2, L3, L4 and L5. We have as axioms or theorems (depending on what can be deduced from the equality axioms):

$\vdash$EQ L1 L1 $\equiv$ TT

$\vdash$EQ L1 L2 $\equiv$ FF

and the like, where EQ is an equality function inherited from equality theory.

The theory of function extension is also simple. We introduce a polymorphic constant for extension:

extend:(* $\rightarrow$ **) $\longrightarrow$ ** $\longrightarrow$ * $\longrightarrow$ (* $\longrightarrow$ **)

and axiomatise it by:

AXEXTEND

$$\text{extend } f \text{ val var } y \equiv (\text{EQ } y \text{ var} \Rightarrow \text{val} \mid y)$$

In the theory of high level syntax, the main new type is HPROGRAM for high level programs. This is a recursive type, so we axiomatise it by introducing a pair of new constants which form an isomorphism between the 'abstract type' HPROGRAM and its representation. The domains ASSIGN, IF, WHILE and COMPOUND, for the four types of programs, are lifted by use of the type operator u, and the coalesced sum is taken, to give us the separated sum we desire.[5]

$$\text{ABSHPROGRAM}:(\text{ASSIGN u} + \text{IF u} + \text{WHILE u} + \text{COMPOUND u}) \longrightarrow \text{HPROGRAM}$$

$$\text{REPHRPOGRAM}:\text{HPROGRAM} \longrightarrow (\text{ASSIGN u} + \text{IF u} + \text{WHILE u} + \text{COMPOUND u})$$

These functions are governed by the axioms

$$\vdash \text{ABSHPROGRAM}(\text{REPHPROGRAM } p) \equiv p$$

$$\vdash \text{REPHPROGRAM}(\text{ABSHPROGRAM } \alpha) \equiv \alpha$$

We define the types for the latter three constructs (assignment will have been introduced in the shared syntax theory: ASSIGN = ID × EXP).

$$\text{IF} = \text{EXP} \times \text{HPROGRAM} \times \text{HPROGRAM}$$

$$\text{WHILE} = \text{EXP} \times \text{HPROGRAM}$$

$$\text{COMPOUND} = \text{HPROGRAM} \times \text{HPROGRAM}$$

We are then able to add constant of the various types, and axioms about them, to supply all of the constructors, destructors and selectors required in the formalisation. For example, we add

constants with the following names and types:

isassign:HPROGRAM $\longrightarrow$ tr

mkassign:ASSIGN $\longrightarrow$ HPROGRAM

destassign:HPROGRAM $\longrightarrow$ ASSIGN

assignidof:ASSIGN $\longrightarrow$ ID

assignexpof:ASSIGN $\longrightarrow$ EXP

and the corresponding new axioms:

$\vdash$ isassign $\equiv$ DOWN o ISL o REPHPROGRAM

$\vdash$ mkassign $\equiv$ ABSHPROGRAM o INL o UP

$\vdash$ destassign $\equiv$ DOWN o OUTL o REPHPROGRAM

$\vdash$ assignidof $\equiv$ FST

$\vdash$ assignexpof $\equiv$ SND

There are, naturally, many more constants and axioms of this sort, for example isif, destif, mkif, iswhile, destwhile, mkwhile, iscompound, destcompound and mkcompound, with the obvious types and definitions.

In the theory of low level syntax, we have similar work to do. We add a new type, LPROGRAM, for low level programs. Corresponding to the syntax equations (p. 94) in which there are two 'loops' (programs consist of sequences of labelled statements followed by a label, and statements, in turn, may be programs) we need four new constants, defining two isomorphisms:

REPLPROGRAM:LPROGRAM $\longrightarrow$ (STATEMENTSEQ $\times$ LABEL)

ABSLPROGRAM:(STATEMENTSEQ $\times$ LABEL) $\longrightarrow$ LPROGRAM

REPSTATEMENTSEQ: STATEMENTSEQ $\longrightarrow$ (LABELLEDSTAT u +
(LABELLEDSTAT $\times$ STATEMENTSEQ) u)

ABSSTATEMENTSEQ: (LABELLEDSTAT u +
(LABELLEDSTAT $\times$ STATEMENTSEQ) u) $\longrightarrow$ STATEMENTSEQ

axiomatised by

$\vdash$ ABSLPROGRAM(REPLPROGRAM q) $\equiv$ q

$\vdash$ REPLPROGRAM(ABSLPROGRAM α) $\equiv$ α

$\vdash$ ABSSTATEMENTSEQ(REPSTATEMENTSEQ s) $\equiv$ s

$\vdash$ REPSTATEMENTSEQ(ABSSTATEMENTSEQ α) $\equiv$ α

We add the other types:

LABELLEDSTAT = LABEL $\times$ STATEMENT

STATEMENT = ASSIGN u + IFNOT u + GOTO u + LPROGRAM u

IFNOT = EXP $\times$ LABEL

GOTO = LABEL

and routine constructors, destructors and selectors, such as

mkLassign:ASSIGN $\longrightarrow$ STATEMENT

isLassign:STATEMENT $\longrightarrow$ tr

destLassign:STATEMENT $\longrightarrow$ ASSIGN

assignLidof:STATEMENT $\longrightarrow$ ID

assignLexpof:STATEMENT $\longrightarrow$ EXP

where, to avoid confusion, `L` indicates that these are low level syntax constants.

Two further constants are

$$ONTO{:}LABELLEDSTAT \longrightarrow STATEMENTSEQ \longrightarrow STATEMENTSEQ$$

to add a labelled statement onto a sequence of labelled statements, and

$$issinglestatement{:}LPROGRAM \longrightarrow tr$$

to determine whether a program consists of <u>exactly</u> one labelled statement. The associated axioms are:

$\vdash mkLassign \equiv INL \circ UP$

$\vdash isLassign \equiv ISL$

$\vdash destLassign \equiv DOWN \circ OUTL$

$\vdash \forall t.\ assignLidof\ t \equiv FST(destLassign\ t)$

$\vdash \forall t.\ assignLexpof\ t \equiv SND(destLassign\ t)$

$\vdash \forall l\ ss.\ ONTO\ l\ ss \equiv ABSSTATEMENTSEQ(INR(UP(l,ss)))$

$\vdash issinglestatement \equiv ISL \circ REPSTATEMENTSEQ \circ FST \circ REPLPROGRAM$

We can then add, for example,

$$firstlabelof{:}LPROGRAM \longrightarrow LABEL$$

to retrieve the first label of a program, where

$$\begin{aligned}\vdash firstlabelof\ q \equiv\ & issinglestatement\ q \Rightarrow \\ & FST(DOWN(OUTL(REPSTATEMENTSEQ(FST \\ & (REPLPROGRAM\ q))))) \mid \\ & FST(FST(DOWN(OUTR(REPSTATEMENTSEQ \\ & (FST(REPLPROGRAM\ q))))))\end{aligned}$$

to fetch the first label of a program, whether it has one or several statements. Again, to construct a program from a single labelled

statement and a terminating label, we specify a constant

```
destlprogram:LABELLEDSTAT ⟶ LABEL ⟶ LPROGRAM
```

and an axiom

```
⊢∀ls l. destlprogram ls l ≡
        ABSLPROGRAM(ABSSTATEMENTSEQ(INL(UPls)),L)
```

A large number of the routine constants (and associated axioms) have to be added, e.g. firststatementof, secondlabelof, restof (for mapping programs to their 'tails'), and lastlabelof, with the meanings suggested by the names.

In the high level semantics theory we add a constant, hsem, for the semantic function, and HSEM for its defining functional:

```
hsem:HPROGRAM ⟶ STORE ⟶ STORE

HSEM:(HPROGRAM ⟶ STORE ⟶ STORE) ⟶
     (HPROGRAM ⟶ STORE ⟶ STORE)
```

and axioms

<u>AXhsem</u>

```
    ⊢hsem ≡ FIX HSEM
```

<u>AXHSEM</u>

```
    ⊢HSEM ≡ λhsem′ p s. isassign p ⇒
                         extend s
                                (eval(assignexpof p,s))
                                (assignidof p) |
                         ...
```

and so on, using the various constants from the high level syntax theory to give the remaining three clauses.

In the same fashion, we create a theory of the low level semantics and add a new type LABELENV

LABELENV = LABEL ⟶ CONTINUATION

where

CONTINUATION = STORE ⟶ STORE

and we add constants for the semantic functions and defining functionals:

```
lsem:LPROGRAM ⟶ LABELENV

LSEM:(LPROGRAM ⟶ LABELENV) ⟶ (LPROGRAM ⟶ LABELENV)

lsemst:STATEMENT ⟶ LABELENV ⟶ CONTINUATION ⟶
        (LPROGRAM ⟶ LABELENV) ⟶ CONTINUATION
```

and another constant

```
createLABELENV:LPROGRAM ⟶ LABELENV ⟶ (LPROGRAM ⟶ LABELENV)
            ⟶ LABELENV
```

with the associated axioms

<u>AXlsem</u>
```
    ⊢ lsem ≡ FIX LSEM
```

<u>AXLSEM</u>
```
    ⊢ LSEM ≡ λlsem′ q. createLABELENV q (lsem′ q) lsem′
```

<u>AXlsemst</u>
```
    ⊢ lsemst t e c lsem′ ≡ isLassign t ⇒
                          λs.c(extend s
                                      (assignLidof t)
                                      (assignLexpof t)) |
                          ...
```

The function createLABELENV is just an intermediate function for

constructing label environments according to the definition of lsem.

Informally,

```
createLABELENV <L1:t1, ..., Ln:tn, L(n+1)> e lsem'
⊥[lsemst ⟦t1⟧ e (e L2) lsem' / L1]
 [lsemst ⟦t2⟧ e (e L3) lsem' / L2]
               .
               .
               .
 [(λs.s) / L(n+1)]
```

so that formally, we we have

<u>AXCLE</u>

```
    ⊢ createLABELENV q e lsem' ≡  issinglestatement p ⇒
      extend(extend  ⊥
                     (lsemst(firststatementof q)
                             e
                             (e(secondlabelof q))
                             lsem')
                      (firstlabelof q))
              (λs.s)
              (lastlabelof q) |
       createLABELENV (restof q)
                      (extend ⊥
                              (lsemst (firststatementof q)
                                      e
                                      (e(secondlabelof q))
                                      lsem')
                              (firstlabelof q))
                      lsem'
```

Finally, the compiler theory; we introduce a new constant, C, for the compiling function:

```
C:HPROGRAM ⟶ LPROGRAM
```

and define it by the axiom

<u>AXC</u>

```
    ⊢ C p ≡ isassign  p ⇒ ABSLPROGRAM(ABSSTATEMENTSEQ(L1,
                          mkLassign(destassign p)),L2) |
                          ...
```

and so on for the other clauses (stringing them together using ONTO).

The correctness theory requires no new types or axioms, but we save in it many new theorems. It inherits from its ancestors all of the new types, constants and axioms defined thus far.

This covers the main points of the structure of theories in which the correctness of the Russell compiler is stated and proved.

Lemma Structure for the Proof

The first stage, in generating the proof in LCF, consists in proving some simple lemmas, and then some more difficult ones. The reasons for proving the lemmas at the outset instead of allowing simplification to take its course during the main proof are twofold. Firstly, the lemmas would have to be reproved many times during the main proof, so efficiency is achieved by proving them once and storing them as facts in the appropriate theories. Also, some of the lemmas have fairly large simplification sets; we can reduce the number of simprules in the simpset of the main goal by dispensing with the simplifications required only for the lemmas.

Secondly, it will become clear that the challenge in managing a proof of this complexity is to leave as much as possible to simplification. After the user constructs successive layers of carefully chosen lemmas, the main proofs can be performed with a minimum of user guidance. The alternative is to guide the proof by a sequence of tactics which unfold and substitute in exactly the correct ways. This is both more tedious and less illuminating than

constructing layers of lemmas.

The first group of (syntactic) lemmas are very routine. We prove, for example, in the theory of low level syntax, that

$\vdash \forall$ a. isLassign(mkLassign a) $\equiv$ TT

For all of the (similar) lemmas in this set, a simpset composed of BASICSS and simprules formed from all of the basic low level syntax axioms is used, and all are proved by an application of SIMPTAC. We save the new theorems in the low level syntax theory. We then form a simpset from all of the new theorems, and, for reference, call it SSLLSYNT (for simpset for low level syntax).

In the compiler theory, we prove another set of syntactic theorems, relating the high and low level languages. These are useful since the compiling function builds low level programs from fragments of high level programs. We prove, for example

$\vdash \forall$p. assignLidof(mkLassign(destassign p)) $\equiv$
    assignidof(destassign p)

The various lemmas in this group share a simpset including all of the basic syntax axioms, high and low level. SIMPTAC solves all of the goals. We call the simpset formed from the resulting theorems SSCOMP (for compiler theory simpset).

It is also useful to prove the following lemmas, the first two in the high and the second two in the low level semantics theories.

<u>thhsem</u>

$\vdash$ hsem p $\equiv$ λs. isassign p $\Rightarrow$ extend s
                                        (eval(assignexpof p,s))
                                        (assignidof p) |
                                         ...

thHSEM

$\vdash$HSEM hsem′ p ≡ isassign p ⇒ ... | ...

thlsem

$\vdash$lsem q ≡ createLABELENV q (lsem q) lsem

thLSEM

$\vdash$LSEM lsem′ q ≡ createLABELENV q (lsem′ q) lsem′

The first and third are easily proved using UNFOLDTAC on AXhsem and AXlsem, respectively (see Chapter 2, p. 64). The second and fourth are proved by simplification, with AXHSEM and AXLSEM in the simpsets of the goals.

From thLSEM and thlsem it is easy to prove the following eight lemmas, where the bracket indicates a choice, which must be the same in both instances:

$$\text{lsem } (C(p)) \begin{Bmatrix} L2 \\ L3 \\ L4 \\ L5 \end{Bmatrix} \equiv \text{createLABELENV}(C(p))(\text{lsem } (C(p)))\text{lsem} \begin{Bmatrix} L2 \\ L3 \\ L4 \\ L5 \end{Bmatrix}$$

$$\text{LSEM lsem}'(C(p\ )) \begin{Bmatrix} L2 \\ L3 \\ L4 \\ L5 \end{Bmatrix} \equiv \text{createLABELENV}(C(p))(\text{lsem}'(C(p)))\text{lsem}' \begin{Bmatrix} L2 \\ L3 \\ L4 \\ L5 \end{Bmatrix}$$

Used as simplifications, these lemmas allow us to select the contexts in which LSEM and lsem are unfolded. In particular, they allow us to avoid unfolding similar expressions involving L1, as these are to be viewed as instances of the various induction hypotheses. (This point is discussed again presently.) We form a simpset from the eight lemmas, called SSLSEMlsem.

Next, we prove, in the compiler correctness theory, four lemmas corresponding to the lemmas which constructed label environments in the informal proof (Lemma 3.2, Lemma 3.3, Lemma 3.4, and Lemma 3.5). We use the intermediate function createLABELENV, here. We prove, for example,

```
|-∀p. isassign p ≡ TT IMP
      createLABELENV (C(p)) e lsem' ≡
      extend(extend ⊥
                    (lsemst(mkLassign(assignidof p,
                                      assignexpof p))
                            e
                            (e L2)
                            lsem')
                    L1)
             (λ s.s)
             L2
```

the goal for which is

createLABELENV (C(p)) e lsem' ≡ ...
SSLLSYNT + BASICSS + (isassign p ≡ TT)
isassign p ≡ TT

where the right hand side of the formula is as above. There are three similar lemmas (and goals) for the other cases. The goals are all proved by using the standard tactic SUBSTAC:thm list→ tactic

SUBSTAC [|-ti ≡ ui]

w[ti/xi]
ss
A

=====

w[ui/xi]
ss
A

in order to unfold the definitions of C and createLABELENV. The tactic which solves the four goals is

(SUBSTAC [AXC])+
((SUBSTAC [AXCLE])+)*

We can include neither AXC nor AXCLE in the simpsets of the goals, as both C and createLABELENV are recursively defined and would therefore loop, as simplification rules. Instead, we make explicit substitutions. The resulting theorems may safely be used as simprules in the main proof. If we do not prove the four lemmas in advance, we would have to make the explicit substitutions during the main proof, whereas by using the lemmas, the substitutions are done automatically. In addition, the lemmas accomplish once a segment of proof which would otherwise have to be performed many times in the course of the main proof.

We call the lemmas CLEa, CLEi, CLEw and CLEc, and the simpset containing the four of them, SSCLE.

In the same spirit, ten more lemmas, also in the compiler correctness theory, can be proved from the above four, from goals of the form

createLABELENV ((C)p)) e lsem′ L2 ≡ (λs.s)
AXEXTEND + (isassign p ≡ TT) + SSLABEL + SSCLE + BASICSS
isassign p ≡ TT

where SSLABEL is a simpset containing the basic axioms and theorems about labels. There are nine more similar lemmas about the environments for the other constructs, applied to the other labels (but again, not L1). These are all proved by simplification. We call the simpset formed from the ten resulting theorems SSCLEL (simpset for creating label environments applied to labels).

Several lemmas aid in the proof of the basis cases of Theorem 3.1b; for example, one achieving the goal

∀t. lsemst t ⊥⊥ ≡ lsem′
AXLSEMST + BASICSS

which we solve by applying

```
GENTAC
(CONDCASESTAC+)*
(EXTTAC*)+
```

to prove

thlsemst⊥
⊢∀lsem′ t. lsemst t ⊥⊥ lsem′ ≡ ⊥

We also prove four lemmas, constructing label environments given ⊥:LABELENV as a parameter, from goals of the form

createLABELENV (C(p)) ⊥ lsem′ ≡ extend(extend ⊥ (λs.s) L2)⊥L1
thlsemst⊥ + SSCLE + BASICSS
isassign p ≡ TT

and similarly for the other cases. The four resulting theorems are placed in a simpset called SSCLE . To prove them, we could write a tactic to find a term p of type HPROGRAM in the formula part of a goal and produce the subgoals shown below:

HCASESTAC

$$\frac{(w,\ ss,\ A)}{\begin{array}{|l|}\hline w \\ \hline ss + (p \equiv \bot) \\ \hline (p \equiv \bot) \\ A \\ \hline \end{array} \quad \begin{array}{|l|}\hline w \\ \hline ss + (\text{isassign } p \equiv TT) \\ \hline (\text{isassign } p \equiv TT) \\ A \\ \hline \end{array} \quad \begin{array}{|l|}\hline w \\ \hline ss + (\text{isif } p \equiv TT) \\ (\text{isif } p \equiv TT) \\ \hline A \\ \hline \end{array} \quad \begin{array}{|l|}\hline w \\ \hline ss + (\text{iswhile } p \equiv TT) \\ (\text{iswhile } p \equiv TT) \\ \hline A \\ \hline \end{array} \quad \begin{array}{|l|}\hline w \\ \hline ss + (\text{iscompound } p \equiv TT) \\ (\text{iscompound } p \equiv TT) \\ \hline A \\ \hline \end{array}}$$

That is, HCASESTAC, for high level cases tactic, produces the four cases corresponding to the four types of high level programs, as well as the case for undefined programs.[6] The proof function of HCASESTAC would use a cases rule, HCASES, say, derived from the standard CASES. (For an example of this sort of derivation, see [15], A1.) We would need the following axiom, in the derivation:

$$\begin{array}{lll} \vdash \forall p.\ p \equiv & \text{isassign } p & \Rightarrow \text{mkassign(destassign } p) \mid \\ & \text{isif } p & \Rightarrow \text{mkif(destif } p) \mid \\ & \text{iswhile } p & \Rightarrow \text{mkwhile(destwhile } p) \mid \\ & \text{iscompound } p & \Rightarrow \text{mkcompound(destcompound } p) \mid \bot \end{array}$$

At any rate, we prove the four lemmas using HCASESTAC. The four lemmas in SSCLE must be used as simplifications. We form a simpset of the four theorems thus proved, called SSCLE⊥.

We also prove a layer of lemmas from four goals of the form

```
createLABELENV (C(p)) ⊥ lsem′ L1 ≡ ⊥
SSCLE⊥ + AXEXTEND + SSLABEL + BASICSS
isassign p ≡ TT
```

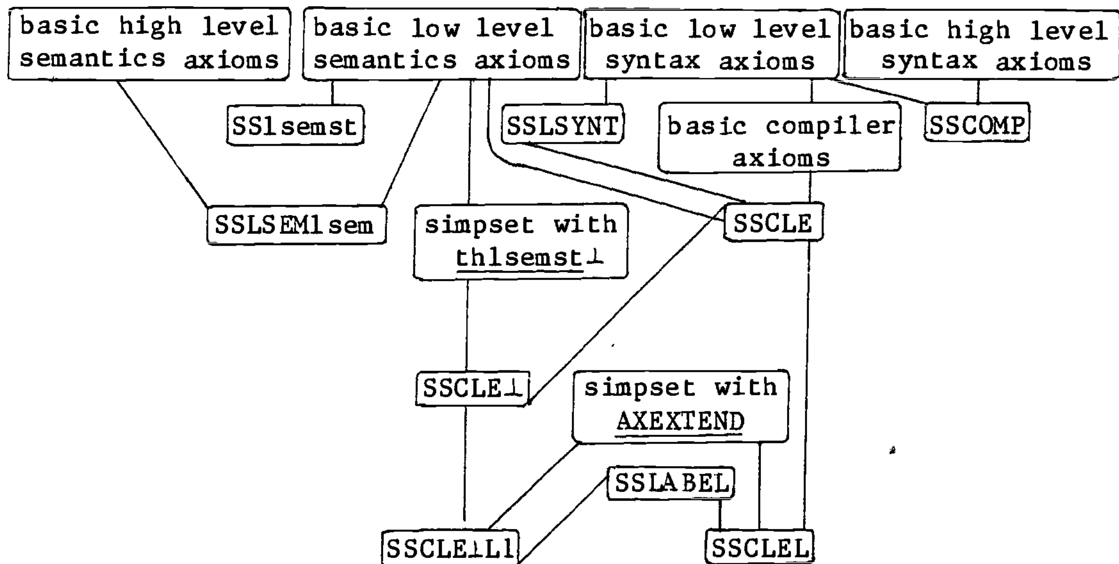
and so on, by applying SIMPTAC. This produces four theorems which we place in a simpset called SSCLE⊥L1.

We finally prove four lemmas about lsemst, analogous to CLEa, etc., from goals of the form

```
lsemst t e c lsem′ ≡ λs. c(extend s
                                  (eval(assignLexpof t,s))
                                  (assignLidof t))
AXlsemst + BASICSS
isassign t ≡ TT
```

by applying SIMPTAC. We put the four resulting theorems in a simpset called SSlsemst. As for the analogous theorems constructing label environments, these four, used as simprules, save unfolding the definition of lsemst many times (and each time simplifying the result).

The logical dependencies amongst the simpsets (representing groups of lemmas) are shown in the tree below:



where the theorems and simpsets named are as follows (to summarise):

SSlsemst is used to unfold lsemst in the various cases;

SSLLSYNT contains simple facts about low level syntax, and SSCOMP, about the relation of high to low level syntax;

SSLSEMlsem contains the eight theorems which unfold LSEM and lsem in the appropriate contexts;

thlsemst⊥ is a theorem, used in the basis cases, for unfolding lsemst with undefined label environments and continuations;

AXEXTEND is the axiom defining function extension, used for label environments and stores;

SSLABEL contains the basic facts about the equality and inequality of the five labels;

SSCLE contains the four basic theorems used for unfolding createLABELENV from which we construct the rest, such as

SSCLEL, which contains the theorems applying the various label environments to the various labels;

SSCLE⊥, for the basis case label environments;

SSCLE⊥L1, for the basis case label environments applied to L1.

The Machine Proof

The beginning and end of the tactical proof of Theorem 3.1 are examined first. At the beginning, we are proving

$$\forall p.\ \text{hsem}\ [\![p]\!] \equiv \text{lsem}\ [\![C(p)]\!] \qquad \text{L1}$$

so we apply SYNTHTAC (Chapter 2, P. 61) to a goal with that formula to obtain two subgoals; the subgoals are achieved by Theorem 3.1a and Theorem 3.1b, respectively. We leave aside the question of simplification sets, for the moment, and begin generating the proofs by applying tactics to do the inductions and specifications to arbitrary variables. For proving Theorem 3.1a, the tactic begins:

```
(INDUCTAC [AXhsem])+
GENTAC
```

SIMPTAC (denoted by the +) solves the basis case produced by INDUCTAC.

For managing the proof of Theorem 3.1b, we require a tactic (ITINDUCTAC) which inverts the rule of iterated induction mentioned on p. 100. The rule, ITINDUCT, and the tactic, must be implemented in ML, ITINDUCT in terms of INDUCT, and ITINDUCT calling INDUCT, first constructing a new basis and step (as discussed), and then selecting the first conjunct of the theorem proved by INDUCT. This is a simple example of the derivation of a rule of induction in LCF.

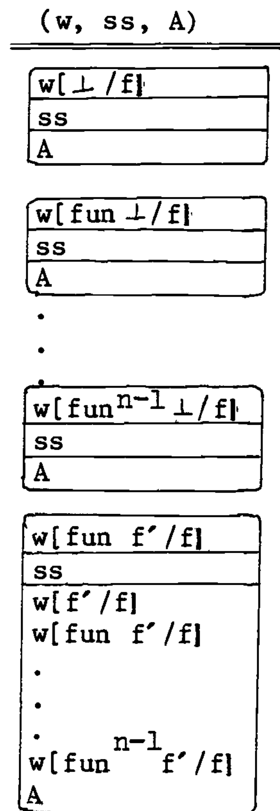
ITINDUCTAC has ML type thm $\longrightarrow$ int $\longrightarrow$ tactic, where the integer represents the number of iterations desired, and the theorem is the least fixed point definition of the function on which induction is being done. Applied to a goal, the tactic returns n

subgoals: n basis subgoals, and a step subgoal which has n induction hypotheses in its assumption list. The proof part of the tactic calls ITINDUCT; ITINDUCT expects n theorems (achieving the subgoals). The new rule and tactic can be depicted as:

ITINDUCT (fun,f)

$$\frac{\begin{array}{l}\vdash w[\perp/f] \\ \vdash w[fun\ \perp/f] \\ \quad\vdots \\ \vdash w[fun^{n-1}\perp/f] \\ \quad [w;\ w[fun\ f/f];\ \ldots\ ;w[fun^{n-1}f/f]]\ \vdash\ w[fun^{n}f/f]\end{array}}{\vdash w[FIX\ fun/f]}$$

ITINDUCTAC ($\vdash$ f ≡ FIX fun) n



(More details on the derivations of the rule and tactic are given in the Appendix.)

We begin the proof of Theorem 3.1b, therefore, by applying

```
(ITINDUCTAC AX1sem 4)+
GENTAC
```

SIMPTAC solves the first of the three basis cases for us.

In both proofs, we would then like to compute the label environment that is constructed for the whole compiled program and apply it to L1; that is, to evaluate, respectively,

$$\text{lsem } [\![C(p)]\!] \text{ L1}$$
$$\text{LSEM}^4 \text{ lsem}' [\![C(p)]\!] \text{ L1}$$

by using the facts AX1sem and AXLSEM, respectively. However, we cannot add these facts to the simpsets of the goals for Theorem 3.1a or Theorem 3.1b, because we wish to regard all subsequent occurences of formulae of the forms

$$\text{LSEM}^n \text{ lsem}' [\![C(p)]\!] \text{ L1} \sqsubseteq \text{hsem } [\![p]\!]$$
$$\text{lsem } [\![C(p)]\!] \text{ L1} \sqsubseteq \text{hsem } [\![p]\!]$$
$$\text{hsem } [\![p]\!] \sqsubseteq \text{lsem } [\![C(p)]\!] \text{ L1}$$

as instances of the various induction hypotheses, not to be further simplified. In the while case (of either proof), in fact, we eventually arrive at a subgoal whose formula part matches the one of above formulae exactly, so there is no way to distinguish the formula part of the current subgoal (i.e. the subgoal we have after induction, simplification and specification) from formulae which

occur in subsequent subgoals. We wish to simplify the former, but not the latter.

The solution we adopt is to write a tactic, called TEMPSIMPTAC (for temporary simplification tactic), of type thm ⟶ tactic, whose effect on a theorem and a goal is to add the theorem to the simpset of the goal, simplify, and return a subgoal having the resulting formula but with the original simpset. TEMPSIMPTAC thus temporarily uses a theorem as a simprule. In this case, use of TEMPSIMPTAC precludes unwanted simplifications of (LSEM ...), (lsem ...) and (hsem ...) which arise later in the course of the proofs, while allowing the simplifications at the outset.

The generation of the proofs of Theorem 3.1a and Theorem 3.1b begin, respectively, with the applications of

```
(INDUCTAC [AXhsem])+
GENTAC
TEMPSIMPTAC thlsem
HCASESTAC+
```

```
(ITINDUCTAC AXlsem 4)+
GENTAC
TEMPSIMPTAC AXLSEM
TEMPSIMPTAC thhsem
HCASESTAC+
```

so that in each proof, after simplification, we have four remaining subgoals.

After all of the unfolding that precedes the use of the induction hypotheses, some further reasoning is required to complete the proofs. In the informal proof of Theorem 3.1b, for example, the formulae of the subgoals in the three cases (the assignment case is solved by this point) are:

$$\lambda s.\ \mathrm{eval}(exp,s) \Rightarrow \mathrm{LSEM}^2\ lsem'\ [\![C(p)]\!]\ \ L1\ s\ |\ \mathrm{LSEM}^2\ lsem'\ [\![C(p)]\!]\ \ L1\ s \ \subseteq$$
$$\lambda s.\ \mathrm{eval}(exp,s) \Rightarrow \mathrm{hsem}\ [\![p1]\!]\ s\ |\ \mathrm{hsem}\ [\![p2]\!]\ s$$

$$\lambda s.\ \mathrm{eval}(exp,s) \Rightarrow (\mathrm{LSEM}\ lsem'\ [\![C(p)]\!]\ \ L1)\ (\mathrm{LSEM}^2\ lsem'\ [\![C(p1)]\!]\ L1\ s)\ |\ s \ \subseteq$$
$$\lambda s.\ \mathrm{eval}(exp,s) \Rightarrow (\mathrm{hsem}\ [\![p]\!])(\mathrm{hsem}\ [\![p1]\!]\ s)\ |\ s$$

$$\lambda s.\ \mathrm{LSEM}^2\ lsem'\ [\![C(p2)]\!]\ \ L1\ (\mathrm{LSEM}^3\ lsem'\ [\![C(p1)]\!]\ s) \ \subseteq$$
$$\lambda s.\ \mathrm{hsem}\ [\![p2]\!]\ (\mathrm{hsem}\ [\![p1]\!]\ s)$$

A general tactic which expects subgoals having formulae of these forms (and the similar ones which occur in the proof of Theorem 3.1 a and the three basis cases) is built from the following two derived tactics:

LAMGENTAC

$\lambda x.\ t1 \subseteq \lambda x.\ t2$
ss
A

$t1 \subseteq t2$
ss
A

COMBTAC

$t1\ u1 \subseteq t2\ u2$
ss
A

$t1 \subseteq t2$		$u1 \subseteq u2$
ss		ss
A		A

The proofs use basic PPLAMBDA inference rules about abstraction and monotonicity.

The general tactic (which we call ENDTAC) which performs the last segment of the proofs is

```
REPEAT(USEASSUMPCHOOSETAC ORELSE ((COMBTAC ORELSE LAMGENTAC
                                   ORELSE CONDCASESTAC) THEN
                                   SIMPTAC))
```

(USEASSUMPCHOOSETAC is described in Chapter 2, p. 83.) ENDTAC tries to apply one of the induction hypotheses, and failing that, tries the tactics COMBTAC, LAMGENTAC and CONDCASESTAC in succession -- then simplifies, and repeats if necessary. If subgoals with formulae of other `shape` were expected, we could add tactics to deal with those shapes (but they are not).

The proofs are completed by adding a middle segment: just a simplification guided by the carefully planned structure of lemmas we have described. The simpset of the goals for both of our theorems is comprised of SSLSEMlsem, SSlemst, SSCLEL and SSCOMP. The proofs of the basis cases require the simpset SSCLELL1, so we also include this in the simpset of the main goals.

The basis cases are proved similarly, by first using AXLSEM and thhsem as simplification rules temporarily, then simplifying, and then using ENDTAC:

```
TEMPSIMPTAC AXLSEM
TEMPSIMPTAC thhsem
ENDTAC
```

As a refinement of the proofs, we observe that it is desirable to unfold occurrences of lsemst before simplifying expressions of the form (LSEM ...) or (lsem ...), to avoid unnecessary expansion. For example, in the course of proving the while case of Theorem

3.1b, a subgoal arises whose formula's left hand side is

```
λs. eval(exp,s) ⇒
(lsemst ⟦goto L1⟧
        (LSEM lsem'  ⟦C(p)⟧)
        (LSEM lsem'  ⟦C(p)⟧  L4)
        (LSEM lsem'))
(LSEM ²lsem'  ⟦C(p)⟧  L1 s) | s
```

The continuation (LSEM lsem' ⟦C(p)⟧ L4) need not actually be evaluated because the semantic function lsemst ignores it. However, ordering of simplification rules is not an option in LCF, and the extra unfolding, if it occurs, does not upset the proof.

Had the lemma structure not been constructed in advance and used to form the simpset for the two main goals, the tactical proof would have had to be guided by successive substitutions. AXC and AXCLE are not suitable theorems to be used as simplification rules, as we have mentioned, as they would obviously loop. Nor are thhsem or thlsem, for the same reason. All of these facts, however, are hypotheses of theorems which are suitable as simprules. Guiding the proof by substitutions requires the user to be aware of the detailed course of the proof, and makes for clumsy, non-transparent tactics. It requires careful indication of the instances (of LSEM, for example) to be unfolded, and careful specification of any quantified theorems to be used as substitutions. In addition, a proof performed in this fashion entails evaluating the same expressions repeatedly.

To summarise, the tactics which solve the two main theorems, collectively called COMPILERTAC, are (respectively):

```
(INDUCTAC [AXhsem])+
GENTAC
TEMPSIMPTAC thlsem
HCASESTAC+
ENDTAC

(ITINDUCTAC AXlsem 4)+
GENTAC
TEMPSIMPTAC AXLSEM
TEMPSIMPTAC thhsem
 ┌──────────┬──────────┬──────────────┐
ENDTAC     ENDTAC     ENDTAC         HCASESTAC+
                                     ENDTAC
```

Conclusions

In this chapter, we have demonstrated the importance of formalising and machine-checking proofs. The logical error in Russell's proof was subtle enough only to be discovered under the constraints of machine-formulation.

More generally, we have shown how a large formal proof has been organised and performed in LCF. Much of the effort was invested in delineating the required theories and in developing a hierarchy of lemmas, each layer forming simpsets for the goals representing the next layer. The resulting lemmas were used as simprules in the main proof, so that as far as possible, the proof is guided by simplification. Beyond this, the control structure for the proof is provided by the use of composite tactics (built using tacticals) which reflect the structure of the informal proof. The tactics themselves are not startling, but the accomplishment of a formal, machine proof of this complexity and magnitude, by the application of high level procedures, is encouraging.

The expression of the problem depends on the theory facility of LCF, which allow new types, new objects of those types, and new axioms about those objects, to be introduced in theories, and theories to be joined in hierarchies. The success of the proof effort rests on the availability of a high level programming language, ML (and its interface, via its abstract type system) to PPLAMBDA, in which strategies for generating proofs can be implemented. It also rests on the power of the simplifier in LCF, which, as we have shown, enables rountine inferences to be done automatically, as a matter of course, and can also be used for automating more advanced proof steps.

The interest of tactical proof lies in (i) the way in which complete, formal proofs can be performed at a high level, (ii) the way in which tactics naturally reflect informal proof plans, (iii) the way in which a tactic that solves a goal abstracts the formal proof in an intelligible form, and (iv) the way in which tactics reflect patterns of inference common to other proofs, and therefore may be helpful in proving other theorems. In particular, one would hope that other compiler proofs would yield to similar tactics. In Chapter 4, we investigate the extent to which this is so.

Notes for Chapter 3

1. An obvious extension to this problem would be the specification of a lower level language without blocks of this sort. A compiler from the current low level language to the new one would 'flatten' blocks by generating unique, new label names, and repairing the labels in all go-to statements.

2. That is, as a solution to the mutually recursive equations

$$lsem = F(lsem, lsemst)$$

$$lsemst = G(lsem, lsemst)$$

where $F = \lambda(lsem', lsemst')\ q.\ \perp[lsemst\ \ldots\ (lsem'\ [\![q]\!])\ (lsem'\ [\![q]\!]\ L2)/L1]\ldots[(\lambda s.s)/(\ldots)]$ and $G = lsem'\ t\ e\ c\ s.\ t = \ulcorner assign(I,exp)\urcorner \Rightarrow \ldots \mid \ldots \mid t = \ulcorner q \urcorner \Rightarrow c(lsem'\ [\![q]\!]\ L1\ s)$ we propose the pair

$$(l\overline{sem}, l\overline{semst}) = F(l\overline{sem}, l\overline{semst}), G(l\overline{sem}, l\overline{semst}))$$

$$= FIX(\lambda P.\ F\ P, G\ P)$$

where the general theorem being used is:

$$\text{For } x{:}*,\ y{:}**,\ F'{:}(* \times **) \longrightarrow *,\ G'{:}(* \times **) \longrightarrow **,\ P{:}* \times **,$$

$$FIX(\lambda P.\ F'\ P, G'\ P) = (FIX(\lambda x.\ F'(x,\hat{y})), \hat{y})$$

$$\text{where } \hat{y} = FIX(\lambda y.\ G'(FIX(\lambda x.\ F'(x,y)), y)$$

$$= G'(FIX(\lambda x.\ F'(x,\hat{y})), \hat{y})$$

In this case, as lsemst contains no recursive calls to itself, G has a first argument of the type of lsem rather than the type of the pair (lsem,lsemst), so the solution lsemst is G(lsem), which is

$$\lambda t\ e\ c\ s.\ \ldots \mid t = \ulcorner q \urcorner \Rightarrow c(l\overline{sem}\ [\![q]\!]\ L1\ s)$$

3. We remark briefly, here, on the slightly more complicated alternative proof by structural induction. It requires reformulating the semantics hsem by using the By-law (Chapter 2, p. 58-59) so that we take a <u>local</u> fixed point in the while case:

$$hsem\ [\![\underline{while}\ exp\ p1]\!] = FIX(\lambda h\ s.\ eval(exp, s) \Rightarrow h(hsem\ [\![p1]\!]\ s) \mid s)$$

$$= FIX\ \mathcal{H}$$

We assume that for p1 and p2, lsem $[\![C(p1)]\!]$ L1 = hsem $[\![p1]\!]$, and lsem $[\![C(p2)]\!]$ L1 = hsem $[\![p2]\!]$. The basis and assignment cases

are obvious. We then consider the remaining three cases. If p = ⌜while exp p1⌝, we must show

$$\lambda s.\ eval(exp,s) \Rightarrow lsem\ [\![C(p)]\!]\ L1\ (lsem\ [\![C(p1)]\!]\ L1\ s)\ |\ s$$

$$\equiv\ FIX\ \mathcal{H}$$

The RHS = $\lambda s.\ eval(exp,s) \Rightarrow (FIX\ \mathcal{H})(hsem\ [\![p1]\!]\ s)\ |\ s$. The induction hypothesis applies and it remains to show

$$lsem\ [\![C(p)]\!]\ L1\ =\ FIX\ \mathcal{H}$$

We achieve this by proving

(i) $FIX\ \mathcal{H} \sqsubseteq lsem\ [\![C(p)]\!]\ L1$

(ii) $lsem\ [\![C(p)]\!]\ L1 \sqsubseteq FIX\ \mathcal{H}$

For (i), we show that ($lsem\ [\![C(p)]\!]\ L1$) is a fixed point of $\mathcal{H}$, that is,

$$\mathcal{H}\ (lsem\ [\![C(p)]\!]\ L1)\ =\ lsem\ [\![C(p)]\!]\ L1$$

RHS = $\lambda s.\ eval(exp,\ s) \Rightarrow$
$(lsem\ [\![C(p)]\!]\ L1)(lsem\ [\![C(p)]\!]\ L1\ s)\ |\ s$

LHS = $\lambda s.\ eval(exp,\ s) \Rightarrow$
$(lsem\ [\![C(p)]\!]\ L1)\ (hsem\ [\![p1]\!]\ s)\ |\ s$

and the result follows by hypothesis. This establishes that for always-terminating programs, the compiler is correct. For (ii) we do an inner iterated (3-ary) computation induction on lsem, proving the following conjunction (recalling that lsem = FIX LSEM):

$$lsem\ [\![C(p)]\!]\ L1\ \sqsubseteq\ FIX\ \mathcal{H}\quad \&\quad lsem\ \sqsubseteq\ FIX\ LSEM$$

(For other instances and discussion of this method of proof, see Chapter 4, Proof of Lemma 4.5 (ii), Lemma 4.10 (i), and Chapter 4, Notes 3 and 5.) We assume that

$$lsem'\ [\![C(p)]\!]\ L1 \sqsubseteq FIX\ \mathcal{H}\ \&\ lsem' \sqsubseteq FIX\ LSEM$$

and

$$LSEM\ lsem'\ [\![C(p)]\!]\ L1 \sqsubseteq FIX\ \mathcal{H}\ \&\ LSEM\ lsem' \sqsubseteq FIX\ LSEM$$

and

$$LSEM^2 lsem'\ [\![C(p)]\!]\ L1 \sqsubseteq FIX\ \mathcal{H}\ \&\ LSEM^2 lsem' \sqsubseteq FIX\ LSEM$$

and we show that (after unfolding LSEM and lsemst):

λs. eval(exp, s) $\Rightarrow$
(LSEM lsem′ ⟦C(p)⟧ L1)(LSEM ^{2}lsem′ ⟦C(p)⟧ L1 s) | s
$\sqsubseteq$ FIX $\mathcal{H}$

The RHS =

$\mathcal{H}$ FIX $\mathcal{H}$ = λs.eval(exp,s) $\Rightarrow$ (FIX $\mathcal{H}$)(hsem ⟦p1⟧ s) | s

The result follows by use of the outer hypothesis, and both conjuncts of various inner hypotheses. The other cases are straightforward. Q.E.D.

To compare the two methods of proof:

(i) In the computation induction proof, we prove, at the top level, a pair of inequivalences; in the structural induction proof, we prove an equivalence. In the former, one direction requires iterated induction.

(ii) The computation induction proof requires a semantics with a global fixed point; the structural induction proof, one with a local fixed point for while statements.

(iii) In the computation induction proof, the induction hypothesis is sufficient for proving the while case. In the structural induction proof, an inner computation induction is required, in two directions. One direction requires iterated induction, as in the computation induction proof at the top level, in conjunction with another formula.

This comparison is discussed further in the Conclusions.

To perform the proof by structural induction, we would implement the following rule and tactic in ML:

HINDUCT

$\vdash$w[$\perp$/p]

$\vdash$ w[<u>assign</u>(I,exp)/p]

[w[p1/p]; w[p2/p]] $\vdash$ w[<u>if</u> exp <u>then</u> p1 <u>else</u> p2/p]

[w[p1/p]] $\vdash$ w[<u>while</u> exp p1/p]

[w[p1/p]; w[p2/p]] $\vdash$ w[p1;p2/p]

$\vdash\forall$p. w

HINDUCTAC

(w, ss, A)

---

w[⊥/p]
ss
A

w[<u>assign</u>(I,exp)/p]
ss
A

w[<u>if</u> exp <u>then</u> p1 <u>else</u> p2/p]
ss
w[p1/p] w[p2/p] A

w[<u>while</u> exp p1/p]
ss
w[p1/p] A

w[p1;p2/p]	
ss	
w[p1/p]	w[p2/p]
A	

4. A polymorphic theory of equality is inconsistent in general. It is sufficient to restrict the theory to flat domains, that is domains in which for all x and y

$$x \sqsubseteq y \supset (x \equiv y \vee x \equiv \bot)$$

These are, in any case, the only domains on which we require equality. All axioms of the theory are conditionalised on the flatness of the domain in question. We therefore add to the theory of labels (for example) an axiom representing flatness:

$$\vdash \forall Li\ Lj.\ EQ\ Li\ Lj \equiv FF\ \ \&\ \ Li \sqsubseteq Lj\ \ IMP\ \ Li \equiv \bot$$

or

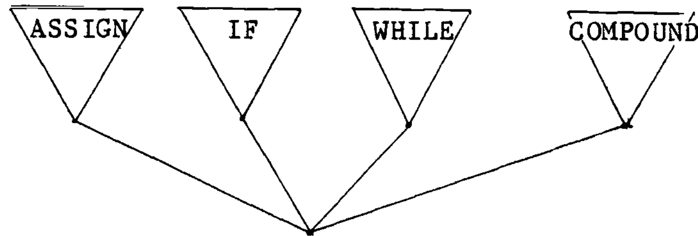
$$\vdash \forall Li\ Lj.\ Li \sqsubseteq Lj\ IMP\ Li \equiv (DEF\ Li \Rightarrow Lj \mid \bot)$$

and use the axiom to discharge the antecedents of any of the conditionalised equality axioms we wish to use. Similarly, the theory of function extension requires the theory of equality, so all extension axioms (e.g. <u>AXEXTEND</u>) must be conditionalised on the flatness of the domain in which they are to be instantiated, that is, to the type of x in

extend f val x

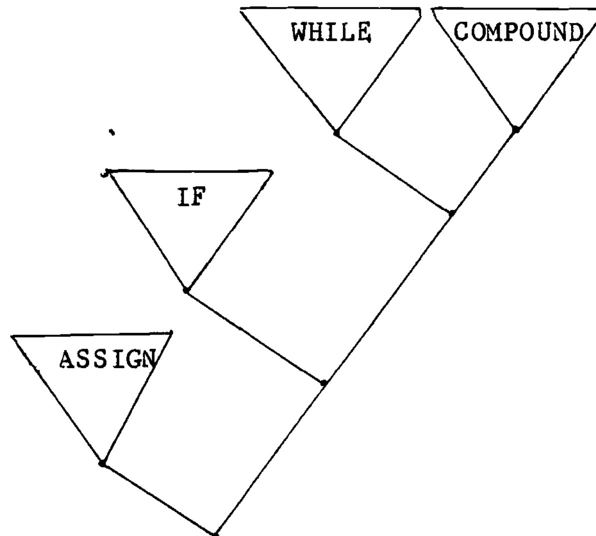
Therefore, we add to the shared semantics theory an axiom expressing the flatness of the domain ID, and use it to discharge the antecedents of the equality axioms we wish to use.

5. That is, the shape of the domain of HPROGRAM is



We could neaten the presentation by assuming the definition of an n-ary sum operator, and so conceal the lifting operations, but this introduces the problem of generating names and axioms for the injection, projection and selector functions, e.g., IN1, IN2, etc.? -- and it requires these functions to be defined differently than at present. Thus, to avoid confusion, we leave things as they are. At any rate, the UP's and DOWN's are seen <u>only</u> in the first layer of lemmas proved.

6. This is, in fact, one of the points at which our presentation is idealised. The standard type operator for sum, at the time this proof was produced in LCF (but not any longer) was binary separated sum, and there was no facility then for defining new type operators. Therefore, the domain of high level programs had the shape



The apprcrpiate cases tactic used a cases rule which did nested case analysis:

HCASESTAC

(w, ss, A)

---

w
(isassign p ≡ TT) + ss
(isassign p ≡ TT)
A

w
(isassign p ≡ ⊥) + ss
(isassign p ≡ ⊥)
A

w
(isassign p ≡ FF) + (isif p ≡ TT) + ss
(isassign p ≡ FF) (isif p ≡ TT)
A

w
(isassign p ≡ FF) + (isif p ≡ ⊥) + ss
(isassign p ≡ FF) (isif p ≡ ⊥)
A

w
(isassign p ≡ FF) + (isif p ≡ FF) + (iswhile p ≡ TT) + ss
(isassign p ≡ FF) (isif p ≡ FF) (iswhile p ≡ TT)
A

w
(isassign p ≡ FF) + (isif p ≡ FF) + (iswhile p ≡ ⊥) + ss
(isassign p ≡ FF) (isif p ≡ FF) (iswhile p ≡ ⊥)
A

w
(isassign p ≡ FF) + (isif p ≡ FF) + (iswhile p ≡ FF) + ss
(isassign p ≡ FF) (isif p ≡ FF) (iswhile p ≡ FF)
A

The four subgoals that form a column on the left are the main ones; simplification solves the other three 'spurious' cases.

7. In fact, we must add the proviso that the 'highest powered' induction hypothesis is used, so that, for example, for the formula arising during the while proof of Theorem 3.1b, the step,

$$\text{LSEM lsem}' [\![C(p)]\!] \text{ L1} \sqsubseteq \text{hsem} [\![p]\!]$$

IH2, that is,

$$\forall p.\ \text{LSEM lsem}' [\![C(p)]\!] \text{ L1} \sqsubseteq \text{hsem} [\![p]\!]$$

is used in preference to IH1, that is,

$$\forall p.\ \text{lsem}' [\![C(p)]\!] \text{ L1} \sqsubseteq \text{hsem} [\![p]\!]$$

Chapter 4: Implementation of Procedure Declaration

In this chapter, we consider the correctness of the implementation of another pair of constructs: procedure declaration and invocation. We define a block structured high level language in which recursive and non-recursive procedures may be declared and called, and a low level language whose semantics reflects a standard stack implementation. Both languages are as streamlined as possible to our purposes. An algorithm to compile high level into low level programs is presented and informally proved. To facilitate the proof, the compilation is factored into three stages. We concentrate on the theoretical difficulties in expressing the relations between the various levels, and on the three informal proofs.

The emphasis in this work has been to supply semantics and proofs amenable to expression in LCF. Although the proofs have not been performed in the system, they have reached their present structure only because machine proof was envisaged; formalisation in LCF requires a level of rigour which reveals the need for extreme care. We believe that the generation in LCF of the proofs presented in this chapter would be a feasible undertaking; our optimism is based on the results of the machine generated proof described in Chapter 3, and on the machine proof outlined in this chapter.

The Problem

The High Level Language

In the source language, we allow blocks in which local variables may be declared, and procedures invoked. We model static binding of variables. For simplicity, we consider blocks with exactly one declaration apiece, and procedures without parameters. We allow identifiers to denote only procedures. All of this makes for a rather odd language, but enables us to focus on the issue at hand: the correctness of the implementation of recursive procedures. We believe that enrichments of the language, such as inclusion of a parameter-passing mechanism, or multiple declarations in blocks, would require more detailed, but not essentially different proofs.

We let I range over a domain ID of identifiers, p1 and p2 over a domain HPROGRAM of high level programs (distinct from HPROGRAM in Chapter 3), and the variable a, over a domain A of (unspecified) atomic programs. High level programs are given by:

p ::= <u>let</u> I = p1 <u>in</u> p2 |
<u>letrec</u> I = p1 <u>in</u> p2 |
<u>call</u> I |
p1;p2 |
a

The first two constructs specify blocks with non-recursive and recursive procedure declarations, respectively; the third is procedure invocation, and the fourth is sequencing.

A standard (direct) denotational semantics is given for this language. It requires a domain, HENV, of environments, in which identifiers are mapped to the meanings of programs (since all identifiers denote procedures). The meanings of programs are transformations on stores; the structure of the domain STORE is not important for our purposes. We let σ range over STORE. Thus we have

$$\rho \in \text{HENV} = \text{ID} \longrightarrow \text{STORE} \longrightarrow \text{STORE}$$

We define semantic functions $\mathcal{A}$ for atomic programs, and $\mathcal{B}$, for whole programs:

$$\mathcal{A} : \text{A} \longrightarrow \text{STORE} \longrightarrow \text{STORE}$$

$$\mathcal{B} : \text{HPROGRAM} \longrightarrow \text{HENV} \longrightarrow \text{STORE} \longrightarrow \text{STORE}$$

The clauses for $\mathcal{B}$ are:

$$\mathcal{B}[\![\underline{\text{let}}\ \text{I} = \text{p1}\ \underline{\text{in}}\ \text{p2}]\!]\rho = \mathcal{B}[\![\text{p2}]\!]\ (\rho[\mathcal{B}[\![\text{p1}]\!]\rho/\text{I}])$$

$$\mathcal{B}[\![\underline{\text{letrec}}\ \text{I} = \text{p1}\ \underline{\text{in}}\ \text{p2}]\!]\rho = \mathcal{B}[\![\text{p2}]\!]\ (\text{FIX}(\lambda\rho'.\rho[\mathcal{B}[\![\text{p1}]\!]\rho'/\text{I}]))$$

$$\mathcal{B}[\![\underline{\text{call}}\ \text{I}]\!]\rho = \rho\,\text{I}$$

$$\mathcal{B}[\![\text{p1};\text{p2}]\!]\rho = \lambda\sigma.\ \mathcal{B}[\![\text{p2}]\!]\ (\mathcal{B}[\![\text{p1}]\!]\rho\sigma)$$

$$\mathcal{B}[\![\text{a}]\!]\rho = \mathcal{A}[\![\text{a}]\!]$$

We assume that $\forall \text{a}.\ \mathcal{A}[\![\text{a}]\!]\perp = \perp$. The only difficult clause is the one for procedure declaration. In the case in which I is recursively declared to denote p1 throughout p2, we take the meaning of the body p2 in an environment, say $\hat{\rho}$, which is like ρ except on I; I is bound to the meaning of p1 in $\hat{\rho}$:

$$\hat{\rho} = \rho[\mathscr{D}[\![pl]\!]\hat{\rho}/I]$$
$$= FIX(\lambda\rho'.\rho[\mathscr{D}[\![pl]\!]\rho'/I])$$

The Low Level Language

We consider a low level language whose corresponding machine allows an implementation (albeit rather abstract) of the source language. The idea is to maintain an activation stack while running low level programs, each of whose entries represents a block entry or precedure invocation. In each stack entry, certain information must be preserved, namely, the dynamic link (which we take to be the previous stack element), for return upon exit; the static like (a pointer to the activation record representing the textually enclosing block in the program), for finding the meanings of non-local variables; and the meanings for the local variables declared in the current block. The basic instructions in the low level language include instructions for making new entries on the activation stack, for deleting entries, and for restoring the declaration time environment when procedures are invoked, by using the static link (to reflect static binding).

We let I range over ID (as before), q and qi over a domain LPROGRAM of low level programs (distinct from LPROGRAM in Chapter 3), and the variable a over A (as before). Low level programs are then given by:

q ::= PRENTRY(I,q) |

RECENTRY(I,q) |

EXIT |

CALL(I) |

q1;q2 |

a

PRENTRY(I,q) creates a stack entry for a non-recursive block in which the identifier I is declared to denote the procedure q. RECENTRY(I,q) creates a stack entry for a block in which I recursively denotes q. EXIT is for all exits from blocks and procedures. CALL(I) creates a stack entry appropriate for entry to the procedure denoted by I. The fifth clause is for sequencing, and the sixth, for atomic statements.

The intended semantics for this language is an operational semantics based on an abstract machine, one of whose components is a stack. (The semantics is still denotational, though, in the sense that the meanings of the various constructs are functions of the meanings of their components.) Each entry in the stack is a whole environment. The stack is indexed by integers. An environment, at this level, maps identifiers to pairs consisting of a low level program (the procedure body which the identifier denotes) and an integer (a pointer back into the stack). Environments form a domain LENV:

$$LENV = ID \longrightarrow (LPROGRAM \times INT)$$

The integer component of the meaning of an identifier plays the role

of static link; it points to another level in the activation stack, at which another environment is to be found which maps identifiers to program-integer pairs, and so on upwards. We let δ range over a domain LAS (for low level activation stacks):

$$\delta \in \text{LAS} \quad = \quad \text{INT} \longrightarrow \text{LENV}$$

The context, or configuration, in which a low level program is 'executed' has three components: an integer pointer into the activations stack (representing the current dynamic level), the activation stack itself, and a store (which we take to be the same as in the high level semantics). We define a domain CONFIG of configurations:

$$\text{CONFIG} \quad = \quad \text{INT} \times \text{LAS} \times \text{STORE}$$

The low level semantic function maps programs and configurations to new configurations; that is, running programs can affect the stack, the pointer and the store. We call the semantic function $\mathcal{Run}$.

$$\mathcal{Run}\text{:LPROGRAM} \longrightarrow \text{CONFIG} \longrightarrow \text{CONFIG}$$

The clauses for $\mathcal{Run}$ are given below. We use the notation

$$\delta\,[n \longmapsto x]$$

to mean δ extended at n to the value x. This is to avoid confusion with the usual extension notation, which is reserved for environments.

$$\mathcal{R}un [\![\mathrm{PRENTRY(I,q)}]\!] \ (m,\delta,\sigma) = (m+1, \ \delta[(m+1) \mapsto (\delta\ m)[(q,m)/I]], \ \sigma)$$

$$\mathcal{R}un [\![\mathrm{RECENTRY(I,q)}]\!] \ (m,\delta,\sigma) = (m+1, \ \delta[(m+1) \mapsto (\delta\ m)[(q,m+1)/I]], \ \sigma)$$

$$\mathcal{R}un [\![\mathrm{CALL\ (I)}]\!] \ (m,\delta,\sigma) = \mathcal{R}un [\![q']\!] \ (m+1, \ \delta[(m+1) \mapsto \delta\ m'], \ \sigma) \quad \text{where } (q',m') = \delta\ m\ I$$

$$\mathcal{R}un [\![\mathrm{EXIT}]\!] \ (m,\delta,\sigma) = (m-1,\delta,\sigma)$$

$$\mathcal{R}un [\![q1;q2]\!] \ (m,\delta,\sigma) = \mathcal{R}un [\![q2]\!] \ (\mathcal{R}un [\![q1]\!] \ (m,\delta,\sigma,))$$

$$\mathcal{R}un [\![a]\!] \ (m,\delta,\sigma) = \langle\!\langle m,\delta, \mathcal{A} [\![a]\!] \sigma \rangle\!\rangle$$

In the atomic case, we wish the whole triple to be undefined if $\mathcal{A} [\![a]\!] \sigma$ is undefined, so we introduce a notation for triples strict in the third argument:

$$\langle\!\langle x,y,z \rangle\!\rangle$$

is $\perp$ if z is $\perp$, and (x,y,z) otherwise. The reasons for this are technical, and are discussed later.

For procedure entry (where I is declared to denote q), $\mathcal{R}un$ makes a new entry to the stack at level (m+1), consisting of the previous environment extended at I so that I now denotes the program q paired with the (declaration) level m. $\mathcal{R}un$ also increments the pointer so that it points to the new stack entry.

For entry to recursive procedures where I recursively denotes q, the environment for the new stack entry (again at level (m+1)) maps I to q paired with the level (m+1), so that occurrences of I within q denote q, but with the declaration time environment at level (m+1).

For calls, *Run* determines the denotation of I in the current environment (that is, the procedure it denotes, and its declaration time level, q' and m', respectively), makes a new stack entry consisting of the declaration time environment, increments the pointer to point to the new entry, and applies itself to q', with the new stack and pointer. Exits are just decrements of the pointer.

At first view this model may not appear to be very concrete; although procedures are now represented concretely, activation stacks are <u>functions</u> (infinite vectors) mapping integers to whole environments (also functions), rather than to new layers on old environments. Nonetheless, we believe that this is a good level at which to aim because it captures the essence of the implementation. In particular, a fixed point in the semantics for recursive procedures has been replaced by a 'knot' in the activation stack. (This is the transition which presents the theoretical difficulties to which we referred.) A model in which displays, in the usual sense (e.g. as defined in [48]) were kept would be a natural next step in the transition from an abstract semantics to an implementation. For concretisations of this sort, the low level language described here, and its semantics, would serve as a useful intermediate stage between the abstract and the more concrete semantics.[1]

The Compiler

The compiling algorithm, C, (distinct from C in Chapter 3), maps high level to low level programs. As in Chapter 3, we take the abstract syntax of the two languages as our starting point, and do not consider problems of parsing. C has type

C:HPROGRAM ⟶ LPROGRAM

and is defined by the following clauses:

C ⌜let I = p1 in p2⌝	=	PRENTRY(I,C(p1)) C(p2) EXIT
C ⌜letrec I = p1 in p2⌝	=	RECENTRY(I,C(p1)) C(p2) EXIT
C ⌜call I⌝	=	CALL(I) EXIT
C ⌜p1;p2⌝	=	C(p1) C(p2)
C ⌜a⌝	=	a

(For appearances, we have concealed the sequencing operator in low level programs.) The compiler produces entry and exit instructions for blocks, with the compiled bodies in between, and it produces low level calls with exits, for high level calls. Sequenced programs are compiled into sequences of compiled programs, and atomic programs are uncompiled.

The Equivalence Proofs

Several complications arise in proving the equivalence of all high level programs to their compiled images. Some of these are related to stating the equivalences; others to the transition from a semantics with an explicit fixed point for recursive procedures to one with a knot.

In stating the equivalences, we wish, for several reasons, to avoid the use of recursively defined relations, although they seem natural at first glance. One reason is that the formal theory of recursive relations, unlike that of recursive functions, is not fully understood. More particularly, recursive relations cannot be expressed in PPLAMBDA, and would therefore place the proof outside the scope of LCF.

Typically, statements of the equivalence of semantics at different levels have the form

> If the contexts of the semantic functions are suitably related (i.e. simulate each other) then the results of applying the semantic functions to corresponding programs are also suitable related.

By context we mean simply the parameters to the semantic function; the environment and store, or the stack and pointer, or whatever the functions require. It is in stating these 'suitable' relations that the problems arise; the obvious relations are often recursive.

We have found, in stating the relations, that only certain properties are required to hold of the contexts. As long as these properties imply the recursive properties, the recursive properties

need not be taken as definitions. That is, we explicitly construct solutions to the simulation relations that we need, and thus do not have to appeal to any existence theorems about recursive relations. Whether this can always be done is a question we do not address.

We have found that the difficulties raised by trying to relate 'incompatible' kinds of semantics, in the examples considered here, can be largely sorted out by factoring the proof into three stages, introducing two intermediate levels. In the first, a closure semantics is given for the high level language. In the second, a more abstract version of the activation stack implementation (for the *high* level language) is considered. The fixed-point-to-knot problem arises in the transition from the former to the latter. The key to solving the problem is the introduction of *abstracting functions* which map activation stacks (or other concrete sorts of contexts) to more abstract structures which *can* be compared with the environments containing fixed points.

Standard to Closure Semantics Proof ($\mathcal{S}$ to $\overline{\mathcal{S}}$)

The first stage in the transition from $\mathcal{S}$ to $\mathcal{Run}$ is to define a closure semantics, $\overline{\mathcal{S}}$, for the high level language, and to prove it equivalent to $\mathcal{S}$. The environments, in $\overline{\mathcal{S}}$, map identifiers to *closures*, which are pairs consisting of programs and (declaration time) environments. Closures are representations of the meanings of procedures in $\mathcal{S}$, that is, representations of store transformations.

We let v range over HCENV, a reflexive domain of closure environments:

$$v \in \text{HCENV} = \text{ID} \longrightarrow (\text{HPROGRAM} \times \text{HCENV})$$

The semantics function $\overline{\mathcal{S}}$ has type

$$\overline{\mathcal{S}} : \text{HPROGRAM} \longrightarrow \text{HCENV} \longrightarrow \text{STORE} \longrightarrow \text{STORE}$$

Its clauses are

$$\begin{aligned}
\overline{\mathcal{S}}[\![\underline{\text{let}}\ I = p1\ \underline{\text{in}}\ p2]\!]\ v &= \overline{\mathcal{S}}[\![p2]\!](v[(p1,v)/I]) \\
\overline{\mathcal{S}}[\![\underline{\text{letrec}}\ I = p1\ \underline{\text{in}}\ p2]\!]\ v &= \overline{\mathcal{S}}[\![p2]\!](\text{FIX}(\lambda v'.v[(p1,v')/I])) \\
\overline{\mathcal{S}}[\![\underline{\text{call}}\ I]\!]\ v &= \overline{\mathcal{S}}[\![p']\!]\ v' \quad \text{where } (p',v') = v\ I \\
\overline{\mathcal{S}}[\![p1;p2]\!]\ v &= \lambda\sigma.\overline{\mathcal{S}}[\![p2]\!]\ v\ (\overline{\mathcal{S}}[\![p1]\!]\ v\ \sigma) \\
\overline{\mathcal{S}}[\![a]\!]\ v &= \mathcal{A}[\![a]\!]
\end{aligned}$$

In stating the equivalence of $\mathcal{S}$ and $\overline{\mathcal{S}}$, we must first state the simulation relation between the repective contexts, that is, between

$$\rho \in \text{HENV} = \text{ID} \longrightarrow \text{STORE} \longrightarrow \text{STORE}$$

and

$$v \in \text{HCENV} = \text{ID} \longrightarrow (\text{HPROGRAM} \times \text{HCENV})$$

The obvious relation is

$$\forall I.\ \rho\ I = \mathcal{S}[\![\text{FST}(v\ I)]\!]\ (\text{SND}(v\ I))$$

which is to say

$$\forall I.\rho\ I = \overline{\mathcal{S}}[\![\underline{\text{call}}\ I]\!]\ v$$

We abbreviate this relation by writing $\rho \sim v$. Our goal is to prove

<u>Theorem</u> <u>4.1</u>

$$\forall p\ \rho\ v.\ \rho \sim v \supset \mathcal{S}[\![p]\!]\rho = \bar{\mathcal{S}}[\![p]\!]\ v$$

The proof is facilitated by a lemma.

<u>Lemma</u> <u>4.2</u>

$$\forall \rho\ v\ I.\ \rho \sim v \supset \forall v'\ p'.\ \rho[\bar{\mathcal{S}}[\![p']\!]\ v'/I] \sim v[(p',v')/I]$$

<u>Proof</u> <u>of</u> <u>Lemma</u> <u>4.2</u>

We show

$$\rho \sim v \supset \forall v'\ p'\ J.\ \rho[\bar{\mathcal{S}}[\![p']\!]\ v'/I]\ J =$$

$$\bar{\mathcal{S}}[\![\underline{\text{call}}\ J]\!](v[(p',v')/I])$$

We assume that $\rho \sim v$.

<u>Case</u> <u>$J \neq I$</u>

We must show that

$$\rho\ J = \bar{\mathcal{S}}[\![\underline{\text{call}}\ J]\!]\ v$$

This follows from the assumption.

<u>Case</u> <u>$J = I$</u>

We must show that

$$\bar{\mathcal{S}}[\![p']\!]\ v = \bar{\mathcal{S}}[\![\underline{\text{call}}\ I]\!](v[(p',v')/I])$$

$$= \bar{\mathcal{S}}[\![p']\!]\ v'$$

by definition of $\bar{\mathcal{S}}$. Q.E.D.

The proof of <u>Theorem</u> <u>4.1</u> is by structural induction on high level programs.

<u>Proof</u> <u>of</u> <u>Theorem</u> <u>4.1</u>

<u>Bases</u>

If $p = \perp$ or $\ulcorner a \urcorner$, the proofs are easy, assuming that $\rho \sim v$.

<u>Case</u> <u>$p = \ulcorner\underline{\text{call}}\ I\urcorner$</u>

$$\mathcal{S}[\![\underline{\text{call}}\ I]\!]\rho = \rho\ I = \bar{\mathcal{S}}[\![\underline{\text{call}}\ I]\!]\ v,$$

by the definition of $\mathcal{S}$ and the assumption.

Step
We assume the theorem with p1 and p2 for p, and we assume $\rho \sim v$.

Case p = ⌜let I = p1 in p2⌝
We must show

$$\mathcal{S}[\![\underline{let}\ I = p1\ \underline{in}\ p2]\!]\rho = \bar{\mathcal{S}}[\![\underline{let}\ I = p1\ \underline{in}\ p2]\!]v$$

that is,

$$\mathcal{S}[\![p2]\!](\rho[\mathcal{S}[\![p1]\!]\rho/I]) = \bar{\mathcal{S}}[\![p2]\!](v[(p1,v)/I])$$

In order to use the induction hypothesis, we must show

$$\rho[\mathcal{S}[\![p1]\!]\rho/I] \sim v[(p1,v)/I]$$

This follows by a use of the induction hypothesis with the assumption that $\rho \sim v$, and by Lemma 4.2.

Case p = ⌜letrec I = p1 in p2⌝
We must show

$$\mathcal{S}[\![p2]\!](FIX(\lambda\rho'.\rho[\mathcal{S}[\![p1]\!]\rho'/I])) =$$
$$\bar{\mathcal{S}}[\![p2]\!](FIX(\lambda v'.v[(p1,v')/I]))$$

We call the two functional $\mathcal{R}$ and $\mathcal{V}$ respectively, and prove

$$\mathcal{S}[\![p2]\!](FIX\ \mathcal{R}) = \bar{\mathcal{S}}[\![p2]\!](FIX\ \mathcal{V})$$

This requires an inner computation induction in order to prove $FIX\ \mathcal{R} \sim FIX\ \mathcal{V}$, and so complete the proof, by hypothesis.

Assume
$\bar{\rho} \sim \bar{v}$, for arbitrary $\bar{\rho}$ and $\bar{v}$.

Show
$\mathcal{V}\ \bar{v} \sim \mathcal{R}\ \bar{\rho}$, that is,

$$\rho[\mathcal{S}[\![p1]\!]\bar{\rho}/I] \sim v[(p1,\bar{v})/I]$$

By the outer hypothesis, with p1 for p, and the inner hypothesis, we have

$$\mathcal{S}[\![p1]\!]\bar{\rho} = \bar{\mathcal{S}}[\![p1]\!]\bar{v}$$

Hence, by Lemma 4.2, the result follows.

Case p = p1;p2
$$\mathcal{S}[\![p1;p2]\!] = \lambda\sigma.\ \mathcal{S}[\![p2]\!]\rho\ (\mathcal{S}[\![p1]\!]\rho\sigma) =$$
$$\lambda\sigma.\ \bar{\mathcal{S}}[\![p2]\!]\ v\ (\bar{\mathcal{S}}[\![p1]\!]\ v\ \sigma) = \bar{\mathcal{S}}[\![p1;p2]\!]\ v \quad \text{--Q.E.D.}$$ [2]

Closure to Abstract Stack Semantics Proof ($\bar{\mathcal{S}}$ to $\mathcal{D}$)

The more concrete semantics $\bar{\mathcal{S}}$ is a good point from which to consider and prove equivalences to implementation-oriented models. As a step toward proving the equivalence of $\mathcal{S}$ to $\mathcal{R}un$, we next consider a stack semantics called $\mathcal{D}$, more abstract than $\mathcal{R}un$, which implements the high level language. This factors out (and defers until the next stage) the problem of compiling into the low level language.

$\mathcal{D}$ uses an activation stack similar to the one for $\mathcal{R}un$, but it is used more abstractly. Rather than changing the stack by running a program, $\mathcal{D}$ simply interprets programs to determine the store transformations they denote. We introduce environments which map identifiers to pairs consisting of high level programs and pointers into the stack. Activations stacks again map integers into environments. We introduce a domain DENV of environments, and let d range over a domain HAS of high level activation stacks:

$$\text{DENV} = \text{ID} \longrightarrow (\text{HPROGRAM} \times \text{INT})$$

$$d \in \text{HAS} = \text{INT} \longrightarrow \text{DENV}$$

The semantic function $\mathcal{D}$ has type

$$\mathcal{D} : \text{HPROGRAM} \longrightarrow \text{HAS} \longrightarrow \text{INT} \longrightarrow \text{STORE} \longrightarrow \text{STORE}$$

and is given by the following clauses:

$$\mathcal{D}[\![\underline{\text{let}}\ I = p1\ \underline{\text{in}}\ p2]\!]\ d\ n \quad = \quad \mathcal{D}[\![p2]\!]\ (d[(n+1) \mapsto (d\ n)[(p1,n)/I]])\ (n+1)$$

$$\mathcal{D}[\![\underline{\text{letrec}}\ I = p1\ \underline{\text{in}}\ p2]\!]\ d\ n \quad = \quad \mathcal{D}[\![p2]\!]\ (d[(n+1) \mapsto (d\ n)[(p1,n+1)/I]])\ (n+1)$$

$$\mathcal{D}[\![\underline{\text{call}}\ I]\!]\ d\ n \quad = \quad \mathcal{D}[\![p']\!]\ d\ n' \quad \text{where } (p',n') = d\ n\ I$$

$$\mathcal{D}[\![p1;p2]\!]\ d\ n \quad = \quad \lambda\sigma.\ \mathcal{D}[\![p2]\!]\ d\ n\ (\mathcal{D}[\![p1]\!]\ d\ n\ \sigma)$$

$$\mathcal{D}[\![a]\!]\ d\ n \quad = \quad \mathcal{A}[\![a]\!]$$

We observe that in $\mathcal{D}$, we do not `over-write` the stack, but use it only for reference, and also that procedure invocation does not cause a new stack entry; we simply revert to the declaration time level in the activation stack. The semantics of procedure values are now fully `defunctionalised`; procedures are represented by texts and integers, rather than by functions.

The formulation of the equivalence of $\mathcal{S}$ and $\mathcal{D}$ requires that the simulation relation between contexts be defined. We relate a closure environment to an abstract stack with its pointer; that is

$$v \in \text{HCENV} = \text{ID} \longrightarrow (\text{HPROGRAM} \times \text{HCENV})$$

to

$$d \in \text{HAS} = \text{INT} \longrightarrow \text{DENV}, \quad \text{and } n$$

where

$$\text{DENV} = \text{ID} \longrightarrow (\text{HPROGRAM} \times \text{INT})$$

The relation $\approx$ between v:HCENV and a pair (d:DENV, n:INT) that we seek should have the property that

$$v \approx (d,n) \quad \text{iff} \quad \forall I.\ \text{FST}(v\ I) = \text{FST}(d\ n\ I)\ \ \& \ \ \text{SND}(v\ I) \approx (d, \text{SND}(d\ n\ I))$$

but of course this is not a well formed <u>definition</u>. We therefore construct a relation satisfying the above property.

To this end, we introduce a function H which <u>abstracts</u> pairs consisting of an activation stack and pointer, to a closure environment, so that the pairs can be compared to closure environments:

$$H:(HAS \times INT) \longrightarrow HCENV$$

We define H recursively:

$$H(d,n)I \quad = \quad FST(d\ n\ I),\ H(d,SND(d\ n\ I))$$

that is,

$$H = FIX\ FUNH$$

where

$$FUNH \quad = \quad \lambda H'\ (d,n)\ I.\ FST(d\ n\ I),\ H'(d,\ SND(d\ n\ I))$$

H (intuitively) traces up the static chain to construct whole environments. We now define $v \approx (d,n)$ to abbreviate the formula $v = H(d,n)$. It can easily be shown that $\approx$ satisfies the desired property (above).

For the proof of the equivalence of $\bar{\mathcal{S}}$ to $\mathcal{D}$, we need a well-foundedness property of activation stacks paired with pointers, to express the condition that (up to a certain point), the declaration level of a procedure is never greater than the level

from which the procedure is called. We express this property of a stack d and an integer n by the formula

$$\forall I\ n'.\ n' \leqq n \supset SND(d\ n'\ I) \leq n'$$

which we abbreviate as hgood d n. To shorten the statements of theorems, we include this property in the definition of the relation $\approx$:

$$v \approx (d,n) \quad \text{iff} \quad v = H(d,n) \quad \& \quad \text{hgood } d\ n$$

The property hgood is used in proving that the closure environments abstracted from two stacks, at some point n, where the two stacks agree up to n, are the same; a fact which is used in the proof of the main theorem relating $\bar{\mathcal{S}}$ and $\mathcal{D}$. We let $d1 \underset{n}{=} d2$ abbreviate the formula

$$\forall\ n'.\ n' \leqq n \supset d1\ n' = d2\ n'$$

<u>Lemma 4.3</u>

$\forall d1\ d2\ n.\ d1 \underset{n}{=} d2 \quad \& \quad \text{hgood } d1\ n \supset H(d1,n) = H(d2,n)$

<u>Proof of Lemma 4.3</u>

By induction on H. We assume the theorem for H′ and we assume that $d1 \underset{n}{=} d2$ and hgood d n. We show

$$\forall I.\ FST(d1\ n\ I),\ H'(d1,SND(d1\ n\ I)) =$$
$$FST(d2\ n\ I),\ H'(d2,SND(d2\ n\ I))$$

Since by assumption d1 n = d2 n, and $SND(d1\ n\ I) \leqq n$, by hgood-ness, the rest follows by assumption, with SND(d1 n I) for n. Q.E.D.

For the main theorem relating $\bar{\mathcal{S}}$ and $\mathcal{D}$, a separate lemma is

required for the letrec case, for which we use a more general lemma, analogous to Lemma 4.2.

Lemma 4.4

$\forall m\ n\ v\ d\ pl.\ m \leqq n+1 \quad \& \quad v \approx (d,n) \quad \supset$

$v[(pl,H(\hat{d},m))/I] \approx (\hat{d},n+1)$

where $\hat{d} = d[(n+1) \longmapsto (d\ n)[(pl,m)/I]]$

This lemma is useful in both the letrec and let cases.

Proof of Lemma 4.4

Assume $m \leqq n+1$ and $v \approx (d,n)$. That hgood $\hat{d}$ (n+1) holds is obvious. For the rest we must show

$\forall I'.\ v[(pl,H(d,m))/I]\ I' = H(\hat{d},n+1)I'$

Case $I' \neq I$

LHS = v and RHS = $FST(\hat{d}\ n\ I'),\ H(\hat{d},SND(\hat{d}\ n\ I'))$

Since in this case $\hat{d}\ n\ I' = d\ n\ I'$, it follows by Lemma 4.3 that

$H(\hat{d},SND(\hat{d}\ n\ I')) = H(d,SND(d\ n\ I'))$

Therefore, by the definition of H, RHS = H(d,n), and the result follows by the assumption.

Case $I' = I$

Both sides reduce to $(pl,H(\hat{d},m))$. Q.E.D.

The lemma for the letrec case is:

Lemma 4.5

$\forall v\ d\ n\ pl.\ v \approx (d,n) \supset \hat{v} \approx (\hat{d},n+1)$

where $\hat{v} = FIX\ \mathcal{V}$

where $\mathcal{V} = \lambda v'.\ v[(pl,v')/I]$

and $\hat{d} = d[(n+1) \longmapsto (d\ n)[(pl,n+1)/I]]$

Proof of Lemma 4.5

We assume that $v \approx (d,n)$ and prove the consequent in two directions, showing

(i) $\hat{v} \subseteq H(\hat{d},n+1)$

(ii) $H(\hat{d},n+1) \subseteq \hat{v}$

That hgood $\hat{d}$ (n+1) is obvious.

Proof of (i)

It is easy to show that $H(\hat{d},n+1)$ is a fixed point of $\mathcal{V}$, and thus that $v \subseteq H(\hat{d},n+1)$. We show that $\mathcal{V}(H(\hat{d},n+1)) = H(\hat{d},n+1)$:

LHS = v[(p1,H($\hat{d}$,n+1))/I]

By Lemma 4.4 with m = n+1, we have

v[(p1,H($\hat{d}$,n+1))/I] = H($\hat{d}$,n+1)

= RHS. Q.E.D.

Proof of (ii)
We prove instead

$H \subseteq$ FIX FUNH & $H(\hat{d},n+1) \subseteq \hat{v}$

recalling that H = FIX FUNH [3]. The proof is by induction on both occurrences of H.

Assume
$H' \subseteq$ FIX FUNH & $H'(\hat{d},n+1) \subseteq \hat{v}$

Show
FUNH $H' \subseteq$ FIX FUNH & FUNH $H'(\hat{d},n+1) \subseteq \hat{v}$

The first conjunct is easy. The second unfolds to

$\forall I'.\ FST(\hat{d}(n+1)I'),\ H'(\hat{d},SND(\hat{d}(n+1)I')) \sqsubseteq \hat{v}\ I'$

<u>Case</u> <u>$I' = I$</u>
LHS = $p1, H'(\hat{d},n+1)$

RHS = $p1,\ \hat{v}$

so the result follows by hypothesis.

<u>Case</u> <u>$I' \neq I$</u>
LHS = $FST(d\ n\ I'),\ H'(\hat{d},SND(d\ n\ I'))$

RHS = $v\ I'$

= $H'(d,n)I'$
by assumption, second part

$\sqsubseteq H(d,n)I'$
by assumption, first part

= $FST(d\ n\ I'),\ H(d,SND(d\ n\ I'))$
by definition of H

= $FST(d\ n\ I'),\ H(\hat{d},SND(d\ n\ I'))$
by <u>Lemma 4.3</u>

But by hypothesis, $H' \sqsubseteq H$, and this completes the proof. Q.E.D.

The main theorem relating $\bar{\mathcal{S}}$ and $\mathcal{D}$ is:

<u>Theorem 4.6</u>
$\forall p\ v\ d\ n.\ v \approx (d,n) \supset \bar{\mathcal{S}}[\![p]\!]\ v = \mathcal{D}[\![p]\!]\ d\ n$

<u>Proof of Theorem 4.6</u>
The proof is by computation induction on $\bar{\mathcal{S}}$ and $\bar{\mathcal{D}}$.

<u>Assume</u>
$\forall p\ v\ d\ n.\ v \approx (d,n) \supset \bar{\mathcal{S}}'[\![p]\!]v = \mathcal{D}'[\![p]\!]\ d\ n$

for arbitrary $\bar{\mathcal{S}}'$ and $\mathcal{D}'$, and assume that $v \approx (d,n)$ for some v, d and n. The $\perp$ case is straightforward. For the step, the various cases are considered. The atomic case is easy.

<u>Case</u> <u>p</u> = ⌜<u>let</u> <u>I</u> = <u>p1</u> <u>in</u> <u>p2</u>⌝
<u>Show</u>
$\bar{\mathcal{S}}'[\![p2]\!]v[(p1,v)/I] = \mathcal{D}'[\![p2]\!]\ (d[(n+1)\mapsto (d\ n)[(p1,n)/I]])\ (n+1)$

where we abbreviate the stack on the right hand side as $\hat{d}$. By Lemma 4.4 with m = n, we have

$$v[(p1,H(d,n))/I] \approx (\hat{d},n+1)$$

By assumption, v = H(d,n), so by Lemma 4.3, v = H(d,n). Thus,

$$v[(p1,v)/I] \approx (\hat{d},n+1)$$

and the induction hypothesis applies.

Case p = ⌜call I⌝
Show

$$\bar{\mathcal{S}}'[\![p']\!]v' = \mathcal{D}'[\![p'']\!]\ d\ n''$$

where (p′,v′) = v I and (p″,n″) = d n I

By assumption, v I = (FST(d n I), H(d,SND(d n I)))

so we show that

$$\bar{\mathcal{S}}'[\![p']\!]\ (H(d,SND(d\ n\ I))) = \mathcal{D}'[\![p']\!]\ d\ (SND(d\ n\ I))$$

As it is obvious that hgood d (SND(d n I)), this is true by hypothesis.

Case p = ⌜letrec I = p1 in p2⌝
Show

$$\bar{\mathcal{S}}'[\![p2]\!]\ \hat{v} = \mathcal{D}'[\![p2]\!]\ \hat{d}\ (n+1)$$

where $\hat{v}$ and $\hat{d}$ are as in Lemma 4.5. That lemma enables use of the induction hypothesis.

Case p = ⌜p1;p2⌝
Show

$$\lambda\sigma.\ \bar{\mathcal{S}}'[\![p2]\!]\ v\ (\bar{\mathcal{S}}'[\![p1]\!]\ v\,\sigma) =$$

$$\lambda\sigma.\ \mathcal{D}'[\![p2]\!]\ d\ n\ (\mathcal{D}'[\![p1]\!]\ d\ n\,\sigma)$$

This follows directly, by two uses of the hypothesis. Q.E.D.

Abstract Stack to Concrete Stack Semantics Proof ($\mathcal{D}$ to $\mathcal{R}un$)

The transition from $\mathcal{S}$ to $\mathcal{R}un$ is completed by proving $\mathcal{D}$ equivalent to $\mathcal{R}un$. The key, again, is in relating the contexts: a high level activation stack and pointer, and a low

level activation stack and pointer:

$d \in HAS$ and n, where $HAS = INT \longrightarrow DENV$

and $DENV = ID \longrightarrow (HPROGRAM \times INT)$

and

$\delta \in LAS$ and m, where $LAS = INT \longrightarrow LENV$

and $LENV = ID \longrightarrow (LPROGRAM \times INT)$

At first glance, the following relation may appear to be adequate:

$$(d,n) \approx (\delta,m) \quad \text{iff} \quad n = m \quad \&$$

$$\forall n' \; I. \; n' \leqq n \supset C(FST(d \; n' \; I)) = FST(\delta \; n' \; I) \quad \&$$

$$SND(d \; n' \; I) = SND(\delta \; n' \; I)$$

That is, at corresponding levels the two stacks have corresponding programs and pointers. However, the two semantics $\mathcal{D}$ and $\mathcal{R}un$ affect the stacks differently; in particular the call semantics are different. Therefore, this relation is not general enough.

Instead we employ two abstracting functions, J and L, similar in nature to H, but abstracting to a new sort of environment, a low level closure environment, in which identifiers are mapped to pairs consisting of <u>low</u> level programs and low level closure environments (reflexively):

$$LCENV = ID \longrightarrow (LPROGRAM \times LCENV)$$

We define

$J:(HAS \times INT) \longrightarrow LCENV$

$L:(LAS \times INT) \longrightarrow LCENV$

as J = FIX FUNJ and L = FIX FUNL, where

$$FUNJ = \lambda J' \ (d,n)\ I.\ C(FST(d\ n\ I)),\ J'(d,SND(d\ n\ I))$$

$$FUNL = \lambda L' \ (\delta,m)\ I.\ FST(\delta\ m\ I),\ L'(\delta,SND(\delta\ m\ I))$$

The property desired of the relations, this time, is

$$(d,n) \approx (\delta,m) \quad \text{iff} \quad \forall I.\ C(FST(d\ n\ I)) = FST(\delta\ m\ I) \ \ \& \ \ (d,SND(d\ n\ I)) \approx (\delta,SND(\delta\ m\ I))$$

The property is satisfied by the relation $\approx$, where $(d,n) \approx (\delta,m)$ abbreviates the formula

$$J(d,n) = L(\delta,m)$$

As in relating $\bar{\mathcal{S}}$ to $\mathcal{D}$, we need a well-formedness property of low level stacks. We define lgood δ m to mean

$$\forall I\ m'.\ m' \leqq m \supset SND(\delta\ m'\ I) \leqq m'$$

As before, we include the well-formedness property in the relation, so that

$$(d,n) \approx (\delta,m) \quad \text{iff} \quad J(d,n) = L(\delta,m) \ \ \& \ \ \text{hgood } d\ n \ \ \& \ \ \text{lgood } \delta\ m$$

A first approximation to the theorem relating $\mathcal{D}$ and $\mathcal{R}un$ is:

$$\forall p\ d\ n\ \delta\ m\ \sigma.\ (d,n) \approx (\delta,m) \supset$$

$$THIRD(\mathcal{R}un[\![C(p)]\!]\ (m,\delta,\sigma)) = \mathcal{D}[\![p]\!]\ d\ n\ \sigma$$

which asserts that the store transformation induced by running a compiled program is the same as that produced by interpreting the original program. Looking ahead, however, if p = ⌜p1;p2⌝, we must apply $\mathcal{R}un$ to C(p2) in the configuration resulting from applying $\mathcal{R}un$ to C(p1). To apply the induction hypothesis, with p2 for p, we must know that the stack and pointer resulting from applying $\mathcal{R}un$ to C(p1) in the configuration $(m,\mathcal{S},\sigma)$ -- call them $\mathcal{S}'$ and m′ -- must be such that $(d,n) \approx (\mathcal{S}',m')$, and lgood $\mathcal{S}'$ m′. It is sufficient and convenient to show that $\mathcal{S}' \underset{m}{\equiv} \mathcal{S}$ and $m' = m$:

$$\forall p\ d\ n\ \mathcal{S}\ m\ \sigma.\ (d,n) \approx (\mathcal{S},m) \supset$$
$$\mathcal{R}un[\![C(p)]\!]\ (m,\mathcal{S},\sigma) = (m,\mathcal{S}',\ \mathcal{D}[\![p]\!]\ d\ n\ \sigma)$$
$$\text{where } \mathcal{S}' \underset{m}{\equiv} \mathcal{S}$$

However, if it is the case that $\mathcal{R}un[\![C(p)]\!](m,\mathcal{S},\sigma)$ does not terminate, then $\mathcal{R}un[\![C(p)]\!](m,\mathcal{S},\sigma) = (\bot,\bot,\bot)$, while $(m,\mathcal{S}',\ \mathcal{D}[\![p]\!]\ d\ n\ \sigma)$ is not necessarily undefined. In order to account for this possibility, we employ the notation introduced on p. 145:

<u>Theorem</u> <u>4.7</u>

$$\forall p\ d\ n\ \mathcal{S}\ m\ \sigma.\ (d,n) \approx (\mathcal{S},m) \supset$$
$$\mathcal{R}un[\![C(p)]\!]\ (m,\mathcal{S},\sigma) = \langle m,\mathcal{S}',\ \mathcal{D}[\![p]\!]\ d\ n\ \sigma\rangle$$
$$\text{where } \mathcal{S}' \underset{m}{\equiv} \mathcal{S}$$

The proof is by computation induction on $\mathcal{R}un$ and $\mathcal{D}$. (See Conclusions for further discussion of this fact.) Again, we prove a separate lemma for the letrec case, and for

convenience, for the let and call cases as well. The remainder of the proof of Theorem 4.7 is detailed but straigntforward.

We need lemmas about J and L, analogous to Lemma 4.3.

Lemma 4.8

$\forall d1\ d2\ n.\ \text{hgood}\ d1\ n \quad \& \quad d1 \underset{n}{=} d2 \supset J(d1,n) = J(d2,n)$

Lemma 4.9

$\forall \delta 1\ \delta 2\ m.\ \text{lgood}\ \delta 1\ m \quad \& \quad \delta 1 \underset{m}{=} \delta 2 \supset L(\delta 1,m) = L(\delta 2,m)$

The proofs are similar to that of Lemma 4.3.

The lemma for the letrec case is:

Lemma 4.10

$\forall p1\ d\ n\ \delta\ m\ I.\ (d,n) \approx (\delta,m) \supset (\hat{d},n+1) \approx (\hat{\delta},m+1)$

where $\hat{d} = d[(n+1) \longmapsto (d\ n)[(p1,n+1)/I]]$

and $\hat{\delta} = \delta[(m+1) \longmapsto (\delta\ m)[(C(p1),m+1)/I]]$

Proof of Lemma 4.10

We assume that $(d,n) \approx (\delta,m)$ and prove

(i) $J(\hat{d},n+1) \sqsubseteq L(\hat{\delta},m+1)$

(ii) $L(\hat{\delta},m+1) \sqsubseteq J(\hat{d},n+1)$

The proofs of hgood $\hat{d}$ (n+1) and lgood $\hat{\delta}$ (m+1) are obvious.

Proof of (i)

We prove instead

$J \sqsubseteq \text{FIX FUNJ} \quad \& \quad J(\hat{d},n+1) \sqsubseteq L(\hat{\delta},m+1)$

by computation induction on both occurrences of J, recalling that J = FIX FUNJ.

Assume

$J' \sqsubseteq J \quad \& \quad J'(\hat{d},n+1) \sqsubseteq L(\hat{\delta},m+1)$

Show

$\text{FUNJ}\ J' \sqsubseteq J \quad \& \quad \forall I'.\ C(\text{FST}(\hat{d}(n+1)I'),\ J'(\hat{d},\text{SND}(\hat{d}(n+1)I'))$

$\sqsubseteq \text{FST}(\hat{\delta}(m+1)I'),\ L(\hat{\delta},\text{SND}(\hat{\delta}(m+1)I'))$

The first conjunct is easy.

Case $I' = I$
LHS = $C(pl), J'(\hat{d}, n+1)$

RHS = $C(pl), L(\hat{\delta}, m+1)$

and the induction hypothesis applies.

Case $I' \neq I$
Since $J' \subseteq J$,

$$
\begin{aligned}
\text{LHS} &\subseteq C(FST(\hat{d}(n+1)I')), \quad J(\hat{d}, SND(\hat{d}(n+1)I')) \\
&= C(FST(d\ n\ I')), \quad J(\hat{d}, SND(d\ n\ I')) \\
&= C(FST(d\ n\ I')), \quad J(d, SND(d\ n\ I')) \quad \text{by Lemma 4.8} \\
&= J(d,n)I' \quad \text{by definition of J} \\
\text{RHS} &= FST(\hat{\delta}\ m\ I'), \quad L(\hat{\delta}, SND(\hat{\delta}\ m\ I')) \\
&= FST(\delta\ m\ I'), \quad L(\delta, SND(\delta\ m\ I')) \quad \text{by Lemma 4.9} \\
&= L(\delta, m)I' \quad \text{by definition of L}
\end{aligned}
$$

And we are finished, by the assumption that $(d,n) \approx (\delta,m)$. The proof of (ii) is similar. We prove instead

$L \subseteq$ FIX FUNL & $L(\hat{\delta}, m+1) \subseteq J(\hat{d}, n+1)$

Q.E.D.

The lemma for the let case is:

Lemma 4.11
$\forall pl\ d\ n\ \delta\ m\ I.\ (d,n) \approx (\delta,m) \supset (\hat{d}, n+1) \approx (\hat{\delta}, m+1)$

where $\hat{d} = d[(n+1) \mapsto (d\ n)[(pl,n)/I]]$

and $\hat{\delta} = \delta[(m+1) \mapsto (\delta\ m)[(C(pl),m)/I]]$

Proof of Lemma 4.11
Assume $(d,n) \approx (\delta,m)$. The proofs of hgood $\hat{d}$ (n+1) and lgood $\hat{\delta}$ (m+1) are obvious.

Show
$\forall I'.\ J(\hat{d}, n+1)I' = L(\hat{\delta}, m+1)I'$

Case $I' \neq I$
By Lemma 4.8 and Lemma 4.9 and hgood d n and lgood δ m, this reduces to showing

$\forall I'.\ J(d,n)I' = L(\delta,m)I'$

which follows from the assumption.

<u>Case $I' = I$</u>
Likewise, it is sufficient to show

$C(pl),J(d,n) = C(pl),L(\delta,m)$

and again, the result follows from the assumption. Q.E.D.

The lemma for the call case is:

<u>Lemma 4.12</u>
$\forall I.\ (d,n) \approx (\delta,m) \supset (d,n') \approx (\hat{\delta},m+1)$

where $n' = SND(d\ n\ I)$

and $\hat{\delta} = \delta[(m+1) \mapsto \delta\ m']$

where $m' = SND(\delta\ m\ I)$

<u>Proof of Lemma 4.12</u>
Assume $(d,n) \approx (\delta,m)$. The proofs of hgood d n' and lgood δ (m+1) are obvious. It is easy to show, using the assumptions and <u>Lemma 4.8</u> and <u>Lemma 4.9</u>, that for all I'

$J(d,n')I' = L(\delta,m')I'$

$= FST(\delta\ m'\ I'), L(\delta, SND(\delta\ m'\ I'))$ by definition of L

$= FST(\delta\ m'\ I'), L(\hat{\delta}, SND(\delta\ m'\ I'))$ by <u>Lemma 4.9</u> and lgood δ m'

$= FST(\hat{\delta}(m+1)I'), L(\hat{\delta}, SND(\hat{\delta}(m+1)I'))$

since $\hat{\delta}\ (m+1) = \delta\ m$ by definition of $\hat{\delta}$

$= L(\hat{\delta},m+1)$ by definition of L

Q.E.D.

The main theorem requires two more lemmas, about the strictness of $\mathcal{D}$ and $\mathcal{R}un$.

<u>Lemma 4.13</u>
$\forall q.\ \mathcal{R}un[\![q]\!] \perp \equiv \perp$

Lemma 4.14

$\forall p. \mathcal{D}[\![p]\!]\ d\ n \perp \equiv \perp$

Proofs of Lemma 4.13 and Lemma 4.14

Simple, by induction on $\mathcal{R}un$ and $\mathcal{D}$, respectively.

For convenience, we define a function $\mathcal{E}$:CONFIG $\longrightarrow$ CONFIG, such that

$$\mathcal{E}(m,\delta,\sigma) = (m-1,\delta,\sigma)$$

We also define functionals FUNR and FUND such that

$$\mathcal{R}un = \text{FIX FUNR}$$

$$\mathcal{D} = \text{FIX FUND}$$

in the obvious ways. We then prove the main theorem, Theorem 4.7, by computation induction on $\mathcal{R}un$ and $\mathcal{D}$.

As in the proof of the Russell compiler, Chapter 3, we prove the theorem in two directions, using iterated induction when doing induction on the low level semantic function, to account for the fact that each high level program is compiled into a low level program with (possibly) more than one instruction. In this case, we need 2-ary iterated induction on $\mathcal{R}un$. We prove

(i) $\forall p\ d\ n\ \delta\ m\ \sigma.\ (d,n) \approx (\delta,m) \supset$

$\mathcal{R}un[\![C(p)]\!]\ (m,\delta,\sigma)\ \sqsubseteq\ \langle m,\delta',\mathcal{D}[\![p]\!]\ d\ n\sigma\rangle$

where $\delta' \underset{m}{\equiv} \delta$

(ii) $\forall p\ d\ n\ \delta\ m\ \sigma.\ (d,n) \approx (\delta,m) \supset$

$\langle m,\delta',\mathcal{D}[\![p]\!]\ d\ n\ \sigma\rangle \sqsubseteq \mathcal{R}un\ [\![C(p)]\!]\ (m,\delta,\sigma)$

where $\delta' \underset{m}{=} \delta$

(i) is by 2-ary iterated induction on $\mathcal{R}un$, and

(ii) is by ordinary induction $\mathcal{D}$.

We prove (i) here, as the proof of (ii) is similar and easier. The four facts below, which follow from the definition of FUNR, are helpful. We let r be an arbitrary variable with the type of $\mathcal{R}un$.

<u>Lemmas 4.15</u>

$FUNR^n\ r\ [\![a]\!]\ (m,\delta,\sigma) = \langle m,\delta',\mathcal{A}[\![a]\!]\sigma\rangle$ where $\delta' \underset{m}{\equiv} \delta$

$FUNR^n\ r\ \left[\!\!\left[\begin{matrix}C(p1)\\C(p2)\end{matrix}\right]\!\!\right]\ (m,\delta,\sigma) = FUNR^{n-1}\ r\ [\![C(p2)]\!]\ (FUNR^{n-1} r\ [\![C(p1)]\!]\ (m,\delta,\sigma))$

$$FUNR^n\ r\ \left[\!\!\left[\begin{matrix}CALL(I)\\EXIT\end{matrix}\right]\!\!\right]\ (m,\delta,\sigma) = FUNR^{n-1} r\ [\![EXIT]\!]\ (FUNR^{n-1} r\ [\![CALL(I)]\!]\ (m,\delta,\sigma)) = \mathcal{E}\,(FUNR^{n-1}\ r\ [\![CALL\ (I)]\!]\,(m,\delta,\sigma))$$

$$FUNR^n r \left[\!\!\left[\begin{matrix}PRENTRY(I,C(p1))\\C(p2)\\EXIT\end{matrix}\right]\!\!\right]\ (m,\delta,\sigma) = FUNR^{n-1} r\ [\![EXIT]\!]\ (FUNR^{n-2} r\ [\![C(p2)]\!]\ (FUNR^{n-2} r\ [\![PRENTRY(I,C(p1))]\!]\ (m,\delta,\sigma)))$$

$$=\mathcal{E}(\text{FUNR}^{n-2}r\ [\![C(p2)]\!]\ (\text{FUNR}^{n-2}r\ [\![\text{PRENTRY}(I,C(p1))]\!]\ (m,\delta,\sigma)))$$

$$\text{FUNR}^{n}\ r \begin{bmatrix} \text{RECENTRY}(I,C(p1)) \\ C(p2) \\ \text{EXIT} \end{bmatrix} (m,\delta,\sigma) =$$

$$\text{FUNR}^{n-1}r\ [\![\text{EXIT}]\!]\ (\text{FUNR}^{n-2}r\ [\![C(p2)]\!]\ (\text{FUNR}^{n-2}r\ [\![\text{RECENTRY}(I,C(p1))]\!]\ (m,\delta,\sigma)))$$

$$=\mathcal{E}(\text{FUNR}^{n-2}r\ [\![C(p2)]\!]\ (\text{FUNR}^{n-2}r\ [\![\text{RECENTRY}(I,C(p1))]\!]\ (m,\delta,\sigma)))$$

These lemmas unfold FUNR for us, for the various shapes of compiled programs.

For the proof of the main theorem, iterated induction entails proving two basis cases, and a step with two hypotheses. We let w be the formula (i) above.

<u>BASIS1</u>

$w[\bot / \mathit{Run}]$

<u>BASIS2</u>

$w[\text{FUNR}\ \bot / \mathit{Run}]$

<u>IH1</u>

$w[r/\mathit{Run}]$

<u>IH2</u>

$w[\text{FUNR}\ r/\mathit{Run}]$

<u>STEP</u>

$w[\text{FUNR}^{2} r/\mathit{Run}]$

The proof of <u>BASIS1</u> is obvious. For <u>BASIS2</u>, we use <u>Lemma 4.15</u>, with

for r, and 1 for n. We show that the antecedent of w implies

$$\text{FUNR } \perp [\![C(p)]\!] \; (m,\mathcal{S},\sigma) \;\sqsubseteq\; \langle m,\mathcal{S}', \mathcal{D}[\![p]\!] \; d \; n\sigma\rangle$$

where $\mathcal{S}' \equiv_{m} \mathcal{S}$

In the atomic case, both sides are the same. Otherwise, Lemma 4.13 can be used to show that the left hand side is $\perp$.

Proof of Theorem 4.7 (i) STEP

We consider the various cases for p. We assume IH1 and IH2, and the antecedent of w, namely, $(d,n) \approx (\mathcal{S},m)$ and show

$$\text{FUNR } {}^{2}r \; [\![C(p)]\!] \; (m,\mathcal{S},\sigma) \;\sqsubseteq\; \langle m,\mathcal{S}', \mathcal{D}[\![p]\!] \; d \; n\sigma\rangle$$

where $\mathcal{S}' \equiv_{m} \mathcal{S}$

Case $p = \ulcorner a \urcorner$

Obvious, by the definition of $\mathcal{D}$.

Case $p = \ulcorner p1;p2 \urcorner$

$$\text{LHS} = \text{FUNR } r \; [\![C(p2)]\!] \, (\text{FUNR } r \; [\![C(p1)]\!] \; (m,\mathcal{S},\sigma))$$

while $\mathcal{D}[\![p]\!] \; d \; n\sigma \; = \lambda\sigma.\mathcal{D}[\![p2]\!] \; d \; n \; (\mathcal{D} \; [\![p1]\!] \; d \; n\sigma)$, so that

$$\text{RHS} = \langle m,\mathcal{S}', \mathcal{D}[\![p2]\!] \; d \; n \; (\mathcal{D} \; [\![p1]\!] \; d \; n\sigma)\rangle$$

Applying IH2, we get

$$\text{FUNR } r \; [\![C(p1)]\!] \; (m,\mathcal{S},\sigma) \;\sqsubseteq\; \langle m,\mathcal{S}', \mathcal{D}[\![p1]\!] \; d \; n\sigma\rangle$$

where $\mathcal{S}' \equiv_{m} \mathcal{S}$.

Thus $\text{LHS} = \text{FUNR } r \; [\![C(p2)]\!] \; \langle m,\mathcal{S}', \mathcal{D}[\![p1]\!] \; d \; n\sigma\rangle$

Case $\mathcal{D}[\![p1]\!] \; d \; n\sigma = \perp$

Then by Lemma 4.13, LHS $= \perp$, and by Lemma 4.14, RHS $= \perp$.

Case $\mathcal{D}[\![p1]\!] \; d \; n\sigma \neq \perp$

To use the induction hypothesis again, we must show that lgood $\mathcal{S}'$ m and that $J(d,n) = L(\mathcal{S}',m)$, both of which are easy, by assumption, and Lemma 4.9. By IH2, again, we have

$$\text{FUNR } r \; [\![C(p2)]\!] \; (m,\mathcal{S}', \mathcal{D}[\![p1]\!] \; d \; n\sigma) \;\sqsubseteq$$

$$\langle m,\mathcal{S}'', \mathcal{D}[\![p2]\!] \; d \; n \; (\mathcal{D} \; [\![p1]\!] \; d \; n\sigma)\rangle \quad \text{where } \mathcal{S}'' \equiv_{m} \mathcal{S}'$$

and since $\mathcal{S}'' \equiv_{m} \mathcal{S}' \equiv_{m} \mathcal{S}$, this is enough.

Lemma 4.10, Lemma 4.11 and Lemma 4.12 are used in the remaining

three cases.

Case $p = \ulcorner$call I$\urcorner$
LHS $= \mathcal{E}$ (FUNR r $[\![$CALL(I)$]\!]$ (m,δ,σ))

$= \mathcal{E}$ (r $[\![q']\!]$ $(m+1,\hat{\delta},\sigma)$)

where $(q',n') = \delta$ m I

and $\hat{\delta} = \delta[(m+1) \mapsto \delta\ m']$

while $\mathcal{D}[\![$call I$]\!]$ d n $= \mathcal{D}[\![p']\!]$ d n' where $(p',n') =$ d n I,

so RHS $= \langle m, \delta', \mathcal{D}[\![p']\!]\ d\ n'\sigma \rangle$

By the fact that $J(d,n) = L(\delta,m)$, we know that $q' = C(p')$. We can then use Lemma 4.12, which allows us to apply the induction hypothesis (IH1, this time), to get

r $[\![C(p')]\!]$ $(m+1,\hat{\delta},\sigma) \sqsubseteq \langle m+1,\delta'', \mathcal{D}[\![p']\!]\ d\ n'\sigma \rangle$

where $\delta'' \underset{m+1}{\equiv} \hat{\delta}$.

Case $\mathcal{D}[\![p']\!]$ d $n'\sigma = \bot$
Then LHS = RHS = $\bot$ (see Note 4)

Case $\mathcal{D}[\![p']\!]$ d $n'\sigma = \bot$
LHS $\sqsubseteq (m,\delta'', \mathcal{D}[\![p']\!]\ d\ n'\sigma)$, by the definition of $\mathcal{E}$, and

RHS $= (m,\delta', \mathcal{D}[\![p']\!]\ d\ n'\sigma)$. Since $\delta'' \underset{m+1}{\equiv} \hat{\delta} \underset{m}{\equiv} \delta' \underset{m}{\equiv} \delta$,
we are finished.

Case $p = \ulcorner$let I = p1 in p2$\urcorner$
LHS $= \mathcal{E}$ (FUNR r $[\![C(p2)]\!]$ (FUNR r $[\![$PRENTRY(I,C(p1))$]\!]$ (m,δ,σ)))

$= \mathcal{E}$ (FUNR r $[\![C(p2)]\!]$ $(m+1,\hat{\delta},\sigma)$)

with $\hat{\delta}$ as in Lemma 4.11, while

RHS $= \langle m,\delta', \mathcal{D}[\![p2]\!]\ \hat{d}\ (n+1)\sigma \rangle$

where $\hat{d}$ is as in Lemma 4.11, and $\delta' \underset{m}{\equiv} \delta$.

Using Lemma 4.11, we can apply the induction hypothesis, IH2, to obtain

FUNR r $[\![C(p2)]\!]$ $(m+1,\hat{\delta},\sigma) \sqsubseteq \langle m+1,\delta'', \mathcal{D}[\![p2]\!]\hat{d}\ (n+1)\sigma \rangle$

where $\delta'' \underset{\bar{m}}{\equiv} \hat{\delta}$.

Case $\mathcal{D}[\![p2]\!]$ d $(n+1)\sigma = \bot$
LHS = RHS = $\bot$

Case $\mathcal{D}[\![p2]\!]\ d\ (n+1)\sigma \neq \bot$
LHS $\sqsubseteq$ (m, δ'', $\mathcal{D}[\![p2]\!]\hat{d}\ (n+1)\sigma$), so by definition of $\mathcal{E}$ and and the fact that $\delta'' \underset{m+1}{=} \hat{\delta} \underset{m}{=} \delta$, we are finished.

Case p = ⌜letrec I = p1 in p2⌝
The proof is much the same as the let proof, using Lemma 4.10. We omit it.

The proof of (ii) is similar to the proof of (i), using ordinary rather than iterated induction. This completes the equivalence proof of $\mathcal{D}$ and *Run*. Q.E.D.

Summary

This completes the sequence of proofs relating $\mathcal{S}$, a standard denotational semantics for a block structured high level language, to *Run*, a stack implementation of an assembly-like language, into which the high level language is compiled. The stages into which the compilation and proof have been divided include $\bar{\mathcal{S}}$, a closure semantics, and $\mathcal{D}$, an abstract stack semantics, both for the high level language. The types of the semantic functions are:

$\mathcal{S}$: HPROGRAM ⟶ HENV ⟶ STORE ⟶ STORE

where $\rho \in$ HENV = ID ⟶ STORE ⟶ STORE

$\bar{\mathcal{S}}$: HPROGRAM ⟶ HCENV ⟶ STORE ⟶ STORE

where v ∈ HCENV = ID ⟶ (HPROGRAM × HCENV)

$\mathcal{D}$: HPROGRAM ⟶ HAS ⟶ INT ⟶ STORE ⟶ STORE

where d ∈ HAS = INT ⟶ DENV

where DENV = ID ⟶ (HPROGRAM × INT)

Run: LPROGRAM ⟶ CONFIG ⟶ CONFIG

where CONFIG = INT × LAS × STORE

where $\delta \in$ LAS = INT $\longrightarrow$ LENV

where LENV = ID $\longrightarrow$ (LPROGRAM $\times$ INT)

The relations between the levels are:

From $\mathcal{S}$ to $\bar{\mathcal{S}}$:

$\rho \sim v$ iff $\forall I.\ \rho I = \bar{\mathcal{S}}[\![\underline{\text{call}}\ I]\!]\, v$

From $\bar{\mathcal{S}}$ to $\mathcal{D}$:

$v \approx (d,n)$ iff $v = H(d,n)$ & hgood d n

where $H = FIX(\lambda H'\ (d,n)\ I.FST(d\ n\ I),\ H'(d,SND(d\ n\ I)))$

From $\mathcal{D}$ to *Run*:

$(d,n) \approx (\delta,m)$ iff $J(d,n) = L(\delta,m)$ &

hgood d n & lgood δ m

where $J = FIX(\lambda J'\ (d,n)\ I.C(FST(d\ n\ I)),\ J'(d,SND(d\ n\ I)))$

and $L = FIX(\lambda L'\ (\delta,m)\ I.\ FST(\delta\ m\ I),\ L'(\delta,SND(\delta\ m\ I)))$

and hgood d n iff $\forall I\ n'.\ n' \leqq n \supset SND(d\ n'\ I) \leqq n'$

and lgood δ m iff $\forall I\ m'.\ m' \leqq m \supset SND(\delta\ m'\ I) \leqq m'$

The three theorems are:

Theorem 4.1

$\forall p\ \rho\ v.\ \rho \sim v \supset \mathcal{S}[\![p]\!]\rho = \bar{\mathcal{S}}[\![p]\!]\, v$

By structural induction on p, with an inner (parallel) computation induction in the letrec case.

Theorem 4.6

$\forall p\ v\ d\ n.\ v \approx (d,n) \supset \bar{\mathcal{S}}[\![p]\!]\, v = \mathcal{D}[\![p]\!]\, d\ n$

By parallel computation induction on $\bar{\mathcal{S}}$ and $\mathcal{D}$. A separate lemma (Lemma 4.5) is required for the letrec case:

Lemma 4.5
$\forall$ pl v d n. v $\approx$ (d,n) $\supset$ $\hat{v}$ $\approx$ ($\hat{d}$,n+1)

where $\hat{v}$ and ($\hat{d}$,n+1) are, respectively, the contexts in which the recursive declarations have been made. The proof of the lemma is in two directions, $\sqsupseteq$ by computation induction on H.

Theorem 4.7
$\forall$ p d n $\mathcal{S}$ m σ. (d,n) $\approx$ ($\mathcal{S}$,m) $\supset$

$\mathcal{Run}$ $[\![$C(p)$]\!]$ (m,$\mathcal{S}$,σ) = <m, $\mathcal{S}'$, $\mathcal{D}$ [p] d n σ>

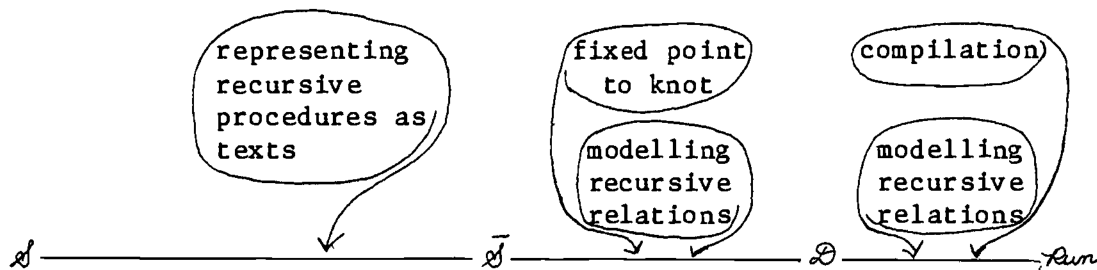
where $\mathcal{S}' \underset{m}{=} \mathcal{S}$

By computation induction on $\mathcal{Run}$ and $\mathcal{D}$, in two directions. In inducting on $\mathcal{Run}$, we use 2-ary iterated induction. A separate induction is required for the letrec case (Lemma 4.10).

Lemma 4.10
$\forall$ pl d n $\mathcal{S}$ m. (d,n) $\approx$ ($\mathcal{S}$,m) $\supset$ ($\hat{d}$,n+1) $\approx$ ($\hat{\mathcal{S}}$,m+1)

where ($\hat{d}$,n+1) and ($\hat{\mathcal{S}}$,m+1) are the respective contexts in which recursive procedures and the compiled images of recursive procedures have been declared. The proof is in two directions by computation induction on J and L respectively.

The major complications in the proof occur at the following points in the transition:



Representing procedure values as texts is straightforward in this case. The heart of the proof, in a sense, is in the transition from

$\bar{\mathcal{S}}$ to $\mathcal{D}$, as this stage involves the essential change from a fixed point semantics for recursive procedures to a 'knotted' one. We manage to avoid the use of recursively defined relations in the two instances indicated by constructing well-defined non-recursive ones which satisfy the desired recursive properties. 'Incompatible' pairs of semantics, in the process, are related by the use of 'abstracting' functions (H,J and L). Compilation appears to offer no special difficulties in this proof, aside from technical ones (the need for iterated induction, and the use of strict tripling in the statement of equivalence).

Speculations on Performing the Proof in LCF

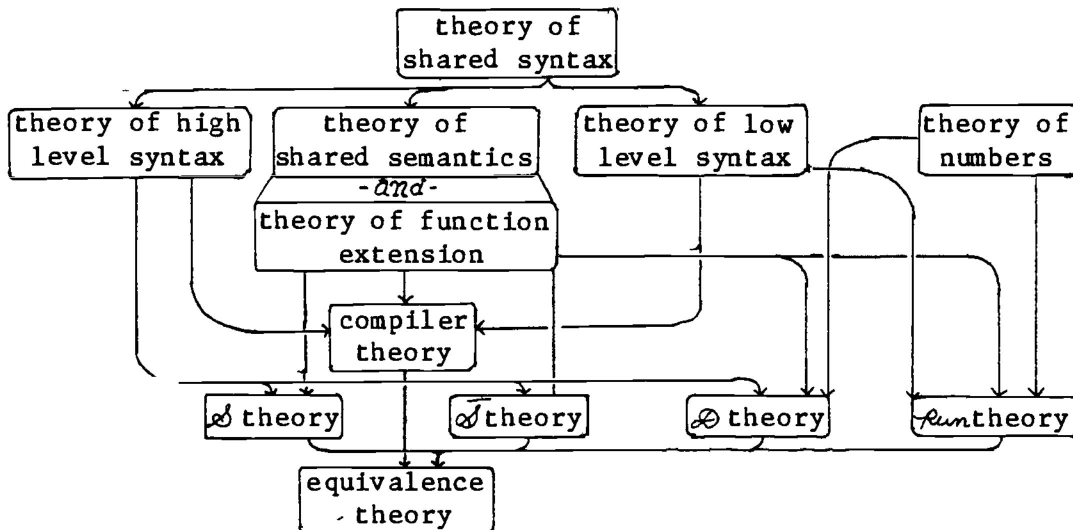
As we indicated at the outset, the emphasis in the work described in this chapter has been to develop theories which could be formalised and proofs which could be generated in LCF. Although we have not in fact done the proofs mechanically, the successful machine proof effort described in Chapter 3, the informal proofs sketched in this chapter, and the remarks below lead us to believe that a machine proof would be a feasible undertaking.

Theory Structure for the Proofs

The proof effort would proceed in much the same way as the previous one; we would construct a network of LCF theories in which to work, including a theory of the syntax common to the high and low level languages (e.g. atomic statements), and theories of the high

and low level syntax; theories of the shared semantics, and of the semantics $\mathscr{S}$, $\bar{\mathscr{S}}$, $\mathscr{D}$, and $\mathscr{Run}$; a compiler theory; and a theory of equivalence. We would also require a theory of natural numbers (as in Chapter 2, but with the additional constant $\leq$, and relevant axioms). A polymorphic theory of function extension (as in Chapter 3) would be useful for reasoning about extensions to high and low level activation stacks, and to environments of various sorts.

The network of theories for the proof might be:



The theories of function extension and of numbers are quite independent from the compiler problem; the others are specific to it.

The theory of high level syntax would inherit the new types ID (for identifiers) and A (for atoms) from the theory of shared syntax, and would include the new recursive type HPROGRAM, defined by two new constants:

ABSHPROGRAM:(ATOM u + CALL u + LET u + LETREC u + SEQUENCE u)
$\longrightarrow$ HPROGRAM

REPHPROGRAM:HPROGRAM $\longrightarrow$
(ATOM u + CALL u + LET u + LETREC u + SEQUENCE u)

The other types are defined as

CALL = ID

LET = ID $\times$ HPROGRAM $\times$ HPROGRAM

LETREC = ID $\times$ HPROGRAM $\times$ HPROGRAM

SEQUENCE = HPROGRAM $\times$ HPROGRAM

and axioms would be added about the representation and abstraction functions:

$\vdash \forall \alpha.$ REPHPROGRAM(ABSHPROGRAM α) $\equiv \alpha$

$\vdash \forall$p. ABSHPROHRAM(REPHPROGRAM p) $\equiv$ p

Other new constants would include:

mkcall:CALL $\longrightarrow$ HPROGRAM

destcall:HPROGRAM $\longrightarrow$ CALL

iscall:HPROGRAM $\longrightarrow$ tr

callidof:CALL $\longrightarrow$ ID

with axioms

$\vdash$ mkcall $\equiv$ ABSHPROGRAM o INR o INL o UP

$\vdash$ destcall $\equiv$ DOWN o OUTL o OUTR o REPHPROGRAM

$\vdash$ iscall $\equiv$ ISL o OUTR o REPHPROGRAM

$\vdash \forall$c. callid c $\equiv$ c

The theory of $\mathcal{S}$, for example, would have as parents the theories of high level syntax, shared semantics and function extension,

inheriting from the second the type STORE and from the third, the constant extend. High level closure environments would be introduced as a new recursive type, defined by two new constants:

$$\text{ABSHCENV}:(\text{ID} \longrightarrow (\text{HPROGRAM} \times \text{HCENV})) \longrightarrow \text{HCENV}$$

$$\text{REPHCENV}:\text{HCENV} \longrightarrow (\text{ID} \longrightarrow (\text{HPROGRAM} \times \text{HCENV}))$$

which are axiomatised by:

$$\vdash \forall\alpha.\ \text{REPHCENV}(\text{ABSHCENV}\,\alpha) \equiv \alpha$$

$$\vdash \forall v.\ \text{ABSHCENV}(\text{REPHCENV}\ v) \equiv v$$

We would introduce a constant for the semantic function:

$$\bar{\mathcal{S}}:\text{HPROGRAM} \longrightarrow \text{HCENV} \longrightarrow \text{STORE} \longrightarrow \text{STORE}$$

and an axiom defining it, of the form

$$\begin{aligned}\vdash \bar{\mathcal{S}} \equiv \text{FIX}(\lambda\bar{\mathcal{S}}'\ \ p\ v.\ & \ldots \Rightarrow \ldots \mid \\ & \ldots \Rightarrow \ldots \mid \\ & \text{iscall } p \Rightarrow \bar{\mathcal{S}}'\ (\text{FST}((\text{REPHCENV } v) \\ & \quad (\text{callidof}(\text{destcall } p)))) \\ & \qquad (\text{SND}((\text{REPHCENV } v) \\ & \quad (\text{callidof}(\text{destcall } p)))) \mid \\ & \ldots \Rightarrow \ldots \mid \\ & \ldots \Rightarrow \ldots \mid \bot)\end{aligned}$$

(where we have shown only the call case). From this we can easily prove facts of the following form, which we would then store in the theory:

$$\begin{aligned}\vdash \bar{\mathcal{S}}(\text{mkcall I})\ v \equiv \bar{\mathcal{S}}\,&(\text{FST}((\text{REPHCENV } v)\text{I})) \\ &(\text{SND}((\text{REPHCENV } v)\text{I}))\end{aligned}$$

and similarly for the other cases. Having both formulations allows us to use computational or structural induction as necessary.

Tactics for the Proofs

By examining the patterns of inference which occur in the informal proofs, it is possible to suggest tactics to assist in generating the proof mechanically. Aside from tactics discussed in the previous chapters, the following reflect the main patterns of reasoning in the three proofs discussed in this chapter (and the various lemmas). The proofs, of course, would be performed in the equivalence theory, so that all types, constants, axioms and theorems from the other theories were available.

Firstly, we require a tactic, IMPTAC, for proving goals whose formulae are implications by assuming the antecendent and returning the consequent as a subgoal:

IMPTAC

$$\frac{(\text{w1 IMP w2},\ \ \text{ss},\ \ \text{A})}{(\text{w2},\ \ \text{ss},\ \ \text{w1.A})}$$

The proof part would use the PPLAMBDA inference rule DISCH (see [15], A5) to discharge the extra hypothesis of the theorem achieving the subgoal. This tactic would be of use, for example, in proving Lemma 4.2, the step of Theorem 4.1, BASIS2 and the STEP of Theorem 4.7, and several other theorems in this chapter. Some calls of IMPTAC would have to be followed by applications of CONJASSUMPTAC (Chapter 2, p. 81), as the antecedents are conjunctions; for example, Lemma 4.4:

$$v \approx (d,n) \quad \& \quad m \leqq n{+}1 \quad \supset \quad v[(p1,H(d,m))/I] \approx (\hat{d},n{+}1)$$

A tactic for induction, simpler than the standard INDUCTAC, is useful; for example, to prove the letrec case of Theorem 4.1, we must show that

$$\mathcal{S}[\![p2]\!]\ (FIX(\lambda\rho'.\rho[\mathcal{S}[\![p1]\!]\ \rho'/I])) =$$
$$\bar{\mathcal{S}}[\![p2]\!]\ (FIX(\lambda v'.\ v[(p1,v')/I]))$$

and so would like

SIMPLEINDUCTAC

w[(FIX funi)/fi]
ss
A

---

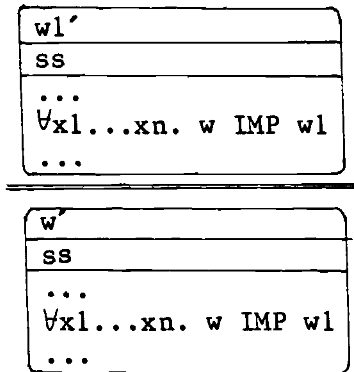
w[⊥/fi]
ss
A

w[(funi fi')/fi]
ss
(w[fi'/fi])
A

which, unlike the standard tactic, would not take recursive function definitions as parameters, or finish by substituting according to those definitions.

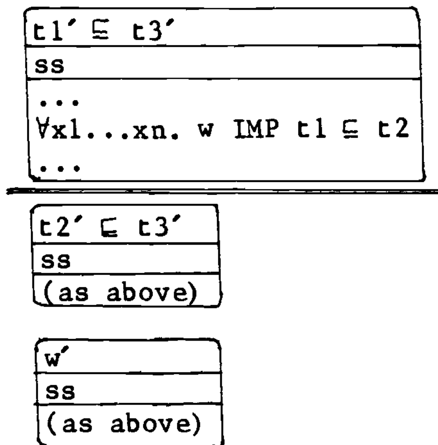
In several instances, for example Theorem 4.6, we must prove implications by induction, so that in the course of the proofs, the induction hypotheses can be instantiated to arbitrary variables for which the antecedents hold. In these cases, IMPTAC is not adequate. Instead, we must do induction (of the appropriate sort) on a formula which is an implication. When we wish to apply the induction hypothesis, we write and call the following tactic:

USEIMPASSUMPTAC



where wl, with some instantiations, matches wl′, and w′ is w with these instantiations. The proof part is similar to that of USEASSUMPTAC (Chapter 2, p. 48), except that here, the inference rule expressing Modus Ponens (called MP, see [15], A5) is used to obtain the theorem achieving the goal. Where necessary we would also use

USEIMPASSUMPLHSTAC



where t2′ is t2 with the instantiations for xl,...,xn determined by matching tl to tl′, and w′ is w with the same instantiations made. That is, USEIMPASSUMPLHSTAC is like USEASSUMPLHSTAC (Chapter 2, p. 68) except that the assumption to be used is conditionalised on some

formula. That formula, appropriately instantiated, is returned as part of a subgoal. The intermediate subgoal whose formula is $t2' \sqsubseteq t3'$ is also returned as a subgoal. In the proof, the formula (the implication $\forall x1...xn.$ w IMP $t1 \sqsubseteq t2$) is assumed, and then specialised to the match; then the PPLAMBDA inference rule MP is applied to the result and the theorem achieving the second subgoal. Transitivity is applied to the result of _that_ and the theorem achieving the first subgoal.

One would also want the dual tactic, USEIMPASSUMPRHSTAC, whose definition is analogous.

Another important pattern of inference in the proofs occurs, for example, in the proofs of _Lemma 4.5_ and _Lemma 4.10_. In both situations we would first apply SYNTHTAC (Chapter 2, p. 61) to obtain two subgoals, and then apply (for _Lemma 4.5_, to the second subgoal thereby obtained, and for _Lemma 4.10_, to both subgoals) a tactic based on the following derived rule of inference:

FIXPTRULE

$$\frac{\vdash w[\bot/f] \qquad\qquad [w[f'/f];\ f' \sqsubseteq \text{FIX FUN}] \vdash w[(\text{FUN } f')/f]}{\vdash w[(\text{FIX FUN})/f]}$$

(This corresponds to the informal induction rule discussed in Note 3.) The tactic which inverts it is:

FIXPTTAC ($\vdash$f $\equiv$ FIX FUN)

$$\frac{(w,\ ss,\ A)}{}$$

w[⊥ /f]
ss
A

w[(FUN f′)/f]
ss
f′ ⊆ FIX FUN w[f′/f] A

The rule FIXPTRULE conceals an ordinary induction on f' in the formula

$$f' \sqsubseteq \text{FIX FUN} \quad \& \quad w[f'/f]$$

The proof of the basis case is obvious. The first conjunct of the step depends on the fact that

$$\text{FIX FUN} \equiv \text{FUN (FIX FUN)}$$

which is expressed by the PPLAMBDA rule FIXPT (see [15], A5). The second conjunct of the step is returned as a subgoal by FIXPTTAC. The proof part of the tactic calls FIXPTRULE.[5]

A related rule and tactic (as suggested by the alternative proof of Theorem 4.1 (see Note 2.) are FIXFUNRULE and FIXFUNTAC:

FIXFUNRULE

$$\frac{\vdash w[\bot/f] \qquad\qquad [w[f'/f];\ f \sqsubseteq \text{FUN}\ f'] \vdash w[(\text{FUN}\ f')/f]}{\vdash w[(\text{FIX FUN})/f]}$$

FIXFUNTAC ($\vdash$ f $\equiv$ FIX FUN)

$$\frac{(w,\ ss,\ A)}{\begin{array}{|l|}\hline w[\bot/f] \\ \hline ss \\ \hline A \\ \hline \end{array} \quad \begin{array}{|l|}\hline w[(FUN\ f')/f] \\ \hline ss \\ \hline f' \subseteq FUN\ f' \\ w[f'/f] \\ A \\ \hline \end{array}}$$

In the rule, INDUCT is called on f$'$, in the formula

$$f' \subseteq FUN\ f' \quad \& \quad w[f'/f]$$

The proof part of the tactic calls the rule.

Another tactic related to FIXPTTAC, useful for proving Lemma 4.5 (i), is:

LFPTAC ($\vdash$ f $\equiv$ FIX FUN)

$$\frac{(f \subseteq g,\ ss,\ A)}{(g \equiv FUN\ g,\ \ ss,\ \ A)}$$

LFPTAC, for least fixed point tactic, proves f $\subseteq$ g by showing that g is *a* fixed point of FUN, and is therefore greater than the least fixed point of FUN, which is f. The proof part of LFPTAC does induction on x in the formula x $\subseteq$ g, proving the basis case internally, and proving the step by assuming that x $\subseteq$ g, applying FUN to both sides to get FUN x $\subseteq$ FUN g, and using induction to conclude that FIX FUN $\subseteq$ g, that is, f $\subseteq$ g. This tactic is often useful.

The tactics MINFIXTAC (Chapter 2, p. 68), UNFOLDTAC (Chapter 2, p. 64), FIXPTTAC, FIXFUNTAC and LFPTAC belong to a class of tactics which use properties of the least fixed point operator to divide goals into subgoals. They reflect various ways of reasoning about recursively defined functions.

Another useful tactic for these proofs is IDCASESTAC, similar to NATCASESTAC (Chapter 2, p. 80), for performing case analysis on the equality of two identifiers (which it finds) in a formula:

IDCASESTAC

w[I/t1][J/t2]
ss
A

═══════════════

w[I/t1][J/t2]
(EQ I J ≡ TT) + ss
EQ I J ≡ TT A

w[I/t1][J/t2]
(EQ I J ≡ FF) + ss
EQ I J ≡ FF A

w[I/t1][J/t2]
(EQ I J ≡ ⊥) + ss
EQ I J ≡ ⊥ A

This is used in numerous places in the proofs.[6]

A tactic (HINDUCTAC) and a tactic (HCASESTAC), to do induction and case analysis, respectively, on the structure of high level programs, analogous to those suggested in Chapter 3, could be written in ML. They are depicted as:

HINDUCTAC

```
(w, ss, A)
================================================================
[w[⊥/p]]          [w[let I = p1 in p2/p]]
[ss    ]          [ss                   ]
[A     ]          [w[p1/p]              ]
                  [w[p2/p]              ]
                  [A                    ]

[w[a/p]]          [w[letrec I = p1 in p2/p]]
[ss    ]          [ss                      ]
[A     ]          [w[p1/p]                 ]
                  [w[p2/p]                 ]
                  [A                       ]

[w[call I/p]]     [w[p1;p2/p]]
[ss         ]     [ss        ]
[A          ]     [w[p1/p]   ]
                  [w[p2/p]   ]
                  [A         ]
```

The cases tactic required is

HCASESTAC

```
(w, ss, A)
================================================================
[w[⊥/p]]          [w[let I = p1 in p2/p]]
[ss    ]          [ss                   ]
[A     ]          [A                    ]

[w[a/p]]          [w[letrec I = p1 in p2/p]]
[ss    ]          [ss                      ]
[A     ]          [A                       ]

[w[call I/p]]     [w[p1;p2/p]]
[ss         ]     [ss        ]
[A          ]     [A         ]
```

which is similar to the induction tactic, but does not add induction hypotheses to the subgoals. These tactics are derived from the (obvious) derived rules HINDUCT and HCASES, respectively.

We would expect the composite tactics required for the theorems and lemmas in this chapter to be similar to those used in Chapter 3. For example, a tactic which reflects the informal proof of Theorem 4.11 would begin with SYNTHTAC (like the tactic for proving Theorem 3.1 b) to produce subgoals for the two directions. For the easy direction, we would then apply

```
(INDUCTAC [thRun])+
GENTAC*
IMPTAC
IMPCONJTAC
HCASESTAC+
USEIMPASSUMPLHSTAC+
CONDCASESTAC+
USEIMPASSUMPLHSTAC+
```

where thRun is the theorem defining Run as the least fixed point of FUNR. Aside from the addition of IMPTAC and IMPCONJTAC (since we are dealing with an implicative formula) and the associated use of USEIMPASSUMPLHSTAC rather than USEASSUMPLHSTAC, the tactic has much the same shape as the previous COMPILERTAC. Of course, this is not altogether surprising, as many proofs are done by induction, specification, case analysis and use of induction hypothesis, but it is reassuring.

Armed with this set of tactics, as well as those already derived in other chapters, it would appear that the proofs in this chapter could be performed in LCF without great difficulty. A minor problem is that one cannot use predicate constants or variables within PPLAMBDA (as one can use function constants or variables), so that the predicates and relations such as $\sim$, $\approx$ and $\approxeq$ (and, in turn, hgood and lgood) would have to appear as the formulae they

abbreviate. This produces rather cumbersome goals and theorems. Nonetheless, we conjecture that the proofs in this chapter could be generated in LCF with a certain investment of effort in the programming of tactics and the formulation of theories, as sketched here. Our optimism is based on the successful generation of the proof of the Russell compiler, and on the speculations in this section. We intend to undertake the proof effort and present the results in subsequent reports.

Notes for Chapter 4

1. In [25], Milne extensively treats a sequence of increasingly concrete semantics for a language called Sal, and a low level language, Sam, into which Sal programs are compiled. Our *Run* corresponds, in level of abstractness, roughly to his stack semantics. Further on in the sequence are 'consecution' and pointer semantics. Eventually, all functional (infinite) objects are replaced by integers or other concrete objects. In regard to modelling displays, Milne proposes (ibid., p. 729) a model in which displays represent static chains. In [24] he discusses the representation of identifiers in a low level language, as integer offsets, and integers as numerals. He also gives a treatment of procedure invocations as jumps [24].

More concrete models of implementations have also been discussed by Aiello, Aiello and Weyhrauch [1], in the context of a model, in Stanford LCF, of the implementation of a subset of PASCAL. In this work, they formalise and reason about the notion of frames (activation records) in which control and access links (dynamic and static links) and binding information for local variables, is represented. Finally, Newey [38] has worked on the problem of modelling an assembly language and register machine in an LCF-like setting.

2. Alternatively, the proof of $\mathcal{S}$ to $\bar{\mathcal{S}}$ can be done by computation induction on $\mathcal{S}$ and $\bar{\mathcal{S}}$. The proof is complicated by the occurence of one of the induction variables ($\bar{\mathcal{S}}$) in the antecedent of the formula to be proved:

$$\forall I. \rho\ I \sqsubseteq \bar{\mathcal{S}}[\![\underline{\text{call}}\ I]\!]\,v \supset \mathcal{S}[\![p]\!]\rho \sqsubseteq \bar{\mathcal{S}}[\![p]\!]\,v$$

As a result, we prove the theorem as a pair of inequivalences, where $\bar{\mathcal{S}} \equiv$ FIX FUN$\bar{\mathcal{S}}$:

$$\text{(i)}\ \forall I. \rho\ I \sqsubseteq \bar{\mathcal{S}}[\![\underline{\text{call}}\ I]\!]\,v \supset \mathcal{S}[\![p]\!]\rho \sqsubseteq \bar{\mathcal{S}}[\![p]\!]\,v$$

$$\text{(ii)}\ \forall I. \bar{\mathcal{S}}[\![\underline{\text{call}\ I}]\!]\,v \sqsubseteq \rho\ I\ \ \&$$

$$\bar{\mathcal{S}} \sqsubseteq \text{FUN}\bar{\mathcal{S}} \supset \bar{\mathcal{S}}[\![p]\!]\,v \sqsubseteq \mathcal{S}[\![p]\!]\rho$$

(i) is by computation induction on $\mathcal{S}$. (ii) is by appeal to the following rule of induction:

$$w[\bot/f]\ \ \&\ \ ((w\ \ \&\ \ f \sqsubseteq \text{fun } f) \supset w[(\text{fun } f)/f]) \supset$$

$$w[(\text{FIX fun})/f]$$

To show that the rule is valid, we do induction on all three occurrences of f in the formula

$$w\ \ \&\ \ f \sqsubseteq \text{fun } f$$

$f \sqsubseteq$ fun f implies that fun $f \sqsubseteq$ fun fun f, and the rest is straightforward. To do the proof using the new induction rule:

<u>Assume</u>
$\forall I.\ \bar{\mathcal{S}}'[\![\underline{\text{call}}\ I]\!]\ v \sqsubseteq \rho\ I \supset \bar{\mathcal{S}}'[\![p]\!]\ v \sqsubseteq \mathcal{S}[\![p]\!]\rho$

<u>Assume</u>
$\bar{\mathcal{S}}' \sqsubseteq \text{FUN}\mathcal{S}\ \bar{\mathcal{S}}'$

<u>Assume</u>
$\forall I.\ \text{FUN}\mathcal{S}\ \bar{\mathcal{S}}'[\![\underline{\text{call}}\ I]\!]\ v \sqsubseteq \rho\ I$

that is,
$\forall I.\ \bar{\mathcal{S}}'\ [\![\text{FST}(v\ I)]\!]\ (\text{SND}(v\ I)) \sqsubseteq \rho\ I$

<u>Show</u>
$\text{FUN}\mathcal{S}\ \bar{\mathcal{S}}'[\![p]\!]\ v \sqsubseteq \mathcal{S}[\![p]\!]\rho$

for the various cases of p.

For example, if p = ⌜<u>let</u> I = p1 <u>in</u> p2⌝, we must show

$\bar{\mathcal{S}}'[\![p2]\!]\ \hat{v} \sqsubseteq \mathcal{S}[\![p2]\!]\hat{\rho}$

where $\hat{\rho}$ and $\hat{v}$ are FIX $\mathcal{R}$ and FIX $\mathcal{V}$, as in the structural induction proof. This, in turn, requires that

$\forall I'.\ \bar{\mathcal{S}}'\ [\![\underline{\text{call}}\ I']\!]\ \hat{v} \sqsubseteq \hat{\rho}\ I'$

By the assumptiom, it is sufficient to show that

$\text{FUN}\mathcal{S}\ \bar{\mathcal{S}}'[\![\underline{\text{call}}\ I']\!]\ \hat{v} \sqsubseteq \hat{\rho}\ I'$

that is,
$\bar{\mathcal{S}}'\ [\![\text{FST}(\hat{v}\ I')]\!]\ (\text{SND}(\hat{v}\ I')) \sqsubseteq \hat{\rho}\ I'$

Where $I \neq I'$, the third assumption is used. Where $I = I'$, we must show that

$\bar{\mathcal{S}}'[\![p1]\!]\ v \sqsubseteq \mathcal{S}[\![p1]\!]\rho$

which requires, in order to use the induction hypothesis, that

$\forall I'.\ \bar{\mathcal{S}}'[\![\underline{\text{call}}\ I']\!]\ v \sqsubseteq \rho\ I'$

The second and third assumptions imply this.

For p = ⌜<u>call</u> I⌝, we must show that

$\text{FUN}\mathcal{S}\ \bar{\mathcal{S}}'[\![\underline{\text{call}}\ I]\!]\ v \sqsubseteq \mathcal{S}[\![\underline{\text{call}}\ I]\!]\rho$

that is,
$\bar{\mathcal{S}}'[\![\text{FST}(v\ I)]\!]\ (\text{SND}(v\ I)) \sqsubseteq \rho\ I$

and this follows by the third assumption. The letrec case is done by an inner induction. The other cases are straightforward.

Intuitively, the extra clause is needed to relate $\bar{\mathcal{S}}$ to $\mathcal{S}$, in which there is a 'hidden' occurrence of $\mathcal{S}$. This explains the presence of $\bar{\mathcal{S}}$ in the relation between high level environments and closure environments. The extra clause,

$$\bar{\mathcal{S}} \sqsubseteq \text{FUN}\mathcal{S}\ \bar{\mathcal{S}}$$

allows us to use the induction hypothesis during the proof. For related discussion, see Notes 3. and 5. below, and the Conclusions.

3. Instead of proving (ii) by induction on H in a formula which is a conjunction, we can appeal to a rule of induction similar to that mentioned in Note 2.:

$$w[\bot/f] \quad \& \quad ((w \quad \& \quad f \sqsubseteq \text{FIX fun}) \supset w[(\text{fun } f)/f]) \supset$$

$$w[(\text{FIX fun})/f]$$

See also p. 182.

4. This requires that $\bot - 1 = \bot$.

5. The use of the derived rule of induction and the corresponding tactic is a more elegant way of accomplishing the proofs than the technique (employed in the informal proofs of Lemma 4.5 and Lemma 4.10) of proving a conjunction, the first conjunct of which is a formula of the form

$$f \sqsubseteq \text{FIX FUN}$$

by induction. The use of the derived rule, FIXPTRULE, saves us having to explicitly prove that conjunct each time this method of proof is used. In addition, it makes a more concise composite tactic; were we to prove a conjunction, we would first have to write a tactic (FIXPTTAC1) to produce a subgoal whose formula was a conjunction, from the original goal:

FIXPTTAC1 ($\vdash f \equiv$ FIX FUN)

(w, ss, A)

$f \sqsubseteq$ FIX FUN & w
ss
A

We would then follow the application of FIXPTTAC1 by the application of:

```
(INDUCTAC [⊢f ≡ FIX FUN])+
CONJASSUMPTAC
CONJTAC
```

This separates the conjunctive assumption put into the assumption list into two assumptions, and separates the subgoal into the main subgoal, and the other, whose formula is

FUN f′ ⊆ FIX FUN

for arbitrary f′. We would <u>then</u> write a tactic, FIXPTTAC2, say, to inspect the assumption list, discover the assumption

f′ ⊆ FIX FUN

and add to the list the assumption

FUN f′ ⊆ FIX FUN

Its proof part would use the rule FIXPT. Finally, we would call USEASSUMPLHSTAC+ to use the newly added assumption, and we would be left with the main subgoal. Although the effect is the same, a single call of FIXPTTAC is clearly a more palatable solution.

6. Just as NATCASESTAC is intended to be used in a theory in which the type nat exists, IDCASESTAC is meant to be used where the type ID exists. It is also to be used in theories of which equality is a parent, as the constant EQ is mentioned.

Conclusions

Note on Computational and Structural Induction

We have deferred, until all of the proofs have been presented, a discussion of the relation between computational and structural induction; we have used both, at various junctures in the proofs. As we have remarked, structural induction can be viewed as concealing a computation induction on a 'copying' function. Where a computation does not follow the well-founded structure of its argument, but rather, 're-enters' the argument (e.g. when traversing knotted structures), computation induction is what is needed. Where the structure of a computation does match the structure of the argument, computation induction on the function involved, and structural induction on the argument produce much the same proofs, and the latter seems more natural. Structural induction is neater when the formula to be proved is an implication whose antecedent contains an occurrence of what would otherwise be an induction variable; for example, the statement of equivalence of $\mathscr{S}$ and $\bar{\mathscr{S}}$:

$$\forall I.\ \rho\ I = \bar{\mathscr{S}}[\![\underline{\text{call}}\ I]\!]\ v \supset \mathscr{S}[\![p]\!]\rho = \bar{\mathscr{S}}[\![p]\!]\ v$$

For further discussion of this point, see Chapter 4, Note 2.

Structural induction also seems more natural when we are considering the relation between two functions which unfold at different rates; for example, lsem and hsem, in Chapter 3. In that instance one can avoid using iterated computation induction, and proving a pair of inequivalences at the top level, by inducting on the structure of high level programs rather than on the semantic

functions. However, in this case, the structural induction proof requires an inner computation induction (for the while construct) which itself mirrors the computation induction proof. (For further discussion, see Chapter 3, Note 3.) We would also expect this to apply to the $\mathscr{D}$ to $\mathscr{R}un$ proof in Chapter 4; however, the semantics of the call case makes formulation of the appropriate rule of structural induction difficult. The natural rule (to which we appealed, for example, in the proof of Theorem 4.1) is:

$$
\begin{array}{l}
w[\bot/p] \quad \& \\
w[a/p] \quad \& \\
\forall I.\ w[\underline{call}\ I/p] \quad \& \\
\quad \forall p1\ p2.\ (w[p1/p] \quad \& \quad w[p2/p] \supset \\
\quad\quad \forall I.\ w[\underline{let}\ I = p1\ \underline{in}\ p2/p] \quad \& \\
\quad\quad\quad w[\underline{letrec}\ I = p1\ \underline{in}\ p2/p] \quad \& \\
\quad\quad\quad w[p1;p2/p]\ ) \quad \supset \\
\forall p.\ w
\end{array}
$$

That is, the undefined, atomic and call cases are the basis cases, and the let, letrec and sequencing cases are the steps. However, in the $\bar{\mathscr{S}}$ to $\mathscr{D}$ and the $\mathscr{D}$ to $\mathscr{R}un$ proofs, the call semantics require that the call construct be treated as an inductive step rather than as a basis case. That is,

$$\underline{\text{In } \bar{\mathscr{S}}}: \quad \bar{\mathscr{S}}[\![\underline{call}\ I]\!]\ v \ = \ \bar{\mathscr{S}}[\![p']\!]\ v' \quad \text{where } (p',v') \ = \ v\ I$$

$$\underline{\text{In } \mathscr{D}}: \quad \mathscr{D}[\![\underline{call}\ I]\!]\ d\ n' \ = \ \mathscr{D}[\![p']\!]\ d'\ n \ \text{where } (p',n') \ = \ d\ n\ I$$

In both cases, in proofs by computation induction, we may instantiate the induction hypothesis to p' in order to reason about $\bar{\mathscr{S}}[\![p']\!]\ v'$ or $\mathscr{D}[\![p']\!]\ d\ n'$. In proofs by structural induction, assumptions about subprograms p1 and p2 are of no assistance in reasoning about $\mathscr{S}[\![p']\!]\ v'$ or $\mathscr{D}[\![p']\!]\ d\ n'$. In the $\mathscr{S}$ to $\bar{\mathscr{S}}$

proof, on the other hand, in which

$$\underline{\text{In } \mathscr{S}} : \mathscr{S}[\![\underline{\text{call}}\ I]\!]\rho \quad = \rho\, I$$

there is no recursive call of the semantic function, and structural rather than computation induction can be employed.

To derive the above rule of structural induction for programs of type HPROGRAM, we define a function (pcopy, say) of type HPROGRAM ⟶ HPROGRAM to be the least fixed point of a functional (pcopyfun) where

$$\begin{array}{lll}
\text{pcopyfun} & = & \lambda\ \text{pcopy p.} \\
& & \text{p} = \ulcorner a \urcorner \Rightarrow \ulcorner a \urcorner \mid \\
& & \text{p} = \ulcorner \underline{\text{call}}\ I \urcorner \Rightarrow \ulcorner \underline{\text{call}}\ I \urcorner \mid \\
& & \text{p} = \ulcorner \underline{\text{let}}\ I = \text{p1}\ \underline{\text{in}}\ \text{p2} \urcorner \Rightarrow \\
& & \quad \ulcorner \underline{\text{let}}\ I = \text{pcopy p1}\ \underline{\text{in}}\ \text{pcopy p2} \urcorner \mid \\
& & \text{p} = \ulcorner \underline{\text{letrec}}\ I = \text{p1}\ \underline{\text{in}}\ \text{p2} \urcorner \Rightarrow \\
& & \quad \ulcorner \underline{\text{letrec}}\ I = \text{pcopy p1}\ \underline{\text{in}}\ \text{pcopy p2} \urcorner \mid \\
& & \text{p} = \ulcorner \text{p1;p2} \urcorner \Rightarrow \\
& & \quad \ulcorner \text{pcopy p1;pcopy p2} \urcorner \mid \bot
\end{array}$$

pcopy returns the well-founded part of high level programs. We make the assumption that

$$\forall \text{p. pcopy p} \ = \ \text{p}$$

This axiom aserts that every program is the limit of its approximants, i.e.

$$\text{p} \ = \ \bigsqcup \text{pcopyfun}^{n} \bot \ \text{p}$$

We then do computation induction on the function pcopy in the new formula w′, where w′ = w[pcopy p/p]. That is, we prove from the basis and step of the structural induction rule

$w'[\perp/pcopy]$

and

$w'[pcopy'/pcopy] \supset w'[pcopyfun\ pcopy'/pcopy]$

and conclude, by normal computation induction, that

$w'[FIX\ pcopyfun/pcopy]$

that is,

$w'[pcopy/pcopy]$

which is

$w[pcopy\ p/p]$

But since we assumed that pcopy p = p, this proves the conclusion of the rule of structural induction.

Another example of a derived induction rule, ITINDUCT, is described in Chapter 3, and in the Appendix.

General Conclusions

In the preceding chapters, we have given accounts of two actual (and one hypothetical) case studies in the generation of formal proofs by the design and application of tactics. These tactics were composed (by the use of tacticals) from standard tactics and from a body of tactics which we *derived*.

The derived tactics can be divided into several (not entirely disjoint) classes. The simplest are ones which invert standard or derived rules of inference, or whose proof functions evaluate short forward inferences. The first class of tactics includes the following, where we distinguish the tactics we have actually implemented in ML from those merely specified in this presentation, by enclosing the latter in parentheses. The tactics are listed with the location in the text of their main appearance.

BOTREFLTAC	Ch. 1	p. 36
MINCOMBTAC	Ch. 1	p. 35
CONJTAC	Ch. 2	p. 80
EXTTAC	Ch. 2	p. 63
LAMGENTAC	Ch. 3	p. 128
COMBTAC	Ch. 3	p. 128
IMPTAC	Ch. 4	p. 179
SYNTHTAC	Ch. 2	p. 61
BYTAC	Ch. 2	p. 60
(FIXPTTAC)	Ch. 4	p. 183

The members of the second class of tactics all use properties of the least fixed point operator in producing subgoals from goals. One can view the tactics in this class as part of a theory of FIX. The proof parts of these tactics rely on standard and derived rules about FIX, including INDUCT. The class includes:

MINFIXTAC	Ch. 2	p. 62
UNFOLDTAC	Ch. 2	p. 64

UNFOLDOCCSTAC	Ch. 2	p. 64
(UNFOLDCHOOSETAC)	Ch. 2	p. 84
(FIXPTTAC)	Ch. 4	p. 183
(FIXFUNTAC)	Ch. 4	p. 184
(LFPTAC)	Ch. 4	p. 184

The tactics in the next class have the common property that they use current assumptions (formulae in the assumption lists of goals) in order to advance proofs. In some cases, the use of assumptions is achieved by recognising tautologies (e. g. USEASSUMPTAC). In others, it is achieved by inspecting and supplementing the list of assumptions, and justifying the additions with appropriate proofs (e.g. CONJASSUMPTAC and FIXPTTAC). Still other tactics in the class use assumptions by proposing intermediate subgoals whose achievements are to be combined (in ways specified by the tactics) with certain of the assumptions in the assumption lists (e.g. USEASSUMPLHSTAC). For all of the tactics in this class, the proof parts evaluate fairly short forward proofs. They include:

USEASSUMPTAC	Ch. 2	p. 48
USEASSUMPRHSTAC	Ch. 2	p. 67
USEASSUMPLHSTAC	Ch. 2	p. 68
(USEASSUMPCHOOSETAC)	Ch. 2	p. 83
CONJASSUMPTAC	Ch. 2	p. 81
(USEIMPASSUMPTAC)	Ch. 4	p. 181
(USEIMPASSUMPRHSTAC)	Ch. 4	p. 181
(USEIMPASSUMPLHSTAC)	Ch. 4	p. 182
(FIXPTTAC)	Ch. 4	p. 183

Further research suggests itself in this area. USEASSUMPLHSTAC and the rest begin to cope with the problem of reasoning about inequivalences, something which is often necessary, since equivalences are frequently proved by different methods in the two directions. Much more work remains be done on using inequivalences in proofs.

A related class of tactics can be envisioned which control simplification in proofs. We have used one tactic of this sort, namely TEMPSIMPTAC, in Chapter 3, p. 127, which uses a theorem as a simprule for one round of simplification, but does not deposit the theorem in the simpsets of ensuing subgoals. One is likely to need other tactics of this genre in more complex proof efforts.

Finally, one can define a class of tactics which invert rules that are derived from the basic PPLAMBDA rules INDUCT and CASES:

NATCASESTAC	Ch. 2	p. 80
(IDCASESTAC)	Ch. 4	p. 185
(HCASESTAC)	Ch. 3	p.121
(HCASESTAC)	Ch. 4	p. 186
(HINDUCTAC)	Ch. 3	p. 136
(HINDUCTAC)	Ch. 4	p. 186
ITINDUCTAC	Ch. 3	p. 125
(INDUCTCHOOSETAC)	Ch. 2	p. 83
(SIMPLEINDUCTAC)	Ch. 4	p. 180

The proof parts of the derived induction tactics (i.e. the rules upon which the tactics are based) construct new bases and steps from achievements of the subgoals (and other proved facts), and call

INDUCT. The derivations of structural induction from INDUCT, for various recursively defined structures, follow the same pattern as sketched for HINDUCT (in the previous section), and one can envision an ML procedure to automatically derive the rules from the specifications of the domains involved. The same remarks apply to the `structural cases' rules, HCASES, etc., which can be regarded as induction rules without induction hypotheses.

From the various standard and derived tactics, we have composed several larger tactics, including some to solve parts of the schema problems, and some to perform parts of the correctness proof of the Russell compiler. For example, the following composite tactics perform the proofs of Theorem 2.5 (Chapter 2, p. 52), and the step of difficult half of the compiler proof, Theorem 3.1b (Chapter 3, p. 99), respectively:

```
TACL3                            Ch. 2   p.  68

     (MINFIXTAC thG)+
     EXTTAC+
     (INDUCTAC [thF])+
     GENTAC*
     (UNFOLDOCCSTAC 2 thF1)+
     (CONDCASESTAC+)*
     (USEASSUMPRHSTAC+)*

COMPILERTAC                      Ch. 3   p. 131

     (ITINDUCTAC AX1sem 4)+
     GENTAC
     TEMPSIMPTAC AXLSEM
     TEMPSIMPTAC thhsem
     HCASESTAC+
     (USEASSUMPRHSTAC ORELSE (COMBTAC ORELSE LAMGENTAC ORELSE
                              CONDCASESTAC)+)*
```

It is perhaps surprising that the small set of standard tacticals

(in conjunction with the ML failure-trapping mechanism) is adequate as a control structure for these complex proof efforts; in fact, THEN and REPEAT account for most of the uses of tacticals we have made. One might have been expected to need a richer language of tacticals, or more subtle ones. It is less surprising, though, in light of our methodology and objectives. Firstly, the structure of the proofs performed was determined by examination of the informal proofs prior to the formalisation of the problems. (One could perhaps call this activity checking of informal proofs, as well as generation of formal proofs by tactics.) As the composite tactics reflect the patterns of inference of the informal proofs, we have tended to anticipate the sequence (or tree) of subgoals, and so not rely, except in after-the-fact generalisations (such as SCHEMATAC, Ch. 2, p. 86) on the tactical ORELSE, or on more complex derived tacticals which would examine alternatives, or backtrack.

Secondly, as we have not addressed issues in automatic theorem proving (such as automatic generalisation of goals, strengthening of induction hypotheses, or discovery of lemmas), but have instead provided the difficult insights before embarking on the machine generated proofs, we have avoided having to write tactics which would naturally require more sophisticated control structures (i.e. more sophisticated tacticals). Possibly, it is simply naive to expect that tactics can be designed to solve goals for which the informal proofs are not, at least in outline, understood in advance. Quite aside from the inefficiency of searching for proofs, it may be that there are just too many fine points to be considered in the proof process for this to work. Nonetheless, further case studies

and analyses of tactics are likely to reveal new tactics and tacticals which require <u>less</u> planning in advance on behalf of the user.

It would also be desirable to research an <u>intermediate</u> level of tactics, tactics midway in complexity between the derived tactics discussed in the preceding chapters and the composite tactics which solve our goals in one application. We would like to investigate further a level of conceptually coherent tactics which do <u>parts</u> of the proofs; ENDTAC (Chapter 3, p. 129) is a possible example of the level sought.

In addition to designing composite tactics for solving various classes of problems, we have begun to develop a methodology for tactical proof. In both case studies, we commenced the proof efforts by building theories, or networks of theories. In the schema proofs, we required theories of the new data types (lists and integers), and so extended PPLAMBDA by introducing and axiomatising the new types and constants. In the Russell compiler proof, we needed a rather more elaborate structure of theories to represent the syntax and semantics of the languages involved, and to express the compiling algorithm. PPLAMBDA was supplemented by a large set of new types, constants and axioms, organised in a hierarchy of theories.

In the (networks of) theories, we then developed structures of lemmas. In the schema proofs, for example, we generalised the original goals, and proposed several subgoals; some of the theorems achieving the subgoals were used as simplification rules in proving the original goals. In the compiler proof, we found it convenient

and efficient to prove several layers of lemmas before embarking on the main goals. Each layer formed simplification rules for the next layer. Often, the lemmas were proved just by simplification.

We observed, after examining the tactics which generated the schema proofs, that the proofs could have been made more automatic (and the tactics more concise) by leaving more of the proofs to simplification; that is, by carefully selecting lemmas to be used as simplification rules, so that the proofs could, to a greater extent, be driven by simplification. This methodology was explored further in the compiler proof effort in Chapter 3. The proof which we actually performed in LCF relied for its control structure on a sequence of user-specified substitutions and unfoldings, but as the analysis in Chapter 3 revealed, it could have been generated more easily as a simplification-guided proof. This requires a certain amount of forethought in order to isolate the correct lemmas; it also requires care that simplification is not carried too far. One wishes, for example, to avoid simplifying a goal whose formula is an instance of an induction hypothesis. The advantage of simplification-guided proof efforts is that they demand much less user intervention and attention to detail during the performance of the proofs. In addition, the tactics which generate the proofs seem more easily generalised, reflect the structure of the proof more transparently, and are more efficient.

One would hope to develop a theory as well as a methodology of tactical proof. Although the refinement of a theory would require more experience with tactical proof than has been gathered to date, we have at least raised some issues which a theory should treat.

Several of these are discussed below.

One issue is the choice between procedural and declarative representations of facts. As indicated in the discussion of BYTAC (Chapter 2, p. 60), for example, we have frequently found it more convenient to represent facts as ML procedures (mapping theorems to theorems) than as theorems (implications) stored in LCF theories. The procedural representation lends itself more naturally to the tactical style of proof; the proof parts of tactics call the corresponding ML procedures, which then prove the theorems desired. This allows all of the matching and instantiation involved in the use of theorems to be done implicitly within the ML procedures. For instance, the tactic MINFIXTAC (Chapter 2, p. 62) returns a proof part which expects a theorem of the form

$$\vdash \text{FUNG F} \sqsubseteq \text{F}$$

and combines that theorem with a (given) theorem of the form

$$\vdash \text{G} \equiv \text{FIX FUNG}$$

in a proof by induction, to return a theorem of the form

$$\vdash \text{G} \sqsubseteq \text{F}$$

We have chosen to write an ML procedure, MINFIX, which maps any theorem of the expected form to the theorem desired in just this way. We could instead have proved and stored a theorem

$$\vdash \forall \text{G}':*.\ \text{F}':*.\ \text{FUNG}':* \longrightarrow *.$$
$$\text{G}' \sqsubseteq \text{FIX FUNG}' \quad \& \quad \text{FUNG}'\ \text{F}' \sqsubseteq \text{F}' \quad \text{IMP} \quad \text{G}' \sqsubseteq \text{F}'$$

When specifying the proof part of the tactic MINFIXTAC we would have to fetch the theorem from the theory in which it were stored (this information having been a parameter to MINFIXTAC), then compute the types of the terms F, G and FUNG in the two theorems to be combined, then call the PPLAMBDA rule INSTTYPE (see [15]) to prove a theorem instantiated to the correct types, then instantiate the result to the correct variables, F, G and FUNG, then conjoin the theorems and call MP (Modus Ponens), all in order to use the theorem. The ML procedure MINFIX simply extracts parts of the two theorems, gives them meta-names, and constructs a new basis and step on which to call induction. We use MINFIX, of course, at cost of reproving the theorem by induction at each invocation; the point, however, is the naturalness of the procedural form for tactical proof. Of course, since the process of translating from a theorem into the corresponding rule is obviously a uniform one, we could standardise it in ML. If a package to translate in this manner were available, the procedural-declarative distinction would be less meaningful than it is at present.

Another issue (already mentioned) is the extent to which (and the ways in which) increasing portions of proof can be left to simplification, as we have begun to do in Chapters 2 and 3, by proving theorems (to be used as simprules) which specify the contexts in which, or conditions under which, terms should be simplified.

Finally, as part of a theory of tactical proof, one would wish to build a larger repetoire of derived tactics, and to identify further dimensions along which to classify them. Since classes of

tactics reflect patterns of inference, this could initiate an explicit and empirical study of patterns of inference.

Many such issues remain to be explored. What we _have_ concluded from this work can be briefly summarised as follows:

(i) We have demonstrated, in the case studies described, that goal-oriented tactical proof is a natural way of generating large, formal proofs. A general purpose programming language, ML, forms an effective interface between user and system, allowing large portions of proof to be performed automatically by the application of procedures (representing general strategies) to data (representing goals).

(ii) In general, LCF has shown itself to be a flexible and powerful vehicle for generating formal proofs. The simplification facility, in particular, contributes to this. The basic simplifications themselves make LCF more than a proof checking system, as they cover a great deal of simple reasoning. Beyond that, we have illustrated how much of the remaining work of proof can be relegated to simplification by careful choice of lemmas to be used as simplification rules.

(iii) The tactics which perform proofs reveal the structure of the proofs in an intelligible and high level way, and lend themselves to further generalisation. SCHEMATAC, for example, (Chapter 2, p. 86), would appear to be useful in a large number of proofs about recursively defined function schemata. Likewise, we would expect a tactic similar to COMPILERTAC (Chapter 3, p. 131) to perform correctness proofs for more sophisticated compilers (e.g. compilers for richer high level languages or more concrete low level languages).

(iv) The ability to incrementally and hierarchically construct theories is vital to the proof efforts described. The organised introduction of new types, constants and axioms, the modular development of theories, and the ability to store and access proved facts, all help to make a wide variety of theorems expressible in LCF.

(v) It seems feasible to perform fairly large proofs by the methods we have described; the effort required on the part of the user is concentrated more on formalising the problems and factoring out useful lemmas than on deriving or applying the tactics. The proof of the Russell compiler, in particular, illustrates this. Of course, the compilers in question are only toy compilers; as for the feasibility of proving `real` implementations by these techniques, research remains to be done.

Extensions to LCF which have presented themselves in the course of this work relate primarily to standard 'packages' which could be added to the system. In particular, it would be helpful to have standard packages to derive structural induction and cases rules (and tactics) for suitable structures; to derive injection, selection and projection functions, and the associated axioms, for arbitrary n-ary separated sums, so that all the UP's and DOWN's (evident in Chapters 3 and 4) could be suppressed; and to derive procedural representations (inference rule schemata) from declarative ones (stored theorems).

An addition that would enlarge the expressive power of PPLAMBDA would be the ability to name relations. For example, one often introduces a relation R by writing

a R b iff w

for some formula w. This is, of course, not unproblematical; questions to do with whether relations admit induction are not fully understood

We would hope that the work described here inspires further research in the direction of formal correctness proofs for implementations of more realistic programming languages.

Future Work

We would like to extend the work described in Chapter 2 by studying more examples of recursive function schemata. We would like to further specify, and to implement, the general tactic,

SCHEMATAC, sketched in the Conclusions of Chapter 2.

Regarding Chapter 3, we would like to extend the compiler and proofs to lower levels, by stages. As we noted, for example, the Russell compiler produces programs which feature a sort of block-structuring used to limit the set of labels needed to a finite set. This circumvents the problem of generating unique, new label names The formulation could be carried a step further by designing and proving a `gensym` mechanism. We would also like to formulate and prove a compiler which produced machine-like code (perhaps as suggested by Newey, [38]).

Regarding Chapter 4, we plan to perform the proofs described in LCF. We would also like to consider, as for Chapter 3, lower level languages. In particular, we would like to formulate an activation stack semantics in which incremental layers were kept, rather than whole environments, as well as a semantics in which displays, in the usual sense, modelled activation stacks. We would also like to study the proof techniques for other high level constructs, e.g., parameter passing mechanisms, co-routines, data structures of various sorts, and exception handling mechanisms, all in a schematic, feature-by-feature way, as we have done so far. It would remain to be investigated whether the methods of dealing with recursively defined relations used in Chapter 4 were useful in other settings. Eventually, we would like to gather the separate high level features into a single language and `compose` the proofs, so that a chain of proofs would link very high level languages with machine-like languages. Some questions relevant to a chain of proofs of this sort would be (i) the order of the compilation of

various features, (ii) the appropriate semantics at each level (for example, in the Russell compiler proof, the low level language has a quite natural continuation semantics, but it is not clear that this would be a graceful semantics for proofs of equivalence to still lower levels), and (iii) the relations between the tactics used to generate the proofs between the various levels.

Finally, as we have suggested earlier, we would like to research more sophisticated tacticals for composing tactics, and, in conjunction, techniques for more automatic proof finding and framing of lemmas, as well as the existence of a body of coherent intermediate-level tactics expressing common `chunks' of reasoning.

Appendix: Some Technical Details

In this appendix we supply some details about the actual performances of the proofs described in Chapter 2. An account of this sort is complicated by two factors. Firstly, both the schema proofs, in Chapter 2, and the Russell compiler proof, in Chapter 3, were performed in an older version of LCF (as documented in [13] rather than in [15]) but we have nonetheless described the proofs as they would be performed in the current LCF. Although at the level of tactics the changes are not profound, they make the theories involved, and some of the tactics (in particular, the derived induction and cases tactics) look rather different. Secondly, in the case of the compiler proof, the scope of the actual proof effort makes it difficult to produce a demonstration of the whole process. That is, the proof was performed over a period of several weeks, by a combination of forward and tactical proof. Some of the lemmas are quite (CPU) time-consuming and it does not seem worthwhile to reprove them simply for the sake of demonstration. Our aim in this appendix is just to give an impression of the nature of the interaction which produced the proofs (and to give some evidence that they were in fact performed!). We hope to achieve this by describing the actual List Stack proof.

LCF is an interactive system in which one can directly introduce definitions and construct theories. The usual mode of interaction, however, is via files prepared by the user before entering LCF and subsequently read in; this saves effort. Files typically contain definitions of ML functions, definitions of particular goals and tactics, and LCF commands for constructing theories. Theories, once constructed, are stored by LCF on \`display\` files, some of which are shown presently.

To enable computer printing of PPLAMBDA, the following conventions are observed:

<table>
<tr><td>==</td><td>is written for</td><td>≡</td></tr>
<tr><td>\</td><td></td><td>λ</td></tr>
<tr><td>UU</td><td></td><td>⊥</td></tr>
<tr><td><<</td><td></td><td>⊑</td></tr>
<tr><td>!</td><td></td><td>∀</td></tr>
<tr><td>#</td><td></td><td>×</td></tr>
<tr><td>|</td><td></td><td>|</td></tr>
<tr><td>=></td><td></td><td>⇒</td></tr>
<tr><td>]-</td><td></td><td>⊢</td></tr>
</table>

Character strings (tokens) representing PPLAMBDA constants are enclosed in quotes \`thusly\`; those representing type constants or the names of theories \`\`thusly\`\`. All PPLAMBDA objects (terms, types and formulae) are written in quotes like "this", and types are preceded by a colon, e.g. ":type". ML expressions are terminated by a double semi-colon, e.g. expr;;, and reserved words in ML are not underlined as they are in the text. Comments appear enclosed in percentage signs %like this%.

Details of the List Stack Proof in LCF

The definitions needed for the formulation of the problem are shown in the fragments of a file below. The proof was performed in a theory of lists which we constructed. The theory is displayed below in a file prepared by LCF.

In the version of LCF at the time, there was no facility for defining polymorphic PPLAMBDA type operators (such as * list). Lists, instead, had to be lists of elements of a particular types (a type ":d", here). New types were introduced by domain equations in which only one type operator could appear in an equation:

```
NEWTYPES [ ``DLIST = . + DPAIR`` ;
           ``DPAIR = D # DLIST`` ] ;;
```

where . denotes a domain consisting of exactly one element ($\perp$). Some new constants were introduced and given a representation in terms of standard PPLAMBDA constants. It must be recalled that the standard sum in LCF at the time was separated sum, so that the definitions of INL, etc. here are not the same as the current definitions, and UP and DOWN, as needed now, were not required.

```
NEWCONSTANT ( `HD` , ":DLIST->D" ) ;;

NEWCONSTANT ( `TL` , ":DLIST->DLIST" ) ;;

NEWCONSTANT ( `CONS` , ":D->(DLIST->DLIST)" ) ;;

NEWCONSTANT ( `NIL` , ":DLIST" ) ;;

NEWCONSTANT ( `DUMMY` , ":D" ) ;;

NEWCONSTANT ( `LIST` , ":D->DLIST" ) ;;
```

```
NEWAXIOMS();;

AXHD    "HD == \DL:DLIST.FST(OUTR DL :DPAIR) :D"

AXTL    "TL == \DL:DLIST.SND(OUTR DL :DPAIR) :DLIST"

AXNIL   "NIL == INL () :DLIST"

AXCONS  "CONS == \D:D.\DL:DLIST.INR(D, DL) :DLIST"

AXNIL2  "EQ NIL NIL == TT"

AXLIST  "LIST == \D:D.INR(D, NIL) :DLIST"

NNCNS1  "!D:D. !S:DLIST. EQ S NIL == FF IMP EQ(CONS D S)NIL == FF"

NNCNS2  "!D:D. !S:DLIST. EQ S NIL == TT IMP EQ(CONS D S)NIL == FF"
```

The `usual` list axioms were then proved, mostly by simplification, and stored in the theory of lists. They are displayed below on an LCF-prepared file. The names of axioms and theorems on files of this sort are shown to the left of the formulae.

```
STRHD    "HD UU == UU:D"

STRTL    "TL UU == UU:DLIST"

HDCONS   "!DL:DLIST. !D:D. HD(CONS D DL) == D"

TLCONS   "!DL:DLIST. !D:D. TL(CONS D DL) == DL"

HDNIL    "HD NIL == UU:D"

TLNIL    "TL NIL == UU:DLIST"

HDLIST   "!D:D. HD(LIST D) == D"

TLLIST   "!D:D. TL(LIST D) == NIL"

LISCNS   "!D:D. LIST D == CONS D NIL"

CNSNIL   "!D:D. EQ(CONS D NIL)NIL == FF"
```

A parent of the list theory was a theory of equality in which a constant EQ was introduced. (EQ s NIL) is used here where (NULL s) is used in the text. Also, (LIST dummy) is used here where (LIST NIL) is used in the text. (For a list theory in which the `real` list axioms are introduced directly, see [15], Appendix 1.)

For the List Stack Proof, some PPLAMBDA constants are first assigned types. By convention (see [15], 3.2.3) these types are assigned to future occurrences of the constants unless otherwise indicated.

```
"F:D->D";;
"F1:D->D->DLIST->D";;
"F:D->D";;
"X:D";;
"P:D->TR";;
"H:D # D -> D";;
"G1:D->D";;
"G2:D->D";;
"Z:D";;
"S:DLIST";;
"Exp:(D->D)->(D#D->D)->D->D->DLIST->D";;
"E:D";;
```

Assumptions were introduced to define the four functions and to represent the associativity assumption and the others. The first four correspond to Chapter 2's thF, thF1, thExp and thG, respectively.

```
LET TH1 = ASSUME "F == FIX (\F'.\x. P x => F x |
                  H (F' (G1 x), F' (G2 x)))";;
LET TH2 = ASSUME "F1 == FIX (\F1'. \x. \z. \s. EQ s NIL => z |
                  P x => F1' (HD s) (H (z, F x)) (TL s) |
                         F1' (G1 x) z (CONS (G2 x) s))";;
LET TH3 = ASSUME "Exp == FIX (\Exp'.\F. \H. \x. \z. \s.
                  EQ s NIL => z | Exp' F H (HD s) (H(z, F x)) (TL s))";;
LET TH4 = ASSUME "G == FIX (\G'. \x. \z. \s. EQ s NIL => z |
                  G' (HD s) (H(z, F x)) (TL s))";;

LET LEFTID = ASSUME "!x:D. H(E, x) == x";;
LET STRICTRH = ASSUME "!x:D. H(x, UU) == UU";;
LET STRICTLH = ASSUME "!x:D. H(UU, x) == UU";;
LET ASSOCH = ASSUME "!A:D.!B:D.!C:D. H((H(A,B)),C) == H(A,(H(B,C)))";;
```

Next, some of the axioms and theorems from list theory were fetched and bound to ML identifiers. (Map is the usual mapping function.) Further details on the commands AXIOM, FACT, etc., are to be found in [15], 3.2.1.

```
LET [HDCONS;TLCONS;HDLIST;TLLIST;CNSNIL;LISCNS] = MAP (FACT `-`)
    [`HDCONS`;`TLCONS`;`HDLIST`;`TLLIST`;`CNSNIL`;`LISCNS`];;
LET [NNCNS1;NNCNS2;AXLIST;AXNIL2] = MAP (AXIOM `-`)
    [`NNCNS1`;`NNCNS2`;`AXLIST`;`AXNIL2`];;
```

Simpsets were then constructed.

itlist:(* ⟶ ** ⟶ **) ⟶ * list⟶ ** ⟶ **

is a standard ML function such that

itlist f [l1;...;ln] x = f l1 (f l2 (...(f ln x)...))

The standard function ssadd is described on p. 215.

```
LET SS23 = ITLIST SSADD [NNCNS1;NNCNS2;HDCONS;TLCONS;ASSOCH;STRICTLH;
                         STRICTRH] BASICSS;;
LET SS5  = SSADD STRICTLH BASICSS;;
LET SS6  = ITLIST SSADD [LISCNS;CNSNIL;LEFTID;HDLIST;TLLIST;
                         AXLIST;AXNIL2] BASICSS;;
```

The union of ss23, ss5 and ss6 is called SSL in the text. Finally, the relevant goals and tactics were constructed. The correspondences to the names in Chapter 2 are:

Appendix	Chapter2
goal1	goalL1
goal2′	goalL2
goal3	goalL3
goal4	goalL4
goal5	goallemL2
goal6	goalL5
goal7	goalL0

UNWINDTAC	UNFOLDTAC
UNWINDOCCSTAC	UNFOLDOCCSTAC
BYRULE	BYLAW
BYTAC2	BYTAC
WEAKFIXTAC	MINFIXTAC
WEAKFIXRULE	MINFIX
APPLYTAC2	EXTTAC
ANYCASESTAC	CONDCASESTAC
USEIHTAC	USEASSUMPTAC
USEIHLESSTAC	USEASSUMPLHSTAC
USEIHMORETAC	USEASSUMPRHSTAC
TAC2b	TACL2
TAClemma	TAClemL2
TAC3	TACL3

```
LET GOAL1 = "G == EXP F H ",BASICSS,[ ]:FORM LIST;;
LET GOAL2' = "F1 << G", SS23, [ ]:FORM LIST;;
LET GOAL3 = "G << F1 ", SS23, [ ]:FORM LIST;;
LET GOAL4 = "F1 == EXP F H ", BASICSS,[ ]:FORM LIST;;
LET GOAL5 = "!x.!s.F1 x UU s == UU", SS5, [ ]:FORM LIST;;
LET GOAL6 = "EXP F H x E (LIST DUMMY) == F x", SS6, [ ]:FORM LIST;;

LET GOAL7 = "F1 x E (LIST DUMMY) == F x",BASICSS,[ ]:FORM LIST;;
```

```
LET TAC1 = REPEAT APPLYTAC2 THEN INDUCTAC [TH4;TH3] THEN SIMPTAC
           THEN REPEAT GENTAC THEN ANYCASESTAC THEN SIMPTAC
           THEN USEIHTAC THEN SIMPTAC;;

LET TAC2B = REPEAT APPLYTAC2 THEN INDUCTAC [TH2] THEN SIMPTAC
            THEN REPEAT GENTAC THEN UNWINDTAC TH4 THEN
            SIMPTAC THEN ANYCASESTAC THEN SIMPTAC THEN
            ANYCASESTAC THEN SIMPTAC THEN UNWINDTAC TH1
            THEN SIMPTAC THEN USEIHLESSTAC THEN SIMPTAC
            THEN UNWINDOCCSTAC [1] TH4 THEN SIMPTAC THEN
            UNWINDOCCSTAC [1] TH4 THEN SIMPTAC;;

LET TAC3 = WEAKFIXTAC TH4 THEN SIMPTAC THEN REPEAT APPLYTAC2
           THEN SIMPTAC THEN INDUCTAC [TH1] THEN
           UNWINDOCCSTAC [2]  TH2 THEN SIMPTAC THEN REPEAT GENTAC THEN
           ANYCASESTAC THEN SIMPTAC THEN ANYCASESTAC THEN
           SIMPTAC THEN USEIHMORETAC THEN SIMPTAC THEN
           USEIHMORETAC THEN SIMPTAC;;

LET TACLEMMA = INDUCTAC [TH2] THEN SIMPTAC THEN REPEAT GENTAC THEN
               SIMPTAC THEN ANYCASESTAC THEN SIMPTAC THEN
               ANYCASESTAC THEN SIMPTAC THEN USEIHTAC THEN SIMPTAC;;
```

```
LET TAC6 = UNWINDTAC TH3 THEN SIMPTAC THEN UNWINDTAC TH3 THEN SIMPTAC;;
```

TAC1 solves goal1, TAC2b solves goal2′, TAC3 solves goal3 with the eventual result of applying TAClemma to goal5 in its simpset. To achieve goal4 we use BYTAC2 to produce goal2 and goal3. To achieve goal7, we add the theorems achieving goal4 and goal6 to the simpset, and call SIMPTAC.

The following (fragments of a) transcript of an actual session with LCF demonstrate the performance of the proofs in the system. Note that theorems are displayed with .′s before the ⊢ , to represent the individual hypotheses (assumptions), so that a theorem

.... ⊢ "F1 == Exp F h"

for example, has seven hypotheses. The standard ML function

hyp:thm ⟶ form list

returns the list of hypotheses of a theorem.

The character # at the beginning of a line marks a user input. System responses are immediately after the terminating ;; and are followed by a blank line. The ML variable it holds the result of the last ML expression to be evaluated. The ML function

ssadd:th ⟶ simpset ⟶ simpset

adds a theorem to a simpset.

```
LET GL1,P1 = TAC1 GOAL1;;
GL1 = [ ] : (GOAL LIST)
P1 = - : PROOF

#LET RES1 = P1[ ];;
RES1 = ..]-"G == EXP F H" : THM
```

```
#TAC2B GOAL2';;
[ ],- : ((GOAL LIST) # PROOF)

#(SND IT)[ ];;
....]-"F1 << G" : THM

#HYP IT;;
["F1 == FIX(\F1'.\x.\z.\s.EQ s NIL=>z|(P x=>F1'(HD s)(H(z, F x))(TL s
)|F1'(G1 x)z(CONS(G2 x)s))))"; "G == FIX(\G'.\x.\z.\s.EQ s NIL=>z|G'(H
D s)(H(z, F x))(TL s))"; "F == FIX(\F'.\x.P x=>F x|H(F'(G1 x), F'(G2
x)))"; "!A. !B. !C. H(H(A, B), C) == H(A, H(B, C))"] : (FORM LIST)
```

```
#LET GL5,P5 = TACLEMMA GOAL5;;
GL5 = [ ] : (GOAL LIST)
P5 = - : PROOF

#LET RES5 = P5[ ];;
RES5 = ..]-"!x. !s. F1 x UU s == UU" : THM

#LET A,B,C = GOAL3;;
A = "G << F1" : FORM
B = - : SIMPSET
C = [ ] : (FORM LIST)

#LET B = SSADD RES5 B;;
B = - : SIMPSET

#LET GOAL3 = A,B,C;;
GOAL3 = "G << F1",-,[ ] : GOAL

#TAC3 GOAL3;;
[ ],- : ((GOAL LIST) # PROOF)

#LET GL3,P3 = IT;;
GL3 = [ ] : (GOAL LIST)
P3 = - : PROOF

#LET RES3 = P3[ ];;
RES3 = ......]-"G << F1" : THM
```

```
#LET RES6 = TAC6 GOAL6;;
RES6 = [ ],- : ((GOAL LIST) # PROOF)

#
LET GL6,P6 = RES6;;
GL6 = [ ] : (GOAL LIST)
P6 = - : PROOF

#LET RES6 = P6[ ];;
RES6 = ..]-"EXP F H X E(LIST DUMMY) == F X" : THM
```

```
#LET A,B,C = GOAL7;;
A = "F1 x E(LIST DUMMY) == F x" : FORM
B = - : SIMPSET
C = [ ] : (FORM LIST)

#LET B = SSADD RES4 BASICSS;;
B = - : SIMPSET

#LET B = SSADD RES6 B;;
B = - : SIMPSET

#LET GOAL7 = A,B,C;;
GOAL7 = "F1 x E(LIST DUMMY) == F x",-,[ ] : GOAL

#
LET GL7,P7 = SIMPTAC GOAL7;;
GL7 = [ ] : (GOAL LIST)
P7 = - : PROOF

#LET RES7 = P7[ ];;
RES7 = ........]-"F1 x E(LIST DUMMY) == F x" : THM
```

Observe that the eventual result has eight hypotheses, representing the eight initial assumptions.

The use of BYTAC2 is demonstrated below. BYTAC2 generates a list of two subgoals (using the ML function gentok, see [15], A3a, to generate a new name, G7859, here) and a proof.

```
#BYTAC2 TH3 GOAL4;;
["G7859 << F1",-,[ ]; "F1 << G7859",-,[ ]],- : ((GOAL LIST) # PROOF)

#LET [G1;G2],P = IT;;
G1 = "G7859 << F1",-,[ ] : GOAL
G2 = "F1 << G7859",-,[ ] : GOAL
P = - : PROOF
```

The proof p depends on a function BYRULE. Suppose we have proved

$\vdash G7859 \sqsubseteq F1$

$\vdash F1 \sqsubseteq G7859$

We can then test the proof part of BYTAC2. A theorem list, thl,

contains these two theorems.

```
thl = [ ]-"G7859 << F1"; ]-"F1 << G7859" ] : (thm list)

#p thl;;
..]-"F1 == Exp F h" : thm

#let res = it;;
res = ..]-"F1 == Exp F h" : thm

#hyp res;;
["Exp == FIX(\Exp'.\F.\h.\x.\z.\s.EQ s NIL=>z|Exp' F h(HD s)(h(z, F x
))(TL s))"; "G7859 == FIX(\G7859.\x.\z.\s.EQ s NIL=>z|G7859(HD s)(h(z
, F x))(TL s))"] : (form list)

#goal4;;
"F1 == Exp F h",-,[ ] : goal
```

The theorem proved thusly has as hypotheses the definition of Exp and a theorem defining G7859, added by BYRULE, which is called in evaluating (p thl).

For completeness, the ML code for BYRULE and BYTAC2 is shown below. Not all of the functions and features of ML used have been explained; the curious reader will have to consult [15], (or better [13], as the functions were defined for use in the older LCF, although the differences, in this case, should not be many). Some necessary auxilliary functions follow the two mains ones.

```
let BYLAW flist th =
   let PHI1 = snd (destcomb (rhs (concl th)))
   in let H', PHIofFs = destabs PHI1
      and F'list = map (\e. mkvar (gentok (), typeof e)) flist
   in let E       = mkvar (gentok (), (itlist (\ty1.\ty2.mkfuntype (ty1,t
y2))
          (map typeof flist)
          (typeof H'))    )
   and pairlist = combine (F'list, flist)
      in let PHI2 = mkabsl (E . F'list)
                          (substinterm  ((mkcombl (E. F'list),H') .
                                             pairlist) PHIofFs)
        and f = mkvar (gentok (), typeof H')
        and g = mkvar (gentok (), typeof E)
        in let w' = mkequiv (f, mkcombl (g . flist))
           in let basis = BMKBASIS flist (typeof E)
              and step  = BMKSTEP PHIofFs PHI1 PHI2 f g w' H' flist
              in let th' = INDUCT [PHI1,f;PHI2,g] w' (basis,step)
                 in TRANS (th, th');;
```

```
LET BYTAC2 TH:TACTIC (W;SS;FML) =
   LET F = LHS W
   AND E . FLIST = DESTCOMBL (RHS W)
   AND H = MKVAR (GENTOK (), TYPEOF (RHS W))
   IN LET W2 = MKINEQUIV (H, F)
      AND W3 = MKINEQUIV (F, H)
      AND (E' . F'LIST), BODY = DESTABSL1 (NIL,
                                  (SND (DESTCOMB (RHS (CONCL TH))))
                                  ((LENGTH FLIST) + 1)
      IN LET BODY' = SUBSTINTERM (COMBINE (FLIST, F'LIST))
                        (SUBSTINTERM [H, MKCOMBL (E' . F'LIST)]
                                  BODY)
      IN LET TYP = MKFUNTYPE (TYPEOF (MKABS (H,BODY')),TYPEOF H)
         IN LET TH' = ASSUME (MKEQUIV (H, MKCOMB (MKCONST
                        (`FIX`,TYP),MKABS (H,BODY'))))
            IN LET SSPLUSTH' = SSADD TH' SS
            IN [W2, SSPLUSTH', FML; W3, SSPLUSTH', FML],
               (\THL. LET [TH2;TH3] = THL
                     IN LET TH1 = SUBS [SYM TH] (BYLAW FLIST TH')
                        IN TRANS (SYNTH (TH3,TH2),TH1));;



LETREC MKCOMBL L =
   NULL L => FAIL |
   NULL (TL L) => FAIL |
   NULL (TL (TL L)) => MKCOMB (HD L, HD (TL L)) |
   MKCOMBL ((MKCOMB ((HD L), (HD (TL L)))) . TL (TL L)) ?
      FAILWITH `MKCOMBL`;;

% [F1; ...;FN]
  ----------------------
  F1 ... FN                    %

LETREC DESTCOMBL T =
   LET FIRST, LAST = DESTCOMB T
   IN ISCOMB FIRST => (DESTCOMBL FIRST) @ [LAST] |
                   [FIRST;LAST] ? FAILWITH `DESTCOMBL`;;

% F1 ... FN
  -------------------
  [F1; ...;FN]               %


LET BMKBASIS FLIST TY =
SYM   (MINAPL FLIST TY) ;;

LET BMKSTEP PHIOFFS PHI1 PHI2 F G W' H' FLIST =
   SUBS [SYM (BETACONV (MKCOMB (PHI1, F)));
         SYM (BETACONVL ([PHI2;G] @ FLIST))]
        (SUBSOCCS [[2], ASSUME W']
                  (REFL (SUBSTINTERM [F, H'] PHIOFFS)));;
```

```
LETREC APTHML TH L =
    NULL L => TH |
    APTHML (APTHM TH (HD L)) (TL L) ? FAILWITH `APTHML`;;

% |- F == G     [x1; ...;xN]
  ---------------------------------
  |- F x1 ... xN == G x1 ... xN          %

LETREC BETACONVL L =
    NULL L => FAIL |
    NULL (TL L) => FAIL |
    NULL (TL (TL L)) => BETACONV (MKCOMB (HD L, HD (TL L))) |
        LET FIRSTSTEP =  BETACONV (MKCOMB (HD L, HD (TL L)))
        IN TRANS (APTHML FIRSTSTEP (TL (TL L)),
                  BETACONVL ((RHS (CONCL (FIRSTSTEP))) . (TL (TL L)) )) ?
                  FAILWITH `BETACONVL`;;

% [(\x1...\xN.T);Y1;...;YN]
  ----------------------------------------
  |- (\x1...\xN.T) Y1 ... YN == T[YI/XI]   %

LETREC MINAPL FLIST TY =
    NULL FLIST => FAIL  |
    NULL (TL FLIST) => MINAP (MKCOMB (MKCONST (`UU`,TY), HD FLIST)) |
    LET FIRSTSTEP = MINAP (MKCOMB (MKCONST (`UU`,TY), HD FLIST))
    IN TRANS (APTHML FIRSTSTEP (TL FLIST),
              MINAPL (TL FLIST) (SND (DESTFUNTYPE TY))) ?
    FAILWITH `MINAPL`;;

% [F1;...;FN]  TY
  ----------------------------------------
  |- (UU:TY) F1 ... FN == UU              %


LETREC MKABSL L T =
    NULL L => T |
    MKABS (HD L, MKABSL (TL L) T) ? FAILWITH `MKABSL`;;

% [x1;...;xN]
  ------------------------------
  \x1...\xN.T                        %

LETREC DESTABSL (L, T) =
    ISABS T => LET VAR, REST = DESTABS T
               IN DESTABSL (L @ [VAR], REST) |
               (L, T) ? FAILWITH `DESTABSL`;;

% [x1;...;xN],  (\Y1...\YN.T)
  ----------------------------------
  [x1;...xN;Y1;...YN], T                 %

LETREC DESTABSL1 (L,T) N = LENGTH L = N => (L,T) |
    ISABS T => LET VAR,REST = DESTABS T
               IN DESTABSL1 ((L @ [VAR]),REST) N |
               (L,T) ? FAILWITH `DESTABSL1`;;

% AS DESTABSL, BUT STOPS WHEN LIST IS OF LENGTH N     %
```

Implementing Iterated Induction

In view of the complications mentioned, we do not give details of the Russell compiler proof here. We do, however, show the code for the rule of iterated induction used in the proof of Theorem 3.1b, ITINDUCT, and for the corresponding tactic, ITINDUCTAC. Auxilliary functions follow the two main ones. Again, not all functions or features of ML will have been explained, and the reader is again referred to [15] or [13]. In the rule, a new formula, basis and step are constructed, and ordinary induction is done.

```
let ITINDUCT Fflist w (basislist,step) =
    let n = length basislist
    and flist = map snd Fflist
    in let substlist = combine ((map mkcomb Fflist),flist)
       in let hypstep = mkitlist w (n - 1) substlist
          in let w' = mkconjl hypstep
             in let basis' = CONJL basislist
                and step' = SEL2 (MP (DISCHL hypstep
                                          (conjth hypstep step))
                                  (ASSUME w'))
                in SEL1 (INDUCT Fflist w' (basis',step')) ?
   failwith `ITINDUCT`;;


let ITINDUCTAC thl n: tactic (w,ss,fml) =
   let Flist = map (\th. (snd (destcomb (rhs (concl th))))) thl
   and flist = map (\th. (lhs (concl th))) thl
   in let f'list = map (\f. variant (f,formlfrees (w.fml))) flist
      in let Ff'list = combine (Flist,f'list)
         and wvar = substinform (combine (f'list,flist)) w
         in let substlist = combine ((map mkcomb Ff'list),f'list)
            in let assump,stepform = destlistback nil
                                     (mkitlist wvar n  substlist )
            in let UUlist = map (\f. let ty = typeof f
                                     in (mkconst (`UU`,ty)))flist
               in let FUUlist = combine (Flist,UUlist)
                  in let UUsubstlist = combine ((map mkcomb FUUlist),
                                                UUlist)
                     and wUU = substinform (combine (UUlist,flist))w
   in let basisformlist = mkitlist wUU (n-1) UUsubstlist
      in let basisgoallist = map (\w.w,ss,fml) basisformlist
         and stepgoal = stepform,ss,(assump @ fml)
         in (basisgoallist @ [stepgoal]),
          (\thml. let basislist,step =
                           destlistback nil thml
                  in SUBS (map SYM thl)
                          (ITINDUCT Ff'list wvar (basislist,step))) ?
   failwith `ITINDUCTAC`;;
```

```
LETREC MKCONJL WLIST =
    NULL WLIST => FAIL !
    NULL (TL WLIST) => HD WLIST !
    MKCONJ (HD WLIST, MKCONJL (TL WLIST)) ? FAILWITH `MKCONJL`;;

% [W1;...;WN]
  -----------------
  W1 & ... &WN                         %

LETREC CONJTH WLIST TH =
    NULL WLIST => FAIL !
    NULL (TL WLIST) => CONJ (ASSUME (HD WLIST),TH) !
    CONJ (ASSUME (HD WLIST), CONJTH (TL WLIST) TH) ?
    FAILWITH `CONJTH`;;

% [W1;...;WN]          A |- W
  -------------------------------------------
  [W1;...;WN], A   |- W1 & ... & WN & W               %



LETREC MKITLIST W N SUBSTLIST =
    N = 0 => [W] !
    LET WW = SUBSTINFORM SUBSTLIST W
    IN W.(MKITLIST WW (N - 1) SUBSTLIST) ? FAILWITH `MKITLIST`;;

% W N [FUNI FI,FI]
  -----------------------------------------------------
  [W; W[FUNI FI/FI]; ... W[FUNI ↑ N FI/FI]]            %

LETREC DISCHL WLIST TH =
    NULL WLIST => TH !
    NULL (TL WLIST) => DISCH (HD WLIST) TH !
    DISCH (HD WLIST) (DISCHL (TL WLIST) TH) ? FAILWITH `DISCHL`;;

% [W1;...;WN] WHICH IS WLIST       A |- W (WHICH IS TH)
  ---------------------------
  A - [W1;...;WN]|- W1 & ... & WN IMP W               %

LETREC DESTLISTBACK L1 L =
    NULL L => FAIL !
    NULL (TL L) => (L1, HD L) !
    DESTLISTBACK  (L1 @ [HD L]) (TL L) ? FAILWITH `DESTLISTBACK`;;
% [A1;...AN]   L
  --------------------
  [A1;...AN-1], AN              %
```

```
LETREC CONJL THL =
    NULL THL => FAIL !
    NULL (TL THL) => HD THL !
    CONJ (HD THL, (CONJL (TL THL))) ? FAILWITH `CONJL`;;

% [TH1;...;THN]
  ------------------
    TH1 & ....& THN           %
```

The following fragments of an LCF session demonstrate the use of the tactic and rule. We take, as an example, the definition of the function F from the Counter problem (Chapter 2), which is called th1 during this session. We apply ITINDUCTAC to do 3-ary iterated induction on F. We thus obtain a list of four subgoals; three basis cases, and an inductive step with three hypotheses; and a proof. The theorem, th1, and the goal are shown, then the tactic is applied to a goal as shown below. (Expo is called Nexpo here -- but this is not the way to solve the goal, just an example of the use of the tactic.

```
#LET THL = [TH1];;
THL = [.]-"F == FIX(\F'.\x.P x=>F x!F'(H(F'(G x))))"] : (THM LIST)

#LET N = 3;;
N = 3 : INT

#H,SS,FML;;
"F1 == NExpo F H",-,[] : GOAL

#ITINDUCTAC THL N (H,SS,FML);;
["F1 == NExpo UU H",-,[]; "F1 == NExpo((\F'.\x.P x=>F x!F'(H(F'(G x))
))UU)H",-,[]; "F1 == NExpo((\F'.\x.P x=>F x!F'(H(F'(G x))))((\F'.\x.P
x=>F x!F'(H(F'(G x))))UU))H",-,[]; "F1 == NExpo((\F'.\x.P x=>F x!F'(
H(F'(G x))))((\F'.\x.P x=>F x!F'(H(F'(G x))))((\F'.\x.P x=>F x!F'(H(F
'(G x))))F')))H",-,["F1 == NExpo F' H"; "F1 == NExpo((\F'.\x.P x=>F x
!F'(H(F'(G x))))F')H"; "F1 == NExpo((\F'.\x.P x=>F x!F'(H(F'(G x))))(
(\F'.\x.P x=>F x!F'(H(F'(G x))))F'))H"]],- : ((GOAL LIST) # PROOF)
```

This produced a goal list and a proof. The proof was named p.
Suppose we have proved the following theorems, which achieve the three subgoals:

```
THA = ]-"F1 == NExpo UU H" : THM

THB = ]-"F1 == NExpo((\F'.\x.P x=>F x!F'(H(F'(G x))))UU)H" : THM
```

```
THC = ]-"F1 == MExpo((\F'.\x.P x=>F x|F'(H(F'(G x))))((\F'.\x.P x=>F
x|F'(H(F'(G x))))UU))H" : THM
```

```
THD = ...]-"F1 == MExpo((\F'.\x.P x=>F x|F'(H(F'(G x))))((\F'.\x.P x=
>F x|F'(H(F'(G x))))((\F'.\x.P x=>F x|F'(H(F'(G x))))F')))H" : THM
```

Finally, we apply the proof p to a list containing these theorems, to achieve our goal:

```
#P[THA;THB;THC;THD];;
.]-"F1 == MExpo F H" : THM

#HYP IT;;
["F == FIX(\F'.\x.P x=>F x|F'(H(F'(G x))))"] : (FORM LIST)
```

(The three hypotheses of thd are shown in the formula list below.)

```
["F1 == MExpo F' H"; "F1 == MExpo((\F'.\x.P x=>F x|F'(H(F'(G x)))
)F')H"; "F1 == MExpo((\F'.\x.P x=>F x|F'(H(F'(G x))))((\F'.\x.P x=>F
x|F'(H(F'(G x))))F'))H"] : (FORM LIST)
```

Bibliography

ACM abbreviates Association for Computing Machinery,
SIAM, Society for Industrial and Applied Mathematics,
SAIL, Stanford Artificial Intelligence Laboratory,
EUCSD, University of Edinburgh Computer Science Department,
and IRIA, Institut de Recherche d'Informatique et d'Automatique

1. L. Aiello, M. Aiello and R. Weyhrauch, Pascal in LCF: Semantics and Examples of Proof, Theoretical Computer Science 5, p. 135-177, 1977

2. L. Aiello, M. Aiello, G. Attardi and G. Prini, PPC (Pisa Proof Checker): A tool for experiments in theory of proving and mathematical theory of computation, Annales Societatis Mathematicae Polonae, Series IV: Fundamenta Informaticae I, p. 251-275, 1977

3. R. S. Boyer and J S. Moore, Proving theorems about LISP functions, Journal of the ACM 22, 1, 1975

4. R. Burstall and J. Darlington, Some Transformations for Developing Recursive Programs, Journal of the ACM 24,1, Jan. 1977

5. R. Burstall and P. Landin, Programs and their Proofs: an Algebraic Approach, Machine Intelligence 4, B. Meltzer and D. Michie, eds., Edinburgh University Press, Edinburgh, Scotland, 1969

6. A. J. Cohn, High Level Proof in LCF, Proceedings, Fourth Workshop on Automated Deduction, Austin, Texas, Feb. 1-3, 1979, and also as EUCSD Internal Report CSR-35-78, Nov. 1978

7. R. L. Constable and S. Johnson, A PL/CV Precis, Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, San Antonio, Texas, Jan. 29-31, 1979

8. J. Darlington and R. Waldinger, Case Studies in Program Transformation and Synthesis, unpublished note, Stanford Research Institute, Menlo Park, Ca., Oct. 1978

9. R. E. Filman and R. W. Weyhrauch, An FOL Primer, SAIL Memo AIM-188, Sept. 1976

10. R. W. Floyd, Assigning Meanings to Programs, Proceedings of the Symposium of the American Mathematical Society, Vol. 19, p. 19-32, 1967

11. D. A. Giles, The Theory of Lists in LCF, EUCSD Internal Report CSR-31-78, Oct. 1978

12. M. Gordon, Denotational Description of Programming Languages, Springer-Verlag, New York, 1979

13. M. Gordon, R. Milner and C. Wadsworth, Edinburgh LCF, EUCSD Internal Report CSR-11-77, Sept. 1977

14. M. Gordon, R. Milner, F. L. Morris, M. Newey and C. Wadsworth, A metalanguage for interactive proof in LCF, Fifth ACM SIGACT-SIGPLAN Conference on Principles of Programming Languages, Tuscon, Arizona, 1978

15. M. Gordon, R. Milner and C. Wadsworth, Edinburgh LCF: A Mechanised Logic of Computation, Springer-Verlag, In Press

16. C. A. R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM, Vol. 12, 10, p. 576-582, Oct. 1969

17. G. Huet and B. Lang, Proving and Applying Program Transformations Expressed with Second Order Patterns, Draft, IRIA-Laboria, France, Feb. 1977

18. D. M. Kaplan, Correctness of a Compiler for Algol-Like Programs, SAIL Memo No. 48, July 1967

19. R. London, Correctness of a Compiler for a LISP Subset, Proceedings of the ACM Conference on Proving Assertions about Programs, New Mexico State University, Las Cruces, New Mexico, Jan. 1972

20. R. London, Correctness of Two Compilers for a LISP Subset, SAIL Memo No.151, Oct. 1971

21. D. S. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak and W. L. Scherlis, Stanford Pascal Verifier User Manual, Stanford Verification Group Report No. 11, Stanford University Computer Science Department Report STAN-CS-79-731, Edition 1, Stanford University, March 1979

22. Z. Manna and R. Waldinger, Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness, SAIL Memo AIM-281, June 1976

23. J. McCarthy and J. A. Painter, Correctness of a Compiler for Arithmetic Expressions, Proceedings of a Symposium in Applied Mathematics, 19, Mathematical Aspects of Computer Science, p. 33-41, American Mathematical Society, J. Schwartz, ed., Providence, Rhode Island, 1967

24. R. Milne, Verifying the Correctness of Implementations, unpublished notes, Advanced Seminar on Semantics, sponsored by IRIA-Laboria, Antibes, France, 1977

25. R. Milne and C. Strachey, A Theory of Programming Language Semantics, Chapman and Hall, London, 1976

26. R. Milner, Logic for Computable Functions: Description of a Machine Implementation, SAIL Memo AIM-169, 1972

27. R. Milner, Implementation and Application of Scott's Logic for Computable Functions, Proceedings of the ACM Conference on Proving Assertions about Programs, SIGPLAN Notices 7, 1, 1972

28. R. Milner and R. Weyhrauch, Proving Compiler Correctness in a Mechanised Logic, Machine Intelligence 7, B. Meltzer and D. Michie, eds., Edinburgh University Press, Edinburgh, Scotland, 1972

29. R. Milner, L. Morris and M. Newey, A Logic for Computable Functions with Reflexive and Polymorphic Types, EUCSD LCF Report No. 1, Jan. 1975

30. R. Milner, LCF: A Methodology for Performing Rigorous Proofs about Programs, Proceedings of the First Symposium on Mathematical Foundations of Computer Science, Amagi, Japan, 1976

31. R. Milner, Program semantics and mechanized Proof, Proceedings of the Second Course on Foundations of Computer Science, Amsterdam, 1976

32. R. Milner, A Theory of Type Polymorphism in Programming, Journal of Computer and Systems Sciences 17, 1978

33. R. Milner, LCF:A Way of doing proofs with a machine, EUCSD Internal Report CSR-41-79, May 1979, and also Proceedings of the Eighth Mathematical Foundations of Computer Science Symposium, Olomouc, Czechoslovakia, 1979

34. F. L. Morris, Correctness of translations of programming languages, Stanford University Computer Science Department Memo CS-72-303, Stanford University, 1972

35. F. L. Morris, Advice on structuring compilers and proving them correct, Proceedings of the ACM Symposium on Principles of Programming Languages, Boston, Mass., 1973

36. M. C. Newey, Formal Semantics of LISP with Applications to Program Correctness, SAIL Memo AIM-257, Jan. 1975

37. M. C. Newey, Axioms and Theorems for Integers, Lists and Finite Sets in LCF, SAIL Memo AIM-184, Jan. 1973

38. M. C. Newey, Proving Properties of Assembly Language Programs, unpublished note, Computer Centre, Australian National University, Australia

39. D. M. R. Park, Fixpoint Induction and Proofs of Program Properties, Machine Intelligence 5, B. Meltzer and D. Michie, eds., American Elsevier, New York, 1970

40. V. R. Pratt, A Practical Decision Method for Propositional Dynamic Logic: Preliminary Report; Total Correctness of the KMP Pattern-Matcher; Role of Gentzen-type Systems in Mechanical Theorem-Proving, unpublished notes, M. I. T., 1978-1979

41. J. C. Reynolds, On the Relation Between Direct and Continuation Semantics, Second Colloquium on Automata, Languages and Programming, Saarbrucken, Germany, July 29-Aug. 2, 1974

42. B. D. Russell, Implementation Correctness Involving a Language with GOTO statements, SIAM Journal of Computing, Vol. 6, No. 3, Sept. 1977

43. D. S. Scott and C. Strachey, Towards a Mathematical Semantics for Computer Languages, Technical Monograph PRG-6, Oxford University, 1971

44. D. S. Scott, Data Types as Lattices, SIAM Journal of Computing, 4, 1976, p. 522-587

45. J. E. Stoy, Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, M. I. T. Press, Cambridge, Mass., 1977

46. C. Strachey and C. P. Wadsworth, Continuations -- a Mathematical Semantics for Handling Full Jumps, Technical Monograph PRG-11, Programming Research Group, Oxford University, 1974

47. J. N. Thatcher, E. G. Wagner and J. B. Wright, More advice on structuring compilers and proving them correct, Draft, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, Nov. 1978

48. P. Wegner, Programming Languages, Information Structures and Machine Organisation, McGraw Hill, London, 1971

49. R. W. Weyhrauch, A User's Manual for FOL, SAIL Memo AIM-235, July 1977

50. R. Weyhrauch and R. Milner, Program semantics and correctness in a mechanised logic, Proceedings of the USA-Japan Computer Conference, Tokyo, 1972