

Data Type Proofs using Edinburgh LCF.

Brian Quentin Monahan

Doctor of Philosophy

University of Edinburgh

1984



ABSTRACT

This thesis contains 4 case studies of machine proof using the Edinburgh LCF system. The theme connecting all of the case studies together concerns the formulation and proof of correctness criteria for simple monoid data types.

The purpose of performing proofs in LCF is not necessarily to prove original mathematical theorems or to discover new proofs of those already known. Rather, such case studies examine the task of formalising mathematical theories within a specific formal deduction system, PPLAMBDA; the formalisation and performance of formal logical arguments within an LCF environment, and the study of the interaction required to generate such proofs.

Chapter 1 and 2 are introductory chapters. The first chapter gives a detailed introduction to Edinburgh LCF, putting particular emphasis on the systematic distinction between Object Language and Meta Language. The second chapter has a more mathematical character, focusing on Domain Theory, Continuous Algebras and the concept of Free and Initial object in a category. Also included is a notion of effective congruence, which is used in Chapter 4 to formulate the correctness of a data type simulated by another.

The third chapter gives 3 related case studies, each of which show that a simple algebra for strings is an appropriate free monoid in a particular category (with respect to an arbitrary collection of generators). The higher-type polymorphic capabilities of PPLAMBDA are used to formulate this result in a uniform manner for each case study.

The fourth chapter contains a major case study in which the correctness of a simple representation of multisets is shown. The approach used is to simulate the multiset algebra in terms of a simple list algebra together with an algorithmically specified function for computing when two lists are equivalent as multisets. The correctness criteria formally proven in LCF state that this function is an effective equivalence, that it is a congruence for the list operations given, and that the properties of the simulation hold, as specified by the equations. Two of the main theorems are then re-proven using a combination of resolution-oriented tactics. Finally, a rigorous set-theoretical proof of freeness of the multiset construction is given.

The fifth chapter describes various tools written in ML which were developed during the course of this research. These include a domain equation axiomatisation package and a structural induction package. The resolution-oriented tactics used in Chapter 4 are also described in detail, as well as tactics for performing backchaining forms of resolution.

The Appendices contain a listing of the theory files generated when using the LCF system, as well as a listing of some of the ML functions used herein.

Declaration.

This thesis was composed by myself. The work presented herein was conducted under the guidance of my supervisor, Robin Milner.

Acknowledgements.

I would first of all like to thank my supervisor, Robin Milner for his (extremely) patient and careful guidance and assistance during the course of the research reported here. I am also very grateful to Rod Burstall, who supervised and guided me initially.

During my period of study in Edinburgh, many people in the Department of Computer Science helped increase my appreciation and understanding of ideas. First of all, I would like to mention David Schmidt who was always willing to discuss LCF and provided a good deal of very helpful and insightful comments. Many thanks also go to Alan Mycroft who did a rapid and much appreciated conversion of the LCF system to Rutgers UCI LISP. David Rydeheard gave various helpful remarks on a draft of part of this thesis.

In addition, I would also like to thank the following for many helpful remarks and general discussions:- Peter Dybjer, Alberto Pettorossi, Don Sannella, Oliver Schoett, Colin Stirling, Allen Stoughton from Computer Science, as well as Leon Sterling and Lincoln Wallen in the Department of Artificial Intelligence.

I am indebted to Cliff Jones and Howard Barringer at the University of Manchester for their encouragement and the generous use of facilities in order to complete the writing of this thesis. I also wish to thank my colleagues at Standard Telecommunications Laboratories Ltd. for their continued interest during this period. I am very thankful to Mrs. Julie Hibbs for typing the bulk of a very long and awkward manuscript.

The SERC provided financial support from October 1979 to October 1982 via studentship no:- 79313447, and sponsored my attendance at the International Summer School on Theoretical Foundations of Programming Methodology, Marktoberdorf, 1981.

Finally, I am deeply grateful to Eleanor Broadhurst who gave me much needed support and practical assistance, as well as all of my family and personal friends who provided help and inspiration when all else failed.

"There cannot be a philosophy, there cannot even be a decent science, without humanity."

J.Bronowski,

Forward to *"The Ascent of Man"*

"Truth is like a vast tree, which yields more and more fruit, the more you nurture it. The deeper the search in the mine of truth, the richer the discovery of the gems buried there, in the shape of openings or an ever greater variety of service."

M.K.Gandhi

"An Autobiography or The story of My experiments with Truth"

"In this way, mankind stumbles on in its task of understanding the world"

A.N.Whitehead, *The Function of Reason.*

CONTENTS

Declaration.

Acknowledgements.

Contents.

Chapter 1 Introduction to Edinburgh LCF.

- 1.1 Introduction.
 - 1.1.1 Previous case studies: Stanford LCF.
 - 1.1.2 Previous case studies: Edinburgh LCF.
 - 1.1.3 Recent developments in LCF.
- 1.2 Edinburgh LCF: The Object Language, PPLAMBDA.
 - 1.2.1 Forms.
 - 1.2.2 Terms.
 - 1.2.3 Types.
 - 1.2.4 Well formedness of terms.
 - 1.2.5 Deduction rules for PPLAMBDA.
- 1.3 Edinburgh LCF: The Meta Language, ML.
 - 1.3.1 An informal introduction to ML.
 - 1.3.2 PPLAMBDA in ML.
 - 1.3.3 Data types for PPLAMBDA syntax.
 - 1.3.4 Representing the semantics of PPLAMBDA in ML.
 - 1.3.5 Auxiliary ML functions for PPLAMBDA.
 - 1.3.6 Quotation and anti-quotation.
 - 1.3.7 LCF display conventions.
 - 1.3.8 Theories in LCF.
- 1.4 Proof using LCF.
 - 1.4.1 Forward deduction.
 - 1.4.2 Simplification.
 - 1.4.3 Goal oriented reasoning: tactics.
 - 1.4.4 Basic tactics for PPLAMBDA.

Chapter 2 Mathematical Background.

- 2.1 Domain Theory.
 - 2.1.1 Domain operators.
 - 2.1.2 Standard PPLAMBDA theories.
 - 2.1.3 Properties of isomorphisms in CPO.
 - 2.1.4 Domain equations.
 - 2.1.5 Extended domain equations.
 - 2.1.6 Axiomatising domain equations in LCF.
 - 2.1.7 Limit points and domain equations.
 - 2.1.8 Polymorphism.
 - 2.1.9 Computational intuition underlying domain theory.

- 2.2 Basic Category Theory.
- 2.2.1 Concrete Categories.
- 2.2.2 Commuting Diagrams and Universal Constructions.
- 2.2.3 Free and initial objects.

- 2.3 Continuous algebras.
- 2.3.1 Free and initial algebras.
- 2.3.2 Anarchic Σ -algebras and domain equations.
- 2.3.3 The initiality functional.

- 2.4 Continuous quotients.
- 2.4.1 Equivalence predicates.
- 2.4.2 General properties of equivalence predicates.
- 2.4.3 Effective congruences.
- 2.4.4 The relation represented by an equivalence predicate.

- 2.5 Monoids.
- 2.5.1 Categories of monoids.
- 2.5.2 An induction principle for monoids.

Chapter 3 Monoid case studies.

- 3.1 The First Monoid Case Study.
- 3.1.1 A domain of lists.
- 3.1.2 Concatenation and unit.
- 3.1.3 The Freeness Functional.

- 3.2 Formalisation in PPLAMBDA.
- 3.2.1 The axiomatisation of lists.
- 3.2.2 Formalising concatenation and unit.
- 3.2.3 Formalising the Freeness Functional.

- 3.3 The Second Case Study.
- 3.3.1 The Theory L1.
- 3.3.2 The Theory LFUN1.
- 3.3.3 The Theory LFREE1.

- 3.4 The Third Case Study.
- 3.4.1 The Theory L2.
- 3.4.2 The Theory LFUN2.
- 3.4.3 The Theory LFREE2

- 3.5 Summary.

Chapter 4 The Multiset case study.

- 4.1 Commutative monoids and multisets.
- 4.2 Simulating multisets.

- 4.3 Theory structure and basic definitions.
- 4.4 Correctness criteria.
 - 4.4.1 Equivalence requirements.
 - 4.4.2 Congruence requirements.
 - 4.4.3 Validity requirements.
- 4.5 Further preliminary details and remarks.
 - 4.5.1 The form of initial goals.
 - 4.5.2 Informal notation.
 - 4.5.3 Auxiliary lemmas.
 - 4.5.4 Tactic composition notation.
 - 4.5.5 Additional tactics.
 - 4.5.6 Presentation of proofs.
- 4.6.1 Multisets: Basic results.
- 4.6.2 Multisets: Symmetry of EqBA.
- 4.6.3 Multisets: Transitivity of EqBA.
- 4.6.4 Multisets: Congruence and Commutativity of Append using EqBA.
- 4.6.5 Multisets: Discussion.
- 4.7. Case studies using resolution oriented tactics.
 - 4.7.1 Auxiliary tactics.
 - 4.7.2 The resolution case studies.
 - 4.7.2.1 First resolution case study.
 - 4.7.2.2 Second resolution case study.
 - 4.7.3 Discussion.
- 4.8 Showing simulation correctness.

Chapter 5 Proof construction aids.

- 5.1 An LCF axiomatisation package.
 - 5.1.1 Using the axiomatisation package.
 - 5.1.2 Discussion.
- 5.2 Structural induction and case analysis package.
 - 5.2.1 Generating structural induction tactics.
 - 5.2.2 Generating case analysis tactics.
- 5.3 Special induction tactics.
 - 5.3.1 Monoid induction.
 - 5.3.2 LINDTAC.
- 5.4 Resolution oriented tactics.
 - 5.4.1 The original RESTAC.
 - 5.4.2 Limitations of Cohn's RESTAC.
 - 5.4.3 LINEARRESTAC and a (new) RESTAC.
 - 5.4.4 The function RESOLVE.
 - 5.4.5 Unification.
 - 5.4.6 Canonicalisation.
 - 5.4.7 Backchaining forms of resolution tactics.

- 5.5 METARULE and METATAC.
- 5.6 Simplification tactics.

Conclusions.

References.

- Appendix 1 A table of standard PPLAMBDA constants.
- Appendix 2 Machine processable concrete syntax for PPLAMBDA.
- Appendix 3 Listing of LCF theories used and generated.
- Appendix 4 Selected ML definitions.
- Appendix 5 The Axiomatisation and Structural Induction Packages.

Chapter 1

Introduction to Edinburgh LCF.

1.1 Introduction.

A general account is given of the background and developments in LCF, including a short discussion of some previous case studies. The basic form of the Edinburgh LCF system is discussed in terms of its principal parts; the object-language, PPLAMBDA, and the meta-language, ML. Basic techniques for performing proof using LCF are then introduced. The first such technique is forwards deduction via the application of inference rules to already known facts. The second technique introduces a goal-oriented style of deduction, which encourages the top-down development of proofs, by working backwards from the formula to be proven towards more easily solved subgoals. The tools used for this kind of goal-directed proof are called tactics.

1.1.1 Previous case studies: Stanford LCF.

LCF (or "Logic for Computable Functions") is a formal deductive system for reasoning about functionals of higher type. Originally, it developed out of Dana Scott's early work on the foundations of denotational semantics and it was introduced in an unpublished paper of his in 1969. This system was further developed by Robin Milner into a logic for general reasoning about the typed λ -calculus and the Scott approximation ordering upon domains.

The logic was first implemented by Milner during 1972 in the form of an (interactive) proof checking program. This program later came to be known as Stanford LCF and was documented in [Milner72a].

Users of Stanford LCF worked by stating goals, as formulas to be achieved, and introducing basic postulates from which these goals would be proven. A proof attempt consisted of the successive application of rules of inference (as commands) to either goals or known facts. In this way, goals could be decomposed into further subgoals, some of which might be directly achieved by facts already

available. Alternatively, further facts could be derived by using the rules of inference.

The program kept track of the user's proof attempt in the form of a proof tree; the root of this tree was the original goal and at the tips were subgoals remaining to be proven. Inference rules could also be applied "tactically" to these subgoals, producing further subgoals. When a subgoal corresponds to a known fact, it is said to have been achieved. When all of a goal's immediate subgoals has been achieved, then it, too, is achieved. Hence, a (successful) proof attempt is one which eventually reduces all subgoals to known facts and given postulates.

In [Milner72b], an indication was given of how Stanford LCF could be used to verify properties of programs formalised in terms of their semantics within the logic. Also see [Gordon82] for a brief introduction to Stanford LCF.

The Milner-Weyhrauch case study (see [MilnerWeyhrauch72] and [WeyhrauchMilner72]) investigated the correctness of a simple compiling algorithm, using Stanford LCF. A simple ALGOL-like source language, *S*, was given meaning in two ways. The first way was to define (by structural induction over the abstract syntax of *S*), a semantic function *MS* mapping the abstract syntax of (well-formed) programs into state-functions, where states are taken to be mappings from names to values. The second way was to define a compilation function (called "comp") that translated from the source language, *S*, into the target language, *T*. This is an elementary assembly language containing unrestricted jumps and instructions for manipulation a stack. This target language was given a semantics in terms of a function *MT* mapping target language into store-functions. Finally a "simulation" function, *SIMUL*, from the store-functions to state functions, was defined.

With this machinery, the formulation of compiler correctness could now be stated:- "The meanings of source programs and of target programs given by compilation were equivalent". This is mathematically stated as:-

$$MS = SIMUL \circ MT \circ comp$$

To prove this equality of meaning, an appeal to methods of proof from Universal Algebra was made. The idea was to show that both MS and (SIMUL o MT o comp) were homomorphisms of appropriate algebraic structures. Since the abstract syntax of source programs has the "word algebra", or "initiality", property that two homomorphisms from such an "algebra" are equal, this gave the desired result.

The motivation for using algebraic concepts was to put structure upon the proof attempt. The natural strategy for the attempt is to proceed by structural induction over the abstract syntax of programs. This factorises the overall proof into a number of separate proofs of individual lemmas. During these proofs, further lemmas were needed, including the McCarthy-Painter lemma [McCarthyPainter67] stating the correctness of expression compilation. Later on, in Chapters 3 and 4, similar algebraic ideas play an important part in the case studies given there.

Case studies such as the one discussed above showed that it was possible to use LCF to formalise some interesting problems from Computer Science. However, it soon became apparent that the style of explicit formal reasoning demanded by Stanford LCF was impractical, even for proofs of modest complexity. Often, short sequences of inference steps could frequently recur throughout a long proof; these had to be repeated each time by hand since Stanford LCF gave no means for abstracting upon such sequences. Also, since each proof was represented by an explicit proof tree, long proofs tended to give rise to large, rather unwieldy, data structures, resulting in poor performance and saturation of the available memory.

1.1.2 Previous case studies: Edinburgh LCF.

In 1973, Milner initiated the Edinburgh LCF project to design and build a fully-fledged proof-assistant system based upon the experience and ideas gained from the work on Stanford LCF. This system contains a general purpose programming language, ML, capable of defining arbitrary proof procedures that permitted abstraction on inference sequences (as well as many other things). This programming language is strongly typed (at "definition-time") so

that the set of proven formulae forms a distinguished type that is separated from the set of all well-formed formulae. The use of strong typing will obviate the system from having to maintain, and store, an explicit proof tree for each proven theorem (see Section 1.3.2).

The type-scheme used in LCF is of independent interest since it incorporates an strongly typed form of parametric polymorphism that can be statically checked, and was first reported in [Milner78].

The design of the programming language ML has been discussed in [Milner83]; ML itself was first reported in [GordonMilner78]. In [Gordon82], the implementation of logics using ML is discussed.

The logical calculus of Edinburgh LCF, known as PPLAMBDA, is an extension of that used in Stanford LCF. The major difference was the addition of more logical connectives (such as implication) and a richer type discipline that included the concept of polymorphism mentioned above. PPLAMBDA, with a basic set of inference rules, was originally introduced in [Milner et al 75]. The inference rules of Pure PPLAMBDA, as used in Stanford LCF, were formally shown to be valid in [Milner72c], with respect to standard models of Scott domain theory.

The final section of [Milner76] contains a sketch of how an LCF system might exploit a concept of goal-directed proof strategy, or tactic, within machine proofs of algorithm correctness. Stanford LCF incorporated some degree of goal-directed inference; however there was no clear distinction between subgoal generation and the validation of facts. Also illustrated there is a basic technique for deriving arbitrary structural induction schemas from the single Computational Induction schema possessed by LCF (See Section 2.1.6). This forms the basis of each of the "structural induction packages" that have been constructed for LCF; a discussion of these appear in Section 5.2 and in [Paulson83a].

Other innovations introduced within Edinburgh LCF is the goal-directed proof methodology of using tactics, general purpose simplification tools and a simple hierarchical database for LCF theories, which provides the means for recording given axioms, and any explicitly proven consequences, in a logically sound manner.

There have been a number of case studies of machine proof using Edinburgh LCF, some of which are briefly described below.

In [Giles78], David Giles illustrated how a simple theory of (finite and infinite) lists could be simply defined within PPLAMBDA. Simple properties of basic list processing functionals, such as "map", were given, as well as a short survey of work in automatic theorem proving.

Jacek Leszczylowski developed two short case studies in [Leszczylowski80a] and [Leszczylowski80b]; the first showed the definedness, or "termination", of a normalisation function for if-expressions. The second study discussed automated proofs within a theory of FP systems, as introduced by [Backus78]. More recently, Leszczylowski and Wirsing have developed an LCF-based system, called PAT, for performing proofs about algebraically specified abstract data types (See [LeszczylowskiWirsing82]).

Compiler-correctness formed the main theme of Avra Cohn's thesis, [Cohn79], and also [Cohn81]. In Chapter 2 of [Cohn79], three studies of transformation from recursive form to iterative form (i.e. using tail recursion functions) are given. Each proof was first motivated informally and then generated by using tactic expressions whose form followed major steps in those informal proofs. Moreover, each of these tactics appeared to be instances of an even more general tactic expression, called SCHEMATAC.

In some sense, this (parameterised) tactic summarised a general pattern of inference applicable to a specific domain of application (in this case, recursion elimination). The hope is that various general purpose, possibly parameterised, tactics can be found which, when suitably instantiated, encapsulate particular proof techniques, or "high level proof outlines", in specific application areas.

Cohn then gives a (complete) compiler correctness case study, based upon an informal proof given in [Russell77]. The source and target languages are similar to those used for the Milner-Weyhrauch case study. However, various details have been abstracted away to simplify the problem (e.g. the particular syntax of expressions in the high-level language or the mechanics of generating unique

labels in the low-level language). This permits the study to focus upon the essential aspects of compilation and the differences in the recursive structure of the high-level and low-level semantic functions.

It was also shown that, by proving certain well-chosen lemmas for use in simplification, the course of the main proof could be skilfully guided to avoid particularly awkward subgoals. Also, since each use of a simplification rule can do in a single step what might otherwise have required many steps, simplification can be used to generate economical, smoothly flowing proofs.

The final chapter of Cohn's thesis gives a detailed plan of a compiler-correctness proof for a source language that includes block structuring and the declaration and call of statically bound, user-defined procedures. This language is given a standard (direct) denotational semantics, using fixed points. The low-level target language gives an (idealised) instruction set for an abstract machine manipulating a procedure activation stack with static links between environment entries. The semantics of this language is basically operational, but formulated using the mathematical tools of domain theory. The essence of the problem is to formulate and show the equivalence of a semantics with nested fixed points to one in which recursive procedures are represented using a 'knot' in the activation stack. To simplify the proof and to avoid the need for recursively defined relations, two intermediate "bridging" semantics for the source language are introduced. The first uses closures, (pairs of programs and declaration-time environments) to represent store transformations. This naturally entails the use of a (reflexive) domain of environments mapping identifiers to closures. The second intermediate semantics uses an "abstract" activation stack from which appropriate store transformations are determined.

In [Cohn81], an account is given of the formalisation, and machine proof of, the equivalence of the standard, direct semantics and the closure semantics described above. These proofs contain the first use of RESTAC, a tactic that encapsulates a simplified form of Robinson's Resolution principle (See [Robinson65]).

Also [CohnMilner82] reports another major case study which investigated the correctness proof of a simple parsing algorithm, prompted by the study given by [Gloess80] for the Boyer-Moore theorem prover. This problem yields to a short tactical proof using RESTAC to "finish up" the remaining subgoals, after doing structural induction. A larger and more complex case study which formalised the correctness proof of a precedence parsing algorithm is contained in [Cohn82]. Here, too, RESTAC plays a major role.

In Section 4.7, similar Resolution-oriented tactics are applied within two tactical proofs taken from the multiset case study presented within Chapter 4. The development of these tactics is given in Section 5.4.

1.1.3 Recent developments in LCF.

During 1982-3 the Cambridge LCF system was developed by Michael Gordon and Lawrence Paulson, based upon the experience of using the Edinburgh LCF system.

Most of the changes lie mainly with further extensions to PPLAMBDA; rounding it out with more logical connectives (negation, disjunction) as well as the existential quantifier. Also added were arbitrary, typed relation constants, bringing PPLAMBDA nearer to traditional predicate calculi (See [Paulson83b]).

A more flexible, tactical approach to simplification has also been incorporated, and this has been reported in [Paulson83c]. Also the basic tactics and tacticals have been revised and extended; this work is documented in [Paulson83d].

Michael Gordon has used Cambridge LCF as a foundation for constructing another LCF derivative, known as LCF-LSM (where LSM stands for "Logic for Sequential Machines"). LSM is a logical calculus for stating and proving the correctness of sequential hardware systems and is a formalisation of the notation used in [Gordon81]. The LCF-LSM system is described in detail in [Gordon83a]. This system has successfully undertaken a graded series of practical examples, culminating in showing the correctness of an idealised 16-bit digital computer, (introduced in [Gordon81]). This case study is described in [Gordon83b].

Recently, Lawrence Paulson has successfully completed a major case study which automated Manna and Waldinger's proof of Robinson's Unification Algorithm (See [MannaWaldinger81]). The formalisation of this proof within LCF provided much of the impetus for the evolution from Edinburgh LCF to Cambridge LCF. The case study is reported in [Paulson84].

Recent work by Stefan Sokolowski has reviewed the use of Skolem constants and variables within the context of goal directed proof. In LCF at present, Skolem constants are implicitly represented by the occurrence of free variables in the hypotheses of theorems or the local assumptions of goals. Because of this, tactics which eliminate universal quantifiers from goal formulae (such as GENTAC) have to carefully avoid introducing variables into the goal formula that already occur freely within the local assumption list.

The need for Skolem variables arises when using an inference rule in the "forwards" direction on the assumption list of the goal, which also has some information "above the line" (i.e. in the rule's hypotheses) that is unavailable below the line (in the conclusion). At the end of a successful proof decomposition, the information is then available to determine the appropriate Skolem constant.

In [Sokolowski83a] (and to some extent [Schmidt83a]) various new forms of goal, tactic and tactical are proposed which contain structures for keeping track of Skolem variables introduced during the proof development phase. This information is then used when the validation component is evaluated so as to ensure that any instantiations of Skolem variables for Skolem constants occurs correctly.

In [Sokolowski83b], an LCF case study is presented which shows formally the soundness of Hoare's logic of programs (See [Hoare69]). Although the formalisation within PPLAMBDA used a non-Scott continuous function constant to formalise Hoare's "triples", no continuity-dependent PPLAMBDA inference rules were actually needed in proofs involving that constant. The performance of this proof provided the motivation for the work on Skolem variables mentioned above.

In [Schmidt83a], David Schmidt presents a general notation for describing the subgoal strategies of theorem provers using a natural deduction style of logical calculus. The language is a generalisation of the standard tactical language used within Edinburgh LCF. Various examples are presented of the application of this notation in formulating well-known theorem proving strategies. The parameterised tactic generator METATAC, presented in Section 5.5 is similar to the "backwards" decomposition tactic schema described in Section 3 of [Schmidt83a]; The resolution-oriented tactics described in Section 5.4 below are sophisticated examples of the forwards chaining tactics also discussed by Schmidt.

In [Schmidt83b], a natural deduction calculus is given for a particular formalisation of the Godel-Bernays theory of sets. It is argued that a "high level" inference rule representation of a particular axiomatic theory can assist the discovery of proofs and the subsequent performance of proofs within that theory. In addition, a simple algorithm for converting a wide class of established theorems into a procedural, inference rule format is described. In Section 5.5 below, a parameterised inference rule generator, called METARULE, is given for performing a similar transformation for a wide class of PPLAMBDA formulae.

Recently, Ketan Mulmuley (see [Mulmuley84]) succeeded in constructing a large semi-automatic LCF based theorem prover for showing the existence of recursively defined predicates over universal domains. This has a natural application to proving the equivalence of the denotational semantics of programming languages.

1.2 Edinburgh LCF: The Object Language, PPLAMBDA.

PPLAMBDA is an acronym for the Polymorphic Predicate typed LAMBDA calculus. It is a logical language for precisely stating assertions about values denoted by typed lambda expressions. The language itself is described here, and the manner in which it is represented in terms of the meta-language is deferred until Section 1.3.2.

The PPLAMBDA calculus is divided into 3 main sub-languages; a

language of forms for expressing assertions, a language of terms for describing values, and a language of types which describe the domains to which values belong.

1.2.1 Forms.

Each expression in the logical sub-language is called a form (short for formula). The structure of each form is conveniently described by the following BNF description:-

$$fm ::= \forall v. fm_1 \mid fm_1 \& fm_2 \mid fm_1 \supset fm_2 \mid tm_1 \equiv tm_2 \mid tm_1 \in tm_2$$

where fm , fm_1 , fm_2 are each forms, both tm_1 and tm_2 are terms of identical type, and v is a (typed) variable.

As usual, the symbols "&", " $\supset$ " and " $\equiv$ " stand for conjunction, implication and equivalence of values respectively, and " $\forall v. fm_1$ " stands for universal quantification.

The syntactic components of an implication are called, from left to right, the antecedent and conclusion respectively. Similarly, the syntactic components of an equation (or inequation) are called the lhs and rhs respectively.

Finally, each type denotes a complete, partially ordered set (or cpo) and the symbol " $\in$ " stands for the corresponding partial ordering.

1.2.2 Terms.

Each expression in the sub-language for describing values is called a term. The structure of each term is given by the following description:-

$$tm ::= v \mid c \mid tm_1(tm_2) \mid \lambda v. tm_1$$

where tm , tm_1 , tm_2 are each terms, v is a (typed) variable and c is a (typed) constant.

The notation " $tm_1(tm_2)$ " stands for the application of function tm_1 to argument tm_2 ; similarly, " $\lambda v. tm_1$ " stands for lambda abstraction on the (typed) variable, v . The statement of the well-formedness conditions on terms (and hence upon forms) makes

use of the notion of well-typed term; discussion of this is deferred until after the next section. Note that both constants and variables possess types. As usual, variables may be instantiated by any other term with identical type.

1.2.3 Types.

Each expression in the sub-language for describing domains is known as a type. The structure of each type is given as follows:-

$$\begin{aligned} \text{ty} ::= & \text{vty} \mid \text{tr} \mid \underline{\text{dot}} \mid \text{ty}_1 \\ & \mid \text{ty}_1 + \text{ty}_2 \mid \text{ty}_1 \# \text{ty}_2 \mid \text{ty}_1 \rightarrow \text{ty}_2 \\ & \mid (\text{ty}_1, \text{ty}_2, \dots, \text{ty}_n) \text{ty-op} \end{aligned}$$

where ty , ty_1 , $\text{ty}_2, \dots$ are all types expressions and vty is a type-variable (ranging over domains), and ty-op is any (user-defined) type operator symbol of arity n , for some chosen $n \geq 0$.

As indicated above, there are various standard type constants and operators. Their meaning is tabulated briefly below:-

$\underline{\text{dot}}$	the trivial single element domain.
tr	the standard domain of truth values
$(-)_\perp$	the domain "lifting" operator
$(- + -)$	the coalesced sum operator
$(- \# -)$	the (full) Cartesian product operator
$(- \rightarrow -)$	the (Scott continuous) function space operator

The detailed meaning of these and other domain operators to be introduced is more fully discussed in section 2.2. However, the informal readings suggested by the above will suffice for the present discussion.

Type-variables, in some sense, range over a class of (small) cpos (or domains). Type expressions containing occurrences of type variables are known as polytypes (or generic types); these may be further instantiated by other type expressions. A type expression which contains no occurrences of a type-variable is known as a monotype. Notationally, type-variables are named by (possibly indexed) lower case Greek letters, (with the exception of λ).

The standard meaning of each monotype will be some appropriate domain. However, each polytype stands for a family of domains

indexed on its type-variables. Hence, a polytype is, in essence, a (large) total function acting on the entire class (qua category) of domains; for each possible instantiation by domains of the type-variables of a polytype, there is a corresponding domain.

1.2.4 Well formedness of terms.

As mentioned in section 1.2.2, for each term to be regarded as meaningful, it must "possess" a well-formed type; that is, each meaningful term can be assigned a type to which its value could belong. So the naive intuition is that a type stands for some collection of values, containing the values of all well-formed terms possessing that type. However, this statement is not sufficiently precise since both terms and types may contain variables of one kind or another.

We shall be content with an informal description of the well formedness relation, by reference to the structure of terms. Let tm , $tm1$, $tm2$ range over terms and ty , $ty1$, $ty2$ range over types. The notation $tm:ty$ means that the term tm possesses the type ty :-

- Variables.

Each variable has a unique type stating the values over which it ranges. In practice, this is used to limit the terms which any given variable may be replaced by.

- Constants.

Each constant is introduced with a specific type, which is, in general, a polytype. Such constants may occur at instances of such types.

- Application.

If $tm1:(ty1 \rightarrow ty2)$ and $tm2:ty1$ then $tm1(tm2):ty1$.

- Abstraction.

If a variable $v:ty1$ and $tm:ty2$ then $(\lambda v. tm):(ty1 \rightarrow ty2)$

A term is said to be well-typed if it possesses some type. Also a term is said to be polymorphic if it possesses a polytype; otherwise it is said to be monomorphic. Intuitively, polymorphic terms stand for a family of (monomorphic) terms, one for each possible instantiation of types. For example, the lambda expression:-

$$\lambda x:\alpha. (x:\alpha)$$

is a polymorphic term of type $\alpha \rightarrow \alpha$, and stands for the family of identity functions, one for each possible domain ranged over by type variable α . Hence, if $\mathbb{N}$ stands for the (flat) domain of natural numbers, the above expression can be type instantiated on α for $\mathbb{N}$ to yield $\lambda(x:\mathbb{N}). (x:\mathbb{N})$, the identity function on $\mathbb{N}$.

In general it is possible to efficiently (most generally) infer the type of an expression, given the general types of all constants used, (and occasionally, some of its variables). The LCF system itself uses an algorithm, originally due to Robin Milner (see [Milner78]), for determining this "most general" type information.

So from now on, a policy of not quoting types is adopted for whenever they could be inferred from the context of use. In general, if no type-quotations (or constraints) are explicitly stated then any typing which renders terms well-typed may be taken.

1.2.5 Deduction rules for PPLAMBDA.

The structure of the linguistic component of PPLAMBDA has been presented above. In this section, the intended meaning of the more logical component of PPLAMBDA is given informally.

A major distinguishing characteristic of PPLAMBDA from, say, first order logic or even the traditional λ calculus is that each domain is a (complete) partially ordered set. Intuitively, this ordering represents "is less defined than" or "contains less information than". This is the traditional interpretation of the Scott ordering upon values, as stated in [Scott70]. Further discussion of this topic is given in Section 2.1.9.

In terms of PPLAMBDA, the basic properties of this ordering are as follows. Firstly, $\sqsubseteq$ is certainly reflexive, anti symmetric and

transitive, which can be stated in PPLAMBDA as follows:-

$$\begin{array}{l} \vdash \forall x:\alpha. \quad x \in x \\ \vdash \forall x:\alpha \ y:\alpha. \quad (x \in y) \ \& \ (y \in x) \supset (x = y) \\ \vdash \forall x:\alpha \ y:\alpha \ z:\alpha. \ (x \in y) \ \& \ (y \in z) \supset (x \in z) \end{array}$$

The turnstyle symbol " $\vdash$ " takes its conventional reading from Mathematical Logic and may be prefixed to any provably valid formula. Also, the symbol " $\vdash$ " may be used to indicate an arbitrary valid formula. Later on, we shall also use this symbol as a sequent separator where hypotheses are recorded on the left of the symbol. Note that we have already abbreviated quantified formulae, of the form:- $\forall x. \forall y. fm$ by formulae of the form:- $\forall x \ y. fm$.

So, returning to properties of the partial ordering, note that the following "analysis" rule is valid:-

$$\vdash \forall x \ y. (x = y) \supset (x \in y) \ \& \ (y \in x)$$

It is further assumed that all objects having function type are monotonic with respect to the ordering:-

$$\vdash \forall x \ y. (x \in y) \supset \forall f:(\alpha \rightarrow \beta). f(x) \in f(y)$$

Note that for this formula to be well-typed, the variables x and y must both possess polymorphic type α .

Turning now to the equality relation, $=$, we observe that the given properties of $\in$ already ensure that $=$ is an equivalence relation and that $=$ is extensional with respect to all the functions of interest, by monotonicity:-

$$\vdash x = y \supset \forall f. f(x) = f(y)$$

Further observe that this is a "substitution" rule for functional terms and, in turn, provides the logical basis for the general schematic rule for substitution of equalities into formulae. Any use of substitution could, in principle, be justified by induction on the length of proofs and the above property. LCF provides the general substitution scheme directly, it's use being justified once and for all within the soundness proof for the logic as a whole. The validity of substitution is an example of a meta-theoretic

result about the logic PPLAMBDA. The substitution scheme is displayed by the following:—

$$\text{Substitution} \quad \frac{\vdash \text{tml} = \text{tm2}, \quad \vdash \text{fm}[\text{tml}/x]}{\vdash \text{fm}[\text{tm2}/x]}$$

(where x is a variable with the same type as tml (or tm2))

This notation for inference rules is read in the conventional way; if each formula (of the appropriate syntactic shape) above the line is known to be true, then so also is the formula below the line.

The (meta) notation $\text{fm}[\text{tm2}/\text{tml}]$ stands for any formula that is obtained by replacing every (free) occurrence of term tml by term tm2 in the formula fm where any bound variables are renamed as necessary to prevent "capture of variables". The notation is also used in a similar manner for types; that is, $\text{fm}[\text{ty2}/\text{ty1}]$ is the result of substituting every occurrence of type ty1 by type ty2 , taking care to avoid "capture of type variables". This notation is extended to multiple simultaneous substitution by writing $\text{fm}[\text{u1}/\text{v1}, \text{u2}/\text{v2}, \dots, \text{un}/\text{vn}]$ where the u_i and v_i are terms (or types).

There is a still more general notation in which only certain occurrences of the terms (or types) being substituted for may be (simultaneously) replaced (see [LCF], Appendix 5, p114 and Appendix 7, p130).

Various formal algebraic properties of substitutions as objects in their own right may be found in Section 5 of [Robinson65] and in a more abstract setting in [Huet80] and [HuetOppen80].

The next few PPLAMBDA formulae formulate a number of basic properties of the functions of interest. The following two formulae capture what it means to say that two functions are equal.

$$\begin{aligned} \vdash f \sqsubseteq g &\supset \forall x. f(x) \sqsubseteq g(x) \\ \vdash (\forall x. f(x) \sqsubseteq g(x)) &\supset f \sqsubseteq g. \end{aligned}$$

In PPLAMBDA, functions are explicitly assumed to be extensional; that is, characterised by their input-output relationship. Finally, the usual conversion principles for the λ -calculus are valid. There is, firstly, the alpha-conversion schema:—

$$\vdash (\lambda v_1. tm) \equiv (\lambda v_2. tm[v_2/v_1])$$

where v_1 and v_2 are any two symbolic variables of identical type. and provided that v_2 is not free in tm . Secondly, there is the beta-conversion schema:-

$$\vdash (\lambda v. tml)(tm2) \equiv tml[tm2/v]$$

where the term $tm2$ has identical type to the variable v .

The basic logical connectives and universal quantification are now considered. For the most part, their inference rules are mainly straightforward, and are stated in Figure 1.1 below.

Basic logical properties of connectives

Conjunction

And	$\vdash fml, \quad \vdash fm2$		
introduction	$\vdash fml \ \& \ fm2$		
And	$\vdash fml \ \& \ fm2$	And	$\vdash fml \ \& \ fm2$
elimination1	$\vdash fml$	elimination2	$\vdash fm2$

Implication

Modus	$\vdash fml \supset fm2, \quad \vdash fml$
Ponens	$\vdash fm2$

FIGURE 1.1

The remaining rules presented possess a number of subtleties. The first such rule is known as "specialisation" and is stated as follows:-

Specialisation	$\vdash \forall v. fm$
	$\vdash fm[tm/v]$

where v is some variable and tm is a term of identical type to v . Note that this rule is still valid even if the variable v does not freely occur in the form fm .

The remaining rules are dependent upon the use of sequents, in the natural deduction style (see, for example, [Prawitz65]). A sequent expression has the form:-

$$H \vdash fm$$

where H is some (possibly empty) set of formulas, which can be regarded as hypotheses. The idea is that the consequent (or the formula to the right of the turnstyle) is known to be valid (i.e. provable) on the assumption that each hypothesis to the left of the turnstyle is valid. In short, sequents of the form $\{fm_1, fm_2, \dots, fm_n\} \vdash fm$ are considered valid precisely when the formula $(fm_1 \ \& \ fm_2 \ \& \ \dots \ \& \ fm_n) \supset fm$ is considered valid.

There is a rule for (simultaneous) term instantiation (related to the Specialisation rule given previously) and is as follows:-

$$\begin{array}{c} \text{Term} \\ \text{Instantiation} \end{array} \quad \frac{H \vdash fm}{H \vdash fm[tm_1/v_1, tm_2/v_2, \dots, tm_n/v_n]}$$

where the variables v_i do not occur free in any hypothesis in the hypothesis set, H , and the terms tm_i have identical type to v_i . There is also an instantiation rule for types, which is as follows:-

$$\begin{array}{c} \text{Type} \\ \text{Instantiation} \end{array} \quad \frac{H \vdash fm}{H \vdash fm[ty_1/vty_1, ty_2/vty_2, \dots, ty_n/vty_n]}$$

where the type-variables vty_i do not occur free in any hypothesis in the hypothesis set, H .

The next rule has similar conditions to the above that concern free variables and is used to introduce universal quantification:-

$$\begin{array}{c} \text{Generalisation} \end{array} \quad \frac{H \vdash fm}{H \vdash \forall v. fm}$$

provided that the variable v does not occur free in any hypothesis in the hypothesis set, H . Of course, if there are no hypotheses then this condition is vacuous. The outermost quantifiers on formulae are often omitted, leaving some variables free. Since

universal generalization on free variables is valid, these quantifiers can be re-introduced so long as the variables do not appear free in any hypothesis in the sequent.

Finally, there are the book-keeping rules for sequents - the Discharge and Assumption rules:-

$$\text{Discharge} \quad \frac{(H \cup \{fm_1\}) \vdash fm_2}{H_1 \vdash fm_1 \supset fm_2}$$

where $H_1 \subseteq H$ does not contain any formulae alpha-convertible to fm_1

$$\text{Assumption} \quad (fm) \vdash fm$$

Previously stated rules that did not mention sequents are extended as illustrated for the "And introduction" rule:-

$$\frac{H_1 \vdash fm_1, \quad H_2 \vdash fm_2}{(H_1 \cup H_2) \vdash (fm_1 \ \& \ fm_2)}$$

The sequent formulas simply keep track of all the "undischarged" assumptions that were used to establish the conclusion.

1.3 The Meta Language: ML

ML is a general purpose declarative programming language based on the typed λ -calculus. The principal design goals were to provide an adequate meta-language for conducting proofs in the logical object language PPLAMBDA. A major requirement is to ensure the integrity of inference (i.e the soundness of deductions made). At the same time, the meta-language must also be powerful enough to express various natural modes of reasoning and deduction strategy. Finally, powerful built-in abstraction facilities ought to be provided for the encapsulation of, for example, proofs and proof strategies.

We first briefly look at how the above requirement for a logical meta-language are satisfied within ML. The central idea of machine proof in LCF lies with the performance of proofs; to behaviourally simulate the act of proof and hence generating valid formulae as a result of this process. This gives a direct behavioural counterpart of the logician's basic concept of proof as

the successive application of inference rules to known results, generating further validities.

This has two immediate corollaries; each phrase in the sub-languages of PPLAMBDA should be a representable, manipulable value in ML. Secondly, inference rules are procedurally represented in ML; that is, each primitive proof rule corresponds to a ML procedure. Such procedures transform input data (of the correct shape) into a corresponding output (sequent) formula, also of a specified shape. Each value produced from a simulated proof rule therefore represents a provable (sequent) formula, on the assumption that the input data is correct.

However, the ML procedures could only generally represent sound proof rules when arguments are valid formulae of the correct shape or form. This raises another two significant requirements:-

- To distinguish between provable formulae constructed using inference rules and arbitrarily constructed formulae.
- To ensure that inference rules cope gracefully when applied to input data of incorrect shape.

The first requirement is satisfied by introducing an ML data type, called `:thm`, whose values can only be generated by the application of inference rules to "good" input. The values of this type precisely represent the provable formulae, relative to the present axioms, constants and type operators in force. For this solution to be effective, ML requires a notion of "trademarking" or strong typing to prevent the erroneous generation of objects of each type.

The second requirement is satisfied by including a notion of "exception value" in ML. In a sense, this provides a controlled form of partiality for ML functions. Allowing inference rules to raise recognisable exceptional (or improper) values when inappropriate input is used means that the need to provide ad-hoc "default" output, of the appropriate type, is avoided. ML's

exception mechanism is briefly discussed later.

The need for the manipulation of higher type objects comes from the requirement that ML be capable of expressing natural forms of reasoning. Since, for example, inference rules are already ML functions, it is natural to use functionals which could take inference rules as arguments and return them as results. It turns out that to express adequate tools for goal-directed deduction within ML, it is necessary to be able to manipulate functions representing proofs and to even form lists of such functions.

The design goals and a discussion of the evolution of ML is given in [Milner83] and also in [Gordon83].

1.3.1 An Informal Introduction to ML

The main starting points for the design of the programming language ML included Landin's ISWIM language [Landin66] and Reynolds's GEDANKEN language [Reynolds70]; major additions are the (definition-time) polymorphic type discipline, an elegant and general method of coping with exceptions, provision for functions of higher types and the inclusion of user-defined abstract data types.

ML is informally described here by example; a definitive account of ML may be found in Chapter 1 of [LCF]. The use of ML to "represent" the logic PPLAMBDA is discussed below in following Sections.

ML is a declarative language; ML texts consist of expressions and definitions of functions, data types and so on. Simple (non-recursive) function definitions start with "let", as below:-

```
let p(x,y) = 2*x + y
;;
```

The end of a completed expression or definition is signalled by a double semi-colon. The same definitional form can also introduce basic constants, such as:-

```

let a = 24
and b27 = [3; 4; 5]
and fred = 'This is a token'
and Mary = true
;;

```

Note that the "and" is used to construct simultaneous bindings from simple ones. ML is a strongly typed language, and so each of the above must possess valid ML types in order to be well formed. This type information is inferred from the syntactic structure of the appropriate definitions using the type checking and inference algorithm described in [Milner78]. For instance, the example ML function p defined above has the ML type $:(\text{int} \rightarrow \text{int}) \rightarrow \text{int}$. Also, the constant a has ML type $:\text{int}$, b27 has ML type $:(\text{int})\text{list}$, fred has ML type $:\text{token}$, and finally Mary has ML type $:\text{bool}$. ML types are also known as meta-types.

ML can define objects of higher type. For instance, a simple example is the second order functional twice which takes an ML function, f, and iterates it twice on a given argument:-

```

let twice(f,d) = f(f(d))
;;

```

The ML typing inferred for twice is $:(\alpha \rightarrow \alpha) \times \alpha \rightarrow \alpha$ where α stands for a generic type or type variable. Therefore, twice is an example of a higher-type function which is also polymorphic. ML types can be stated as constraints within definitions if required. For example, a type instance of the above could be defined as:-

```

let booltwice(f,d) = f(f(d:bool))
;;

```

The inferred ML type of booltwice is $:(\text{bool} \rightarrow \text{bool}) \times \text{bool} \rightarrow \text{bool}$. A curried version of booltwice can also be defined by:-

```

let booltwice2 (f) (d:bool) = f(f(d))
;;

```

The inferred ML type of booltwice2 is $:(\text{bool} \rightarrow \text{bool}) \rightarrow \text{bool} \rightarrow \text{bool}$. This could also have been defined in terms of functional composition:-

```
let booltwice2 (f:bool → bool) = (f o f)
;;
```

By using letrec instead of let, functions may be given recursive definitions. Consider the factorial function, fact, defined by:-

```
letrec fact(n) = if (n < 1) then 1 else n*fact(n-1)
;;
```

The ML type for fact is inferred to be `:int → int`. The notation $(b \Rightarrow e_1 \mid e_2)$ is also used for $(\text{if } b \text{ then } e_1 \text{ else } e_2)$ where the condition, b , is an expression with ML type `:bool` and the expressions e_1 and e_2 have identical ML type. Local definitions are introduced using the let notation again:-

```
let x = 27 in x*(x + 1)/2
;;
```

As another simple example, reconsider the factorial function, fact, which could have been defined by:-

```
letrec fact(n) =
  if (n < 1) then 1 else (let m = fact (n-1) in n*m)
;;
```

A convenient alternative to the let notation is the where notation:-

```
letrec fact(n) =
  if (n < 1) then 1 else (n*m where m = fact(n-1))
;;
```

Local declarations abbreviate expressions and so improve readability and can be used to save duplicated effort.

ML has a small range of basic data types including `:int` (the integers), `:bool` (the truth values), `:token` (simple character strings) and the polymorphic type operator of lists, `:( $\alpha$ )list`. A sample of the standard constants available in ML are given in Figure 1.2 below. The `cons` function which adds an element to the front of a list is denoted by `(--)` and has ML type `: $\alpha$  x ( $\alpha$ )list → ( $\alpha$ )list`. Various list processing functionals, such as map and itlist mentioned above are introduced in Appendix 6, p123, of [LCP]. Explicit list expressions are denoted by, for $n \geq 0$,

```

(- + -)          : int # int → int
(- * -)          : int # int → int
(- or -)         : bool # bool → bool
nil              : (α)list
null(-)          : (α)list → bool
tl(-)            : (α)list → (α)list
(- @ -)          : (α)list # (α)list → (α)list
map (-) (-)      : (α → β) → (α)list → (β)list
itlist (-) (-) (-) : (α → β → β) → (α)list → β → β

```

FIGURE 1.2

$[e_1; e_2; \dots; e_n]$ for expressions e_i possessing identical ML types. For example, the sample list expression $[1; 2; 3]$ has ML type $:(\text{int})\text{list}$. Note that lists in ML are homogeneous; all list elements have identical ML type. The (polymorphic) empty list, nil , may also be denoted by $[]$. Functions can be recursively defined over lists in the usual way. For example, the pre-declared ML list functionals map and itlist could have been defined by:-

```

letrec map fn l =
  if (null l) then nil else fn(hd l) . (map fn (tl l))
;;

letrec itlist fn l val =
  if (null l) then val else fn (hd l) (itlist fn (tl l) val)
;;

```

ML admits the standard mathematical notation for ordered pairs (and tuples) of values. For example, the function QR, which produces the quotient and remainder of integer n when divided by integer m , is defined by:-

```

let QR(n,m) = quorem(o,n)

  whererec quorem(q,r) =
    if r < m then (q,r) else quorem(q+1,r-m)
  ;;

```

The ML type of QR is $:(\text{int} \# \text{int}) \rightarrow (\text{int} \# \text{int})$. Note that this gives the correct result when $n \geq 0$ and $m > 0$. If $(m \leq 0)$ then the result is completely undefined; $\text{QR}(2, -3)$ would, in principle, invoke indefinite computation, i.e. a singularity. There is an exception-handling mechanism which can be used to cope gracefully with such situations.

A (labelled) exception-value can be raised by using the ML

operator failwith of ML type $\text{token} \rightarrow \alpha$. There is a means for trapping any such value, using an ML expression of the form $(e_1 ? e_2)$ whose value is e_1 normally or the value of e_2 whenever e_1 is an exception-value; both e_1 and e_2 must have the same ML type. To raise an exception, a token must be supplied to the ML operator failwith; this token then acts as an "error message" which, if it reached the outermost scope, would be displayed accordingly.

So, returning to the previous example, the function QR can be defined to check its arguments and return error messages as appropriate, before evaluating quorem:-

```
let QR(n,m) =
  if (n < 0) then failwith 'Negative first argument' else
  if (m < 1) then failwith 'Non-positive second argument'
  else quorem(n,m)

  whererec quorem(q,r) =
    if r < m then (q,r) else quorem(q+1,r-m)
;;
```

Now evaluating the expression QR(2,-3) would cause the error message 'Bad second argument' to be displayed (and no "proper" output is delivered).

There is also the so-called varstruct notation which provides a basic "pattern-matching", or structured binding, mechanism. An example is given within the following function definition:-

```
let QplusR(n,m) =
  (q + r) where (q, r) = QR(n,m) ? (0,0)
;;
```

The construct $"(q, r)"$ is a simple example of a varstruct; the variables q and r are bound to the corresponding components of the pair on the right hand side. (Note the use of "?" to nominate a "default" value.) The notation for varstructs is extended to simple pattern matching on lists. As an example of this, consider the rather unwieldy ML expression below:-

```
let (x, a. b. (), [u;v]) = (true, [2;3;4;5], ['a';'b'])
;;
```

This results in x being bound to true, a to 2, b to 3, u to the token 'a' and v to the token 'b'. The symbol $()$ above

matches anything. If data has the wrong form, then an exception is raised, with token 'varstruct'. Varstructs may be used in any place where variables could be bound, including lambda expressions. Simultaneous ML bindings of the form:-

let $v_1 = e_1$ and $v_2 = e_2$ and ... and $v_n = e_n$ in e

are equivalent to a single ML binding using a varstruct of the form:-

let $(v_1, v_2, \dots, v_n) = (e_1, e_2, \dots, e_n)$ in e

More generally, mutually recursive functions can be defined as follows:-

```
letrec square(x) = if x < 2 then x else times(x,x)
and    times(x,y) =
            if x < 1 then x                               else
            if x = y then y + times(x-1,y)                 else
            if x < y then square(x) + times(x,y-x)
            else square(y) + times(y,x-y)
;;
```

ML has a basic notation for defining new abstract data types; an abstract type is specified by giving (possibly recursive) type equations for them. This notation is illustrated in Figure 1.3 below by showing how the built-in data type of polymorphic lists might have been defined in ML itself.

```
absrectype ( $\alpha$ )List = . + ( $\alpha$  # ( $\alpha$ )List))
with
    empty = (absList o inl)()
and
    cons(a,l) = (absList o inr)((a,l)
and
    isempty(l) = (isl o repList)(l)
and
    head(l) = (fst o outr o repList)(l)
and
    tail(l) = (snd o outr o repList)(l)
;;

letrec concat(l1, l2) =
    if isempty(l1) then l2
    else cons(head(l1), concat(tail(l1), l2))
;;
```

FIGURE 1.3

The first part of the definition states the form of the isomorphism relationship between the abstract type to the left and the representation type to the right. This relationship is represented by a pair of (implicitly defined) functions which, in this example, take the form:-

```
repList : (α)List → (. + (α # (α)List))
absList : (. + (α # (α)List)) → (α)List
```

The scope of these two functions is restricted to lie precisely inside the with clause. It is within this clause that each of the primitive data operations can be defined, and then "exported" to the outside. These constants and functions are then the only means of manipulating values from the type concerned. In this way, data integrity can be ensured and unnecessary representation detail hidden from view.

A further example of abstract data types is given in Figure 1.4 below; the data type of "lazy" lists (or Streams). The basic idea is to use function closures to delay the evaluation of stream node components until their selection. An example of how "infinite" streams can be defined is given by constructing the infinite ascending sequence of integer squares.

```
absrectype (α)Stream = . + (( . → α) # ( . → (α)Stream))
with
  lazyempty = (absStream o inl)()
and
  lazycons(fa, fs) = (absStream o inr)(fa, fs)
and
  islazyempty(s) = (isl o repStream) (s)
and
  first(s) = ((fst o outr o repStream) s) ()
and
  next(s) = ((snd o outr o repStream) s) ()
;;

let squares = SqrFrom(0)

  whererec SqrFrom(i) =
    lazycons((λx. i*i), (λx. SqrFrom(i+1)))
  ;;
```

FIGURE 1.4

1.3.2 PPLAMBDA in ML.

The discussion above has focused on ML from the programming language point of view. It is now appropriate to introduce those aspects of ML which deal directly with the LCF object language.

First of all, there are various ML abstract data types for representing syntactic components of PPLAMBDA. The semantics of PPLAMBDA is then mechanically represented by ML functions standing for the various inference rules of the logic. A number of pragmatic tools are provided in ML which assist in the use of PPLAMBDA - quotation and anti-quotation, for example - which are also described. Finally, the LCF theory structuring mechanism is discussed.

1.3.3 Data types for PPLAMBDA syntax.

There are four ML data types which relate directly to PPLAMBDA. They are as follows:-

- form - ML data values representing each well-formed PPLAMBDA form expressions.
- term - ML data values representing each well-typed PPLAMBDA term expressions.
- type - ML data values representing each well-formed PPLAMBDA type expressions.
- thm - ML data values which represent, in general, each provable sequent formula, relative to the prevailing environment of PPLAMBDA constants, axioms and types.

The first three types provide for the (abstract) syntax of PPLAMBDA and their values may be constructed and analysed freely. For each of these types, there are various constructor functions, corresponding to each syntactic alternative, as well as discrimination and (partial) selection functions.

For example, the logical conjunction of two PPLAMBDA formulae

represented by an ML function mkconj :form # form → form which takes pairs of forms into a form that syntactically represents their conjunction. Such conjunctions can then be syntactically analysed into their original constituents by using the ML function destconj :form → form # form (which fails with an appropriate error message if the input form is not a conjunction). Finally, there is a ML function isconj :form → bool which recognises conjunctive formulas; this gives true when the input is a conjunction and false otherwise. This pattern of constructor, analyser and recogniser function is repeated for each syntactic alternative in each sub-language.

The final ML type introduced above, :thm contains ML values standing for provable sequents (with respect to the present theory). Such values can only be generated by the application of certain given ML functions. So, if each of these ML functions correctly implements a PPLAMBDA inference rule, any value so generated must correspond to a provable sequent; this strongly depends upon the security of the type checking system.

Although the construction of :thm values is carefully controlled, the analysis of already existing sequents into its principle constituents of hypotheses and conclusion is freely permitted; an ML function destthm :thm → form list # form is provided for this purpose.

1.3.4 Representing the Semantics of PPLAMBDA in ML.

The syntax of PPLAMBDA is represented within ML as values from the ML types :form, :term and :type. We show here how the semantics of PPLAMBDA is represented in ML for the purpose of performing proofs. As indicated above, this is done by using a predeclared collection of ML functions each of which simulates a PPLAMBDA inference rule.

For example, consider the inference rule for And introduction, as described in Section 1.2.5 above. Corresponding to this rule, a ML function CONJ :thm # thm → thm is given, which takes as input a pair of proven sequents and returning the valid sequent, of ML type thm, corresponding to their conjunction. Some rules require other

kinds of parameters. For example, consider the universal generalisation scheme also given in Section 1.2.5. This rule is represented in ML by the function GEN :term $\rightarrow$ thm $\rightarrow$ thm, whose first argument gives the variable, v , being bound and the second argument gives the sequent being generalised. This rule raises an exception if the term v does not represent a PPLAMBDA variable, or if it occurs freely within some hypothesis from the sequent.

Also, there are axiom schemas which depend upon terms and forms; for example, there is ASSUME :form $\rightarrow$ thm, which simulates the axiom schema for Assumption described in Section 1.2.5, and the function REFL :term $\rightarrow$ thm, which maps any input term, tm , to the tautological sequent $\vdash tm = tm$.

The remaining 46 PPLAMBDA inference rules are described in Appendix 5 of [LCF].

1.3.5 Auxiliary ML functions for PPLAMBDA.

Besides the main ML functions for doing PPLAMBDA syntax processing and performing semantic inference, there are others which do various ancillary tasks that are nonetheless necessary. These functions are documented in Appendix 7 of [LCF].

For example, there are functions for determining useful quantities such as typeof :term $\rightarrow$ type, which takes object language terms and determines their object language type. Other functions of this kind are formfrees :form $\rightarrow$ term list, which calculates a list of all free variables in the given input form or aconvterm :term # term $\rightarrow$ bool, which determines if the given pair of terms are alpha-convertible to each other.

There are also functions for doing term substitution in either forms or terms. For instance substinterm :(term # term)list $\rightarrow$ term $\rightarrow$ term, performs a simultaneous substitution into the given term where the list of term pairs represent right-to-left term replacements. Only free occurrences of right-hand terms are replaced in the given term by the corresponding left-hand term. Also, variants of bound variables are chosen as necessary to avoid capture of bound variables. This property of the basic substitution mechanism makes explicit alpha-conversions

unnecessary. An ML function called variant :term # term list → term, is provided for constructing new variables in a systematic way (i.e. the given variable's name is primed or unprimed so that it does not appear amongst the given list of variables).

There are also functions for substituting for particular free occurrences of terms, and instantiating type variables in forms, terms and types.

Finally, some simple functions are provided for simple one-way pattern matching of terms and forms. For example, the ML function termmatch:term → term → (term # term)list, treats the first argument as a pattern and the second as a term to be matched against. The result is a simultaneous substitution for free variables in the pattern which, when applied, delivers the matched term. If no such substitution exists then failure results. There is a corresponding pattern matching function for forms called formmatch :form → form → (term # term)list.

1.3.6 Quotation and Anti-quotation.

Up till now, the two parts of LCF, the object language and the meta language, have not been presented together. The quotation and anti-quotation mechanisms provided in ML permit them to be judiciously mixed. The general concept of "quotation" was introduced by Quine (see [Quine81] page 23) and his "quasi-quotation" (see [Quine81] page 33) corresponds to "anti-quotation" in LCF.

Essentially, the use of quotation permits the statement of object language phrases within ML as they might be written down in everyday use. For example, consider the object language form*:-

$$\forall z:\alpha. (f:\alpha \rightarrow \alpha \equiv g) \ \& \ g(z) \in y \ \supset \ (\lambda x:\alpha. y) \equiv (\lambda z:\alpha. f(z))$$

This could be rendered in ML by enclosing it in double quotation marks, and using the machine-processable concrete syntax for PPLAMBDA (see Appendix 2):-

* This formula is well-formed, but not logically valid, since the functions f and g could, in general, be non-constant functions.

```
"! z:*. (f:* -> * == g) & g(z) << y IMP
      (\x:*. y) == (\z:*. f(z))"
```

to produce an object of ML type :form. Of course, this value could also be generated by using the given "abstract syntax" construction functions; however, this is much less readable! The quotation mechanism is also available for terms and types. For instance, " $\lambda x:*. z \Rightarrow f(x) \mid g(x):**$ " has ML type :term and stands for the object language term $\lambda x:\alpha . z \Rightarrow f(x) \mid g(x):\beta$ with object language type " $:\alpha \rightarrow \beta$ ". As can be seen, object language types may also be quoted by prefixing the enclosed type expression with a colon symbol as in the following example:-

```
":(* -> tr) # (** -> **) -> tr"
```

To avoid using the machine-processable concrete syntax, as much as possible, the given PPLAMBDA typography will be used in quotations.

From a technical point of view, the quotation facility is little more than engaging a special-purpose parser to translate object language statements into their corresponding phrase structure. What gives this mechanism much greater flexibility is the anti-quotation facility which permits ML phrases (of appropriate ML type) to be embedded into quotations. These are then evaluated to produce a standard object-language quotation value; the mechanism is illustrated as follows. Suppose that the ML identifier tm is bound by using:-

```
let tm = "\ f:( $\alpha \rightarrow \alpha$ ). f(f(x))"
;;
```

Hence, tm has ML type :term. Note that the value that tm is bound to is an object language term whose object language type is " $:(\alpha \rightarrow \alpha) \rightarrow \alpha$ ". Now, the value of tm can now be used within quotations by prefixing tm by the symbol ! as follows:-

```
" $\forall x:\alpha. !tm(F) \equiv F(F(x))$ "
```

which, when evaluated, gives:-

```
" $\forall x:\alpha. (\lambda f:(\alpha \rightarrow \alpha). f(f(x)))(F) \equiv F(F(x))$ "
```

For this to evaluate without failure, the ML identifier tm must

have ML type :term. Also, notice how term substitution and anti-quotation behave differently since the free variable "x" occurring in tm becomes bound when the value of tm is inserted into the formula.

1.3.7 LCF Display Conventions.

When PPLAMBDA values are produced as the results of some computation, these will be displayed in the form of an appropriate quotation by the LCF system.

Of course, no provision is made by the LCF system for the quotation of :thm values by the user. However, it is convenient for :thm values to be displayed in a similar way to other PPLAMBDA values. Since the display of hypotheses of theorems can generate overwhelming quantities of output, there is an abbreviation convention for this; for each hypothesis in the sequent, output a single dot, all followed by the LCF turnstile ']-', and then followed by the quotation of the conclusion. So, the PPLAMBDA tautology:-

$$((x = y), (y = z)) \vdash (z = x),$$

would be displayed by LCF as:-

..]- "z = x"

If necessary, the hypotheses can be extracted by applying the ML function hyp :thm → form list

The following additional convention is adopted for the purpose of the presentation of the case studies given later; if a PPLAMBDA sequent is displayed using the LCF sequent symbol "]-", then this indicates that the stated sequent has been machine-generated or processed, during the course of the research reported herein. PPLAMBDA sequents displayed using the usual sequent symbol, "⊢", may not have been so generated.

1.3.8 LCF Theories.

PPLAMBDA is a family of deductive calculi which is parameterised upon the types and constants (and properties) that may be introduced, or declared, by the user. One or more

declarations may be grouped together to form a named LCF theory. These declarations may depend upon previously defined quantities. Therefore, theories are permitted to inherit the knowledge from other, previously established, theories. This inheritance relationship between pertinent LCF theories then takes the form of a directed acyclic graph.

So, more precisely, an LCF theory is made up of two parts; a definitional part and an archive part. The definitional part records all items introduced to LCF in the context of the particular theory, including any dependancies upon previously defined LCF theories. The archive part records :thm values (i.e. theorems) proven using LCF in the context of that particular theory.

Typically, the definitional part contains the names and the arities of new type operators, the names of new constants (with their types) as well as any (labelled) axioms; the archive part consists of a list of (labelled) theorems that have been proven in LCF from axioms in scope.

The Edinburgh LCF system is always used in the context of a particular theory. This theory determines the PPLAMBDA environment of those constants and type operators that may be legitimately used within object language phrases. The hierarchical dependence of theories upon others transitively permits the mention of constants and types defined in ancestor theories. Already known theorems from presently reachable theories may be retrieved for later use.

LCF theories are represented in the machine by a pair of files, one for each part. Each theory is named to permit later theories to inherit it. There are also various ML operations for dealing with the administration of theories from within LCF (i.e for registering new information, and retrieving known data). For further details, consult sections 3.4.1 to 3.4.4 of [LCF], p80-85.

The theory methodology permits some degree of modularity and encourages the "separation of concerns". Combined with the notion of polymorphic type, this provides a powerful technique for structuring and factorising logical specifications and their deductive consequences. However, no provision is made for the

hiding or even the renaming of constants and types so as to change the view of one theory by another.

Recent attempts to incorporate many of the insights from using the specification language CLEAR into LCF are reported in [Sannella82] and in [BurstallSannella83].

1.4 Proof using LCF.

LCF was designed to provide a programmable, general purpose, proof assistant; we now consider how LCF supports the deduction of theorems via the construction and subsequent performance of proofs.

Various tools are provided in LCF which encourage the use of certain approaches to proof construction and performance. There are two principal modes of working; forward derivation and goal-oriented derivation; these are discussed in more detail in Sections 1.4.1 and 1.4.3 respectively.

There is also a separate discussion of the standard LCF simplification mechanism, the LCF simplifier. The use of the simplifier plays a frequently important role in many proofs and an example of a proof technique with aspects of both of the methodologies mentioned above. This is discussed in Section 1.4.2.

1.4.1 Forward deduction.

We have seen how PPLAMBDA is represented in LCF by ML functions for processing syntax and for performing valid deduction. As already mentioned that the only way to construct theorems in LCF is to compose together inference rules and apply them to other already known theorems. This is the basic idea of forward deduction, which underlies the conventional notion of proof as a sequence of inference rule applications, each of which uses the results of previous deductions as input, in order to provide further results. A statement is then said to be provable (from a basic set of premises, i.e. axioms and assumed hypotheses) if there exists a proof with the statement as its final result (see page 5 of [Schoenfield67], for example).

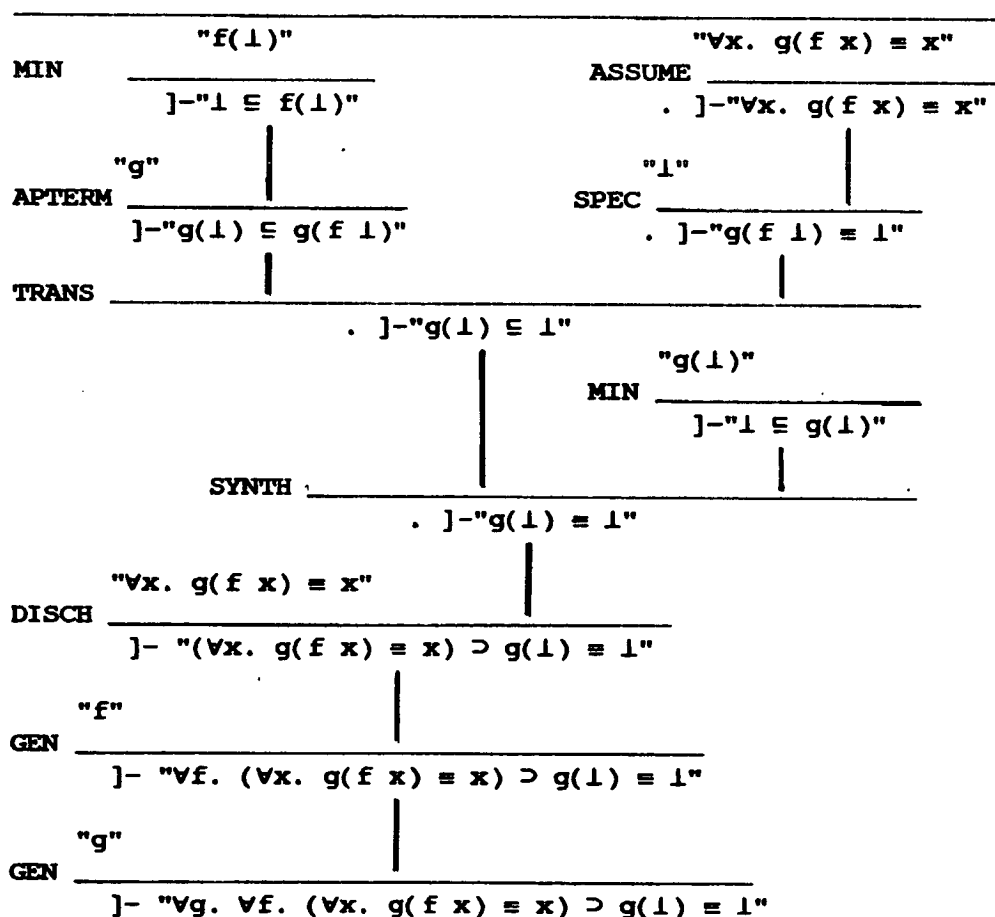
So, in principle at least, all provable theorems of PPLAMBDA (relative to the present axioms) could be enumerated by generating

all possible sequences of inference applications - with any attendant term and form parameters as necessary. But the inherently infinite branching factor of this process renders it unusable for all practical purposes!

However, specific derivations may be examined and evaluated to verify whether or not they behave as expected. Indeed, by encapsulating useful composite derivations as ML functions, larger steps can be built up from smaller steps. Such ML functions are known as derived inference rules, and various examples of these are given later on. A illustrative example of forwards proof using LCP is now given of the simple (polymorphic) tautology:-

]- "Vg f. (V x:α. g(f(x)) = x) ⊃ g(1:β) = 1"

This appears as Lemma 2.1, and states that surjective monotonic functions are strict. Figure 1.5 below gives a proof of this in a



.]- "g(1) ∈ 1"

"Vx. g(f x) = x"

DISCH

]- "(Vx. g(f x) = x) ⊃ g(1) = 1"

"f"

GEN

]- "Vf. (Vx. g(f x) = x) ⊃ g(1) = 1"

"g"

GEN

]- "Vg. Vf. (Vx. g(f x) = x) ⊃ g(1) = 1"

FIGURE 1.5


```

let th1 = (ASSUME "V(x:α). g(f(x):β) = x");;
let th2 = (SPEC "1:α" th1);;
let th3 = (MIN "f(1:α)");;
let th4 = (AFTERM "g" th2);;
let th5 = (TRANS(th4,th2));;
let th6 = (MIN "g(1:β)");;
let th7 = (SYNTH(th5,th6));;
let th8 = (DISCH "Vx. g(fx) = x" th7);;
let th9 = (GEN "f" th8);;
let th10 = (GEN "g" th9);;

```

```

GEN "g"(GEN "f"(DISCH "Vx. g(fx) = x"(SYNTH
(TRANS(AFTERM "g"(MIN "f(1:α)"), SPEC "1:α"(ASSUME
"V(x:α). g(f(x):β) = x")), MIN "g(1:β)"))))
;;

```

FIGURE 1.6

deduction tree style. From this, a sequence of inference rule applications in ML can be very easily obtained. Such a sequence is shown as the first part of Figure 1.6 above. Each of these individual calculations could be invoked together by a single (fairly monstrous) ML expression, as shown in the second part of Figure 1.6.

This is a fairly easily proven theorem and the deduction tree is not too hard to invent in this case. However, at this lowest level of proof using LCF, it is generally found that the essential content of proof are submerged in a welter of extraneous detail. At each point, a suitable choice of parameters usually has to be made.

Clearly this technique embodies the "bottom-up" approach to proof construction. In going from one step to the next, there is no direct correlation with the final result to be achieved. Even with hindsight of the complete proof, many steps appear to be unrelated to their neighbours.

However, all is not entirely lost. The generality of the programming language ML permit more sophisticated derived inference rules to be constructed. They can somewhat raise the level at which proofs can be generated by automating the choice of which rules to apply, when to apply them and with which parameters they are applied. In this way, more and more complex sequences of deduction can be built up to perform more and more complex tasks.

A fairly sophisticated example of a large LCF derived inference rule is the standard LCF simplifier. This is implemented via a suite of ML functions that locate places at which rewriting can happen and then perform the required inferences, within LCF, that justify such rewriting. More will be said later about simplification in Section 1.4.2.

There are also simple and straightforward examples of derived rules which do rather more mundane tasks. Consider, for example, the last three deductions made in our example above. These amount to no more than the routine discharging of remaining assumptions and closing up with respect to universal quantifiers. It is easy to give an ML function called TIDYUP :thm $\rightarrow$ thm that carries out this task. Incidentally, the ML function newfact: token # thm $\rightarrow$ thm, performs this transformation when it stores named facts in the archive of the present theory. This ensures that all theorems are stored as sentences.

```
let TIDYUP(th) =
  let th' = itlist DISCH (hyp th) th in
    itlist GEN ((formfrees o concl) th') th'
;;
```

Another important general facet of doing proof within LCF is that facts can be stored away and retrieved for later use in other deductions. The main point is simply that to use a fact, its proof does not have to be repeated - it is enough that it has already been proven, clearly contributing to economy of proof effort.

As we have seen above, theorems can contain terms which are polymorphic. Such theorems are therefore parameterised, in some sense upon type and may be instantiated to give more and more specific versions. Also, theorems generally have quantified and free variables which therefore represent parameterisation over values.

An application of this is briefly illustrated to demonstrate its versatility and usefulness. In the example of forwards derivation, above, the polymorphic theorem:-

] - " $\forall g \text{ f. } (\forall x. g(fx) = x) \supset g(1) = 1$ "

was constructed. Suppose that this has been archived in a LCF

theory called **KERNEL**, with the (rather odd) name of 'gUU' and can be retrieved later using the ML function **FACT** :token → token → thm.

Now we shall build, in ML, a derived inference rule called **ISO STRICT** :thm → thm which behaves in the following way:-

$$\text{ISO STRICT} \frac{A \vdash \forall z:\alpha. G(F(z):\beta) \equiv z}{A \vdash G(1) \equiv 1}$$

This rule can be expressed in ML as shown in Figure 1.7 below. Essentially, it operates in three stages. Firstly, the given theorem is decomposed into its assumed constituents; if it does not "match" the expected shape, then failure occurs, indicating that the rule **ISO STRICT** has been applied to inappropriate input. On the other hand, if the theorem does "fit" then various data are extracted. This second stage is where the general fact, **gUuthm** :thm, is type instantiated and then specialised to ensure that the antecedent of **gUuthm** matches (i.e. is alpha-convertible

```

let gUuthm = (FACT 'KERNEL' 'gUU');;

let ISO STRICT(th) =

    % Extract data from body of input theorem %

    (let (Z, fml) = (destquant o concl) th in
     let tya = typeof Z
     and tml = (fst o destequiv) fml in
     let tyb = typeof tml
     and (Gtm, tm2) = destcomb tml in
     let Ftm = (fst o destcomb) tm2 in

        % Instantiate and specialise gUuthm %

        let th1 = ( (SPEC Ftm)
                    o (SPEC Gtm)
                    o (INSTTYPE [(":α",tya);(":β",typ)])
                    ) gUuthm
        in

            % Apply theorem th to th1 to get the result %

            (MP th1 th)

    ) ? (failwith 'ISO STRICT')
;;

```

FIGURE 1.7

to) the input theorem. The final step consists of supplying the given input, using MP (the standard inference rule representing Modus Ponens), to the instance just constructed.

Notice that the theorem gUuthm is retrieved from the theory KERNEL precisely once; however it unnecessarily possesses global scope in the above. By using a local binding this untidiness can be avoided as follows:-

```

let ISOSTRICK =
  let gUuthm = (FACT 'KERNEL' 'gUU') in
  let isostrict (th) =

      ( ... as previously for ISOSTRICK ... )

  in
    isostrict
;;

```

Most of the work done by the rule ISOSTRICK is taken up with decomposing the input into certain constituents. It turns out that this process can be systematically automated within ML - an ML functional called METARULE which maps (polymorphic) theorems (of a general form) into derived inference rules can be defined, and is briefly discussed in Section 5.5.

1.4.2 Simplification.

The LCF simplifier is a complex, derived inference rule, providing LCF with much of its basic deductive power. It is frequently used in proving theorems with LCF and can be applied in a very wide range of contexts.

It is discussed as a separate topic here for various reasons. As mentioned previously, it forms a versatile general purpose proof tool of wide applicability and so has independent interest. Secondly, it can be applied either as an inference rule for forwards deduction or in a goal-oriented way; the use of the simplifier does not neatly fall entirely within either approach. Thirdly, the LCF simplifier provides its facilities via a moderately large family of inter-related ML functions. These are briefly described in order to show the variety of ways the simplifier may be applied.

The standard LCF simplifier was designed and implemented by R.Milner and C.Wadsworth in 1978. It consists of about 600 lines of ML text and comes with the basic LCF system.

The primary function of the simplifier is to perform "symbolic evaluation" or term reduction, with respect to a given collection of term rewriting rules. As such, it is the principal concession to conventional automatic theorem proving tools in the basic Edinburgh LCF system.

There are three standard ML functions available for invoking the simplifier, one for each type of object that could be rewritten.

Each basic simplification function firstly depends upon a simpset which is a set of theorems (or simprules) that can be used as term rewriting rules. Its second argument is the object to be rewritten. The reduction process attempts to find occurrence at which term rewriting, using the simprules, can take place. The next phase is to perform the LCF deductions which actually carry out the reductions and which also provide the justification for these replacements. These two phases are successively repeated until no further opportunities for reduction can be found. The resulting object is returned along with any additional overall justification of the reductions performed as proven theorems.

The simplest kind of simprule is the simple equational theorem:-

$$H \vdash \forall \vec{x}. \text{ ltm} = \text{ rtm}$$

where $\vec{x}$ is a (possibly empty) list of variables, and H is a (possibly empty) set of hypotheses. This could be applied as a (left-to-right) term rewrite rule as follows; remove any leading quantifiers, by specialisation, taking care to avoid any free variables occurring in the hypotheses, H. The left-hand-term, ltm, can now be regarded as a pattern to be matched against subterms from the term being rewritten. If a "match" is found, the above simprule is instantiated (for terms and types) and the resulting right-hand term replaces the matched subterm.

For this to be fully justified, the following aspects need to

be discussed. Clearly, the left hand term (or pattern) is used to match both type variables and free term variables that occur within it to parts of the subterm being matched (or matchee). However, because the original simprule must be instantiated, it is necessary to require that either kind of variable used in matching does not occur free within the hypotheses, H , of the simprule; all such term and type variables are known as the instantiable variables of the simprule.

The second point concerns how to perform the general replacement of subterms that match the lhs by their counterpart on the rhs. Generally speaking, if the subterms in question only contain variables that are not lambda-bound in the context then ordinary substitution of rhs for lhs will suffice. However, if this is not the case then the principles of functional abstraction and extensionality must be used to build the left and right hand side terms up until a replacement by substitution is possible. Quantifier bound variables are dealt with by the use of universal generalisation and specialisation. So, it is necessary to traverse the given object (be it a thm, form or term construct) and decompose it into its constituents both to discover match occurrences and to generally provide sufficient material to construct the result after rewriting has taken place.

The most general form of simprule that can be used is:-

$$H \vdash \forall \vec{x}. fm \supset (tm_1 = tm_2)$$

where $\vec{x}$ is a (possibly empty) list of variables, H is a (possibly empty) set of hypotheses, and fm is any form. The previous simple equational form of simprule arises when fm is the formula TRUTH. This more general form of simprule (known as a conditional simprule) requires a more general method of simplification known as conditional simplification.

The idea is quite straightforward. As before, remove all leading quantifiers in the conclusion, taking care to avoid any free variables in the hypotheses, H . Next, proceed to use the left-hand term of the equational consequent as a pattern in the way described above. Having found an appropriate match (of types as

well as terms), instantiate the quantifier stripped *simprule*. The next stage is then to recursively simplify the antecedent formula. If this can be reduced to an easily recognised tautology (i.e. conjunctions of instances of reflexivity, for example) then the original replacement can be justified by an application of Modus Ponens with the appropriate tautology. The remainder of the process for justifying replacement then continues as before.

Although conditional simplification does force recursive use of the simplifier itself, in the majority of cases the antecedent formulas to be established are often rather easy to prove. Moreover, such rules arise naturally in proofs by structural induction (see, for example, Lemma 4.18).

To sum up, conditional *simprules* act in much the same way as equational rewrite rules, except that the permission for its application is granted by proving the antecedent of the rule - also by simplification!

The simplifier is made available via a small collection of ML functions. Some of these manipulate the (special) sets of theorems (known as simpsets) which represent term rewriting rules. The function ssadd : *thm* → *simpset* → *simpset* is used to include new theorems in the given *simpset*; this also checks that the theorem to be added conforms to the above characterisation of *simprules*. There is also a function for making the union of two *simpsets* called ssunion : *simpset* → *simpset* → *simpset*. There are also two basic given *simpsets*; EMPTYSS which contains no *simprules*, and BASICSS which contains various standard rules for the general evaluation of lambda expressions and so on (see Section 2.1.2).

Turning now to those ML functions which apply the simplifier, these all require a *simpset* and an object to simplify relative to it. The result is always the reduced object and, if necessary, additional justification of the reduction performed. The ML function SIMP : *simpset* → *thm* → *thm* applies the simplifier to theorems, producing a theorem as result. This function, although standard, could be thought of as a derived (albeit sophisticated) inference rule.

The function simplterm : *simpset* → *term* → (*term* # *thm*) reduces

terms and returns the reduced term as well as an equational theorem demonstrating that the input term and resulting term are provably equal. So, for simpset ss, we have that:-

$$\text{simp\_term } ss \text{ } tm_1 = (tm_2, H \text{]- } "!tm_1 = !tm_2")$$

where tm_1 simplifies to tm_2 using simprules from the simpset ss. The remaining simplification invoking function is simpform : simpset $\rightarrow$ form $\rightarrow$ (form # (thm $\rightarrow$ thm) # (thm $\rightarrow$ thm)), which as its name suggests, simplifies the subterms of given forms. The result is a triple consisting of the reduced form and a pair of ML functions that demonstrate the validity of the simplification performed. More exactly, for any simpset ss:-

$$\text{simp\_form } ss \text{ } fm_1 = (fm_2, p_1, p_2)$$

where $p_1(\text{]- } fm_1) = H \text{]- } fm_2$ and also, $p_2(\text{]- } fm_2) = H \text{]- } fm_1$. In short the functions p_1 and p_2 show that the formulae fm_1 and fm_2 are logically equivalent, with respect to the simpset ss, by demonstrating that from either one the other can be proven (i.e. they are inter-derivable).

Also, note that all theorems produced as a result of the simplification process may collect some additional hypotheses. In all cases these are the hypotheses of any simprules that are actually engaged to produce the result.

It has been claimed above that the simplifier could be used in a goal-directed, or tactic-like fashion. This application exploits the fact that, in some sense, simplification is symmetric. That is, if a formula fm_1 can be simplified to fm_2 then both formulae are logically equivalent, with respect to some simpset. This further implies that both $\vdash fm_1 \supset fm_2$ and $\vdash fm_2 \supset fm_1$ are valid. The standard function simpform essentially provides proofs of each of these theorems. To use simplification in a goal-directed way, it is the second of these "proof functions", given by simpform that is used to provide the justification for the reduction of a goal to subgoal. A systematic approach to goal oriented reasoning is given in the next Section.

1.4.3 Goal-oriented deduction: Tactics.

A major aspect of proof within LCF is that not only can they be performed by forwards deduction, but also proof sequences can be constructed in a "top-down" fashion. The main idea is to work backwards from what one wants to ultimately achieve to determine more and more refined intermediate subgoals until they directly correspond to already known results.

However, decomposition by itself does not immediately construct the desired theorem; a proof still has to be performed. Since the idea of decomposition of "goals" is to indicate what the proof is, the decomposition process has also to accumulate a corresponding justification; by composing each of these justifications together, a complete justification is obtained. So, once decomposition is itself completed, the justification component can be applied to those facts achieving the final subgoals, yielding a theorem corresponding to the original goal.

This idea is now rendered in more precise terms. Assume that there are classes of objects called goals, and events that may attain goals, via a binary achievement relation. Finally, validations are partial functions which map lists of events into events (i.e functions of type $\text{:event list} \rightarrow \text{event}$).

Now, in the above, there was the implicit notion of "decomposition step" which took a goal and decomposed it into some subgoals and a corresponding justification component. This notion is formalised here by the concept of tactic. Basically, a tactic is a (possibly partial) function which maps goals into a pair consisting of a list of goals and a validation function. Here the list of goals corresponds to the subgoals given by the decomposition and the validation corresponds to the justification component. More formally, the set of all tactics is the set of partial functions $\text{:goal} \rightarrow (\text{goal list} \# (\text{event list} \rightarrow \text{event}))$.

Clearly, not every function of that type behaves in a way that satisfies the informal requirements of a tactic. For example, the validation function could be completely unrelated to the decomposition performed, for each argument.

This leads to the stronger concept of valid tactic. A tactic T

is said to be valid iff whenever, $T(g) = ([g_1; g_2; \dots; g_n], v)$, for some $n \geq 0$ and also for all events $e_1, e_2, \dots, e_n$ such that e_i achieves g_i (for each $1 \leq i \leq n$) then the event e achieves the goal g where $e = v([e_1; e_2; \dots; e_n])$.

Naively, this states that validation functions from valid tactics always produce an event achieving the original goal, given events achieving the subgoals. Hence valid tactics are precisely those tactics whose validation functions match, or justify, the corresponding decomposition of goals into subgoals. Figure 1.8 below describes this relationship diagrammatically.

Further note that the structure of the validation function will usually depend strongly upon the form of the goal to be achieved, and will usually rely upon the form of the subgoals produced as well. A subtle point to note is that if a valid tactic produces subgoals which can be achieved then the validation function is obliged to produce an event when given a list of events achieving these subgoals. A perhaps paradoxical result is that if a tactic never produces any achievable subgoals then it is vacuously valid, whatever the form of the validation component!

This leads onto a yet stronger notion connected with the use of tactics; the concept of strongly valid tactic. A tactic T is said to be strongly valid iff it is valid and for any achievable goal g , for which $T(g)$ is defined, each of the subgoals produced are also achievable.

Informally, a tactic is strongly valid if when applied to an achievable goal to obtain subgoals, these subgoals are also, in turn, achievable. This is clearly a strong requirement to place upon any tactic. In some sense, it says that one can never make a wrong move by applying the tactic. However, even so, this is not

$$\begin{array}{ccc}
 g & \xrightarrow{T} & ([g_1; g_2; \dots; g_n], v) \\
 e & \xleftarrow{v} & [e_1; e_2; \dots; e_n]
 \end{array}$$

if e_i achieves g_i , for $1 \leq i \leq n$, then e achieves g

FIGURE 1.8

quite the same thing as always making a correct move, since strongly valid tactics could simply produce subgoals that are strictly equivalent to the original goal and which are no "easier" to solve. For example, the tactic that roughly speaking makes its argument into a subgoal unchanged is easily seen to be strongly valid!

Before considering how these notions are employed to assist the construction and design of LCF proofs, we introduce some useful functions for manipulating tactics themselves. These functions are known as tacticals by analogy with functionals. The first tactical to be considered is called THEN :tactic # tactic $\rightarrow$ tactic. This provides a composition of tactics that permits larger, complex tactics to be built up from smaller ones. It is written infix between its arguments.

The basic idea of composing a tactic T_1 with a second tactic T_2 is to first apply T_1 to the goal to produce a list of subgoals and corresponding validation. The second tactic T_2 is then applied to each of these subgoals in turn. The result of this is to give a list of subgoal lists with their (individual) validations. So, finally, this list is flattened, giving a simple subgoal list, and the validations are rearranged and composed to produce an overall validation for attaining the original goal. Figure 1.9 below illustrates this behaviour.

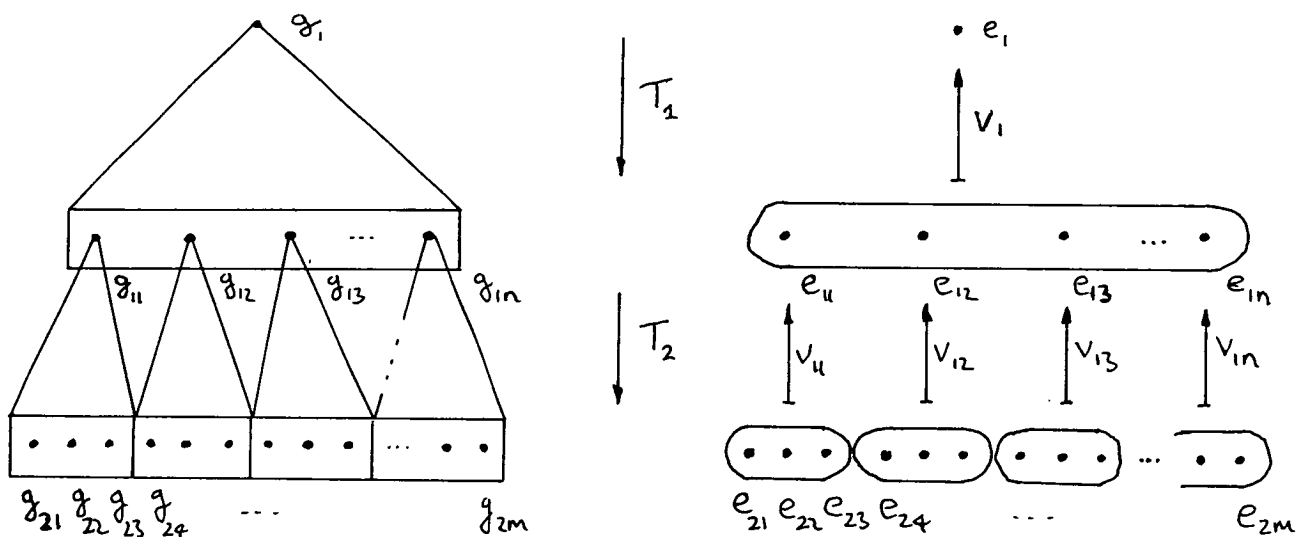


FIGURE 1.9

Figure 1.10 below shows how THEN is defined in the Edinburgh LCF system. The ML functions map, split and flat are standard and can be found in Appendix 6 of [LCF]; the auxiliary function mapshape $:(\text{int list} \# (\alpha \text{ list} \rightarrow \beta) \text{ list}) \rightarrow \alpha \text{ list} \rightarrow \beta \text{ list}$, essentially distributes the input list of validations among the theorem lists in the expected way. To do this, the number of events that each validation requires is determined from the number of corresponding goals produced. This in turn uses another ML function called chop $:\text{int} \# \alpha \text{ list} \rightarrow (\alpha \text{ list} \# \alpha \text{ list})$, which given $n \geq 0$, splits the input list l into the first n elements, and the remainder (with failure if this is not possible).

Note that the THEN tactical is associative, i.e. for any tactics T_1, T_2, T_3 , we have that:-

$$(T_1 \text{ THEN } T_2) \text{ THEN } T_3 = T_1 \text{ THEN } (T_2 \text{ THEN } T_3)$$

as is expected of a composition operation. There is an "identity" tactic called IDTAC :tactic which has the simple one-line ML definition:-

```
let IDTAC(g) = ([g],  $\lambda[e].e$ )
;;
```

As expected, we have that $(\text{IDTAC THEN } T) = T = (T \text{ THEN IDTAC})$, for any tactic T .

```
letrec chop (n,l) =
  if (n = 0) then (nil, l) else
    let (m,l') = chop(n-1,t1 l) in ((hd l) . m , l')
;;

letrec mapshape (nl,fl) l =
  if (null l) then nil else
    let (m,l') = chop(hd nl,l) in
      (hd fl) (m). (mapshape (tl nl, tl fl) l')
;;

let (T1 THEN T2) g =
  let (g1,v) = T1(g) in
  let (g11, v1) = (split o (map T2)) g1 in
    (flat(g11), (v o (mapshape(nl, v1))))

  where nl = (map length g11)
;;
```

FIGURE 1.10

There is a generalised form of the tactical THEN called THENL: tactic # (tactic list) $\rightarrow$ tactic. The effect of the tactic expression (T THENL [$T_1; T_2; \dots; T_n$]) is to apply the first tactic T, and to then apply corresponding tactics to corresponding subgoals. This process fails if the subgoal and tactic lists do not have equal length. THENL can be defined in ML in a similar manner to THEN.

It was mentioned earlier that tactics could be (partial) functions from goals to subgoals and validation function. The tactical ORELSE :tactic x tactic $\rightarrow$ tactic produces a tactic that applies its first tactical argument to the goal, returning the result of that, if successful. However, if failure arises then the second tactical argument is applied to the goal instead. This tactical can be easily defined in ML (using the failure trapping mechanism) as:-

```
let (T1 ORELSE T2) g = T1(g) ? T2(g)
;;
```

The usefulness of the ORELSE operator is that it permits individual tactics to reject their input and register their refusal to conduct further processing by raising an exception. This then permits such failures to be used in a constructive way, to pass the goal onto another tactic which might be able to do something further with it.

The simplicity of the definition of ORELSE using ML's notion of exception handling prompts the question of what it would look like without a similar notion, such as "backtrack" evaluation in Prolog (see [Clocksin81]). If, following [Paulson83d], we define:-

```
let FAILTAC(g) = failwith 'FAILTAC'
;;
```

as the tactic which always fails, then we have that FAILTAC is an identity for ORELSE i.e. for any tactic T,

$$T \text{ ORELSE } \text{FAILTAC} = T = \text{FAILTAC ORELSE } T$$

Also, ORELSE is associative, for any tactics T_1, T_2, T_3 :-

$$(T_1 \text{ ORELSE } T_2) \text{ ORELSE } T_3 = T_1 \text{ ORELSE } (T_2 \text{ ORELSE } T_3)$$

There are also the following relationships between the tacticals THEN and ORELSE, and the tactic FAILTAC:-

- (a) $T_1 \text{ THEN } (T_2 \text{ ORELSE } T_3) = (T_1 \text{ THEN } T_2) \text{ ORELSE } (T_1 \text{ THEN } T_3)$
- (b) $(T_1 \text{ ORELSE } T_2) \text{ THEN } T_3 = (T_1 \text{ THEN } T_3) \text{ ORELSE } (T_2 \text{ THEN } T_3)$
- (c) $\text{FAILTAC THEN } T = \text{FAILTAC} = T \text{ THEN FAILTAC}$

Properties (a) and (b) show that ORELSE distributes over THEN, while property (c) shows that FAILTAC is a zero (or an annihilator) for THEN. The properties follow simply by considering if and when tactics fail. Note that in comparing objects which may fail, the particular failure tokens returned are disregarded.

The last tactical introduced is known as REPEAT :tactic → tactic. Its action is to repeatedly apply the given tactic to an input goal until the tactic generates an exception. This behaviour is defined using tacticals already given, as follows:-

```
letrec (REPEAT T) g = ((T THEN (REPEAT T)) ORELSE IDTAC) g
;;
```

Normal termination is effected by applying T to a goal for which it fails. The ORELSE tactical then arranges for IDTAC to be applied, so terminating the recursion. Note that an indefinite recursion due to the occurrence of (REPEAT T) in the definition is prevented by the outermost application of the goal g to both sides.

Note that all these tacticals preserve validity; that is, if valid tactics are composed using the tacticals above, then the resulting composite tactic is also valid.

All of the tacticals introduced above make no assumptions about the :goal and :event types used. Hence, we are free to use the tacticals with any appropriate notion of :tactic. In Edinburgh LCF, a standard choice of :goal and :event types is generally used, and is discussed in section 2.5.3 of [LCF]. The standard choice made of ML type for :event is simply :thm. It now remains to motivate the corresponding choice for :goal.

Since theorems generally have the form:- $H \vdash fm$, where fm is

some formula and H is a collection of hypotheses, it would appear to be reasonable for goals to have a similar shape. Furthermore, it is conceivable that specific sets of *simprules* depending upon assumptions concerning the goal could prove useful. In fact, intermediate goals often contain assumptions specific to their context (i.e. due to case analysis, or structural induction) and, often, useful *simprules* can be based on these contextual assumptions. Hence, the standard choice of *:goal* is to put:-

`goal = form # simpset # (form)list`

where the *:form* component represents the required conclusion; the *:simpset* is the collection of locally available *simprules* and the *:form list* represents the collection of locally available assumptions (generally known as the assumption list).

Finally, the achievement relation between *:goal* and *:event* is easily defined; roughly, a theorem achieves a goal if both conclusions match and the hypotheses of the theorem are contained, up to alpha-conversion, amongst the assumptions of the goal.

1.4.4 Basic tactics in PPLAMBDA

A simple way of generating basic tactics is as "inverses" of inference rules in PPLAMBDA. Such inference rules are then used to produce the appropriate validation component. More complex tactics can be built up as combinations of these simpler tactics by the use of tacticals.

An example of a basic tactic of this kind is CONJTAC :tactic, defined in Figure 1.11. This tactic "inverts" the basic inference rule, CONJ, for introducing the conjunction of theorems (see Section 1.3.4). Roughly, CONJTAC takes conjunctive goals and

$$\text{CONJTAC} \\ ("fm1 \ \& \ fm2", ss, as1) \longrightarrow ([(fm1, ss, as1); (fm2, ss, as1)] , v)$$

where the validation function, $v : \text{thm list} \rightarrow \text{thm}$ is defined as:-

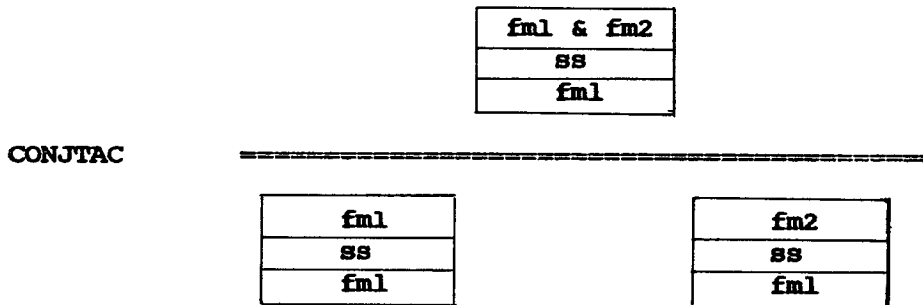
$$\begin{array}{c} v \\ [th1, th2] \longrightarrow \text{CONJ}(th1, th2) \end{array}$$

FIGURE 1.11

splits them appropriately, producing two subgoals; the validation part simply maps lists of theorems (of length 2) into their conjunction. Of course, if the tactic is presented with a goal not of the right form then it fails; similarly, the validation fails if the theorem list does not have length 2. It is easy to see that CONJTAC is valid since if th_1 achieves fm_1 and th_2 achieves fm_2 then, clearly, CONJ(th_1, th_2) achieves " $fm_1 \ \& \ fm_2$ ", which is the original goal.

In Figure 1.12 below, it is shown how CONJTAC may be defined in ML. Note that varstructs have been used in defining both the tactic and its validation component. Also, the test for conjunctive goals is left to whether or not the ML function destconj fails or not.

Avra Cohn in her thesis [Cohn79] introduced a useful notation for describing the functional behaviour of tactics. For example, CONJTAC would be described as:-



The "double line" indicates that a tactic is being defined; each box describes a goal consisting of formula, simpset and assumption list. It is assumed that if a goal does not match the goal described above the line, then the tactic fails. Also, the validation function is assumed to, at least, map lists of theorems achieving subgoals below the line to a theorem achieving a goal

```

let CONJTAC (fm, ss, fml) =
  (let (fml, fm2) = destconj(fm) in
   ([ (fml, ss, fml); (fm2, ss, fml) ], v)

  where v [th1; th2] = CONJ(th1, th2)

  ) ? (failwith 'CONJTAC')
;;

```

FIGURE 1.12



above the line.

This notation can also specify tactics which produce subgoals with "modified" simpset and assumption list components. For example, the standard tactic, CASESTAC, (see [LCF] page 140) does both, and can be represented as:-

fm
ss
fml

CASESTAC (t:term) =====

fm
{. t = TT} ∪ ss
"t = TT" . fml

fm
{. t = FF} ∪ ss
"t = FF" . fml

fm
{. t = I} ∪ ss
"t = I" . fml

Note that CASESTAC is a parameterised tactic and that the parameter is indicated beside the name of the tactic.

It is also useful to have variants of tactics which either do or do not alter simpsets. This situation arises because arbitrary additions to simpsets can destroy the useful characteristics of simpsets (like finite and unique termination, for example). So, there are two versions of IMPTAC, a tactic which deals with implicative goals in the following way:-

"fm ₁ & fm ₂ & ... & fm _n > fm"
ss
fml

IMPTAC =====

"fm ₂ ... & fm _n > fm"
ss
fm ₁ . fml

The antecedent of the goal formula is broken down into its conjunctive sub-formulae, and the first one is placed in the assumption list of the result. Also, the antecedent of the resulting goal then becomes the conjunction of all the remaining subformulae (if any). If no conjuncts remain then the goal formula of the result simply becomes fm.

The second variant of IMPTAC is called IMPTAC'. This acts in the same way as IMPTAC except that the formula placed in the

assumption list is also added to the simpset, if possible. So, diagrammatically this is:-

"fm ₂ & fm ₂ & ... & fm _n > fm"
ss
fml

IMPTAC'

"fm ₂ ... & fm _n > fm"
ss ∪ { . ⊢ fm ₁ }
fm ₁ . fml

If the formula cannot be added to the simpset then the tactic fails. In general, versions of named tactics that "modify" simpsets have their names annotated with an apostrophe, as for IMPTAC'.

In Section 1.4.2 on simplification, it was indicated that the LCF simplifier could be used in a goal directed way. A tactic, called SIMPTAC, is now defined which does this:-

fm
ss
fml

SIMPTAC

fm'
ss
fml

where fm' is obtained from fm by simplification using the local simpset ss. In fact, if fm can be simplified to the formula "TRUTH", then no subgoal need be generated at all. This standard tactic is expressed within ML in terms of the ML simplification function, simpform:-

```

let SIMPTAC (fm, ss, fml) =
  (let (fm', (), prf) = simpform (ss) (fm) in
    if istruth(fm') then ( [], λx. prf(AXTRUTH) )
      else ( [(fm',ss,fml)], prf o hd )
  ) ? (failwith 'SIMPTAC')
;;

```

Chapter 2

Mathematical background.

In this chapter, the semantics of PPLAMBDA is illustrated in the context of a presentation of Scott's theory of domains, which also develops the underlying computational intuition. The categorical approach to the theory of domain equations (initiated by Reynolds and Wand) is briefly sketched, as extended by the work of Lehmann, Smyth and Plotkin (see [SmythPlotkin82]).

Continuous algebras are used throughout the case studies; these are defined and their relation to the domain theoretic approach to data types is described. Also, various basic concepts from the theory of categories are briefly introduced, including the concepts of free and initial object.

A formalisation of a notion of equivalence predicate is given within PPLAMBDA. This notion is extensively used in Chapter 4 to formulate the correctness of a construction of multisets.

2.1 Domain Theory.

The aim of domain theory is to give a sound mathematical basis for the use of general recursive definitions of both functions and data spaces. The basic idea is to interpret recursive definitions in terms of fixed points of certain functionals.

For this approach to be mathematically successful, extra structure on the underlying spaces of data (called domains) and their admissible functions (or morphisms) is required. For example, every morphism will possess fixed points and, indeed, a least fixed point.

The development of domain theory was prompted in response to the mathematical difficulties encountered in defining the meaning of Programming Languages. Such definitions should be machine independent and not biased towards implementation on particular machines. The main idea is that programs in a programming language should precisely specify which function a program defines.

Because of the higher-type, behavioural aspect of such

definitions, the lambda calculus (see [Barendregt81]) plays a natural and almost essential role in language semantics. The use of this notation for combining functions of arbitrary type permits the complete expression of language meanings in a mathematically elegant functional form. However, this, in turn, had its difficulties. For example, a simple and rather natural definition of the meaning of user-definable procedures gave rise to potential self-application of the semantics function. This in turn implies the existence of domains that contain their own function space. But Cantor's theorem forbids this possibility - as long as domains correspond to sets and arbitrary functions are used. Moreover, arbitrary recursive definitions did not always make sense in the classical framework of total functions.

These difficulties were essentially surmounted by Dana Scott's application, in 1969, of lattice-theoretic techniques to construct extensional models of the untyped lambda calculus. The basic idea is to put a partial ordering on data spaces (roughly corresponding to "information content") and then to restrict attention to certain collections of functions. These are the continuous functions with respect to a certain topology on the class of partially-ordered sets used. Such functions may themselves be partially-ordered in a natural way so that they can also be used as data. By focussing attention on particular kinds of partial orders, the corresponding continuous functions can be guaranteed to have a least fixed point of a computationally significant form. Using this property, recursive definitions of functions may then be understood in terms of the least fixed point of a related (second-order) functional. It turns out that domains themselves can also be recursively specified; such domains correspond to initial objects in a category of domains (or, alternatively, as least fixed points of a retract on a universal domain).

The framework originally used by Scott made extensive use of complete lattices (see [Scott71] [Scott76]). However, this meant that each domain had to possess a maximum "overdefined" element, which tended to complicate the definitions of simple constructs such as the conditional (see [Plotkin78] or [Stoy77] p123 p197).

Moreover, for applications to the semantics of non-determinism and parallelism it turned out that this complete lattice framework is not entirely suitable and a weaker form is needed. This weaker, more general foundation uses ω -chain complete posets (with least element) instead of complete lattices. This is also sufficient to develop the semantic basis for recursion and permitted the formulation of the power domains (see [Plotkin76] [Smyth78]). Further conditions on the partial orders of interest, such as ω -algebraicity and consistent completeness, confer other technical advantages mainly concerned with applications to power domains.

The basic notion of partially ordered set (or poset) is assumed and can be found in [MB], for example. Ordering relations are usually denoted by symbols like $\sqsubseteq$ and $\prec$.

Let P be any poset. An ω -chain in P is a (countable) sequence $\langle p_i \rangle_{i \in \omega}$ such that $p_i \sqsubseteq p_{i+1}$, for each $i \in \omega$. The least upper bound of a chain $\langle p_i \rangle_{i \in \omega}$ is, when it exists, an element $p \in P$ such that $p_i \sqsubseteq p$, for each $i \in \omega$ (i.e. p is an upper bound of $\langle p_i \rangle_{i \in \omega}$) and if $q \in P$ such that $p_i \sqsubseteq q$, for all $i \in \omega$, then $p \sqsubseteq q$. When it exists, this element p is unique and is denoted by $\bigsqcup_{i \in \omega} p_i$.

For our purposes, a domain D is any poset with a least element (conventionally denoted by $\perp_D$) such that every ω -chain in D possesses a least upper bound (i.e. it converges). A domain D is said to be flat iff for every $x, y \in D$, $x \sqsubseteq y$ implies that $x = \perp_D$ or $x = y$. Finally, a domain D is easy iff it has no proper infinite chains.

Suppose that A and B are domains and that $f: A \rightarrow B$. Then, f is monotonic iff for all $x, y \in A$, if $x \sqsubseteq y$ then $f(x) \sqsubseteq f(y)$. A monotonic function $f: A \rightarrow B$ is said to be Scott-continuous (or just continuous) iff for any ω -chain $\langle a_i \rangle_{i \in \omega}$ in A ,

$$f\left(\bigsqcup_{i \in \omega} a_i\right) = \bigsqcup_{i \in \omega} f(a_i)$$

Note that the monotonicity of f implies that $\langle f(a_i) \rangle_{i \in \omega}$ is an ω -chain in B .

Now, it turns out that any function that is expressed by means of the λ calculus (using continuous function constants) is in turn a continuous function (see, for example, [Stoy77] pl25-127, [Bird76] pl67-169). This gives an obvious syntactical criterion

for determining the Scott continuity of specified functions.

A function $f: A \rightarrow B$ is said to be strict whenever $f(\perp_A) = \perp_B$. One of the most important properties of Scott-continuous (endo) functions is that they always possess a least fixed point of a computationally useful form. Let $f: A \rightarrow A$ be continuous, and consider:-

$$\bigsqcup_{i \in \omega} f^i(\perp_A) \equiv x$$

where, of course, $f^0 \equiv \text{id}_A$, and $f^{i+1} \equiv (f \circ f^i)$. So, $f(x) \equiv f(\bigsqcup_{i \in \omega} f^i(\perp_A)) \equiv \bigsqcup_{i \in \omega} f^{i+1}(\perp_A) \equiv x$. Hence, x is a fixed point. It's leastness follows from the argument that if p is any fixed point of f , then by induction, $f^{i+1}(\perp_A) \sqsubseteq p$. Hence, in the limit, $x \equiv \bigsqcup_{i \in \omega} f^i(\perp_A) \sqsubseteq p$, by definition of least upper bound.

This property is known as Kleene's 1st recursion theorem (see [Kleene52], p348). The element x certainly exists, for any continuous function $f: A \rightarrow A$, and so the least-fixed-point functional, FIX , can be defined:-

$$\text{FIX } f \equiv \bigsqcup_{i \in \omega} f^i(\perp_A)$$

It can be easily shown that FIX is itself a continuous functional. Note that $f(\text{FIX}(f)) \equiv \text{FIX}(f)$ and if $f(p) \equiv p$ then $\text{FIX}(f) \sqsubseteq p$.

It is important to note that monotonic (endo) function $g: A \rightarrow A$ may not have (least) fixed points, and even if they do, they do not generally have the form given above.

An important application of fixed points is to give the Computational Induction rule, for admissible predicates:-

$$\frac{\vdash \text{fm}[\perp/x], \quad \vdash \forall y. \text{fm}[y/x] \supset \text{fm}[f(y)/x]}{\vdash \text{fm}[(\text{FIX } f)/x]}$$

where:-

1. the formula fm admits induction (is admissible) in the variable $x:\alpha$.
2. the term $f:(\alpha \rightarrow \alpha)$ is not free in fm .
3. the variable $y:\alpha$ is not free in f .

A predicate (or formula) P on domain A is admissible iff, for

every ω -chain $\langle a_i \rangle_{i \in \omega}$ in A , if $P[a_i]$, for each $i \in \omega$, then $P[\bigcup_{i \in \omega} a_i]$.

Hence, an admissible predicate characterises a sub-domain of a domain A . The validity of the above rule follows from the characterisation of least fixed points (of continuous functions) in terms of iterates. This induction rule can essentially be found in [ScottDeBakker69] and in [Park70].

Formulae composed of conjunctions of (in)equations are always admissible in any of their free variables. More generally, if the "induction" variable x does not occur in any antecedent of an implication then, again, the formula is admissible. For further discussion on this matter, consult [LCF] p77-78, [Manna74] p393-394, or [Bird76] p172-173. Note that admissibility of a formula (standing for a predicate in its free variables) can be formalised within LCF, by formalising chains as order-homomorphisms of the natural numbers. However, such a method would naturally require (the standard model of) the natural numbers to be given in a yet more fundamental way.

2.1.1 Domain Operators.

In domain theory, there are many useful constructions of domains, frequently dependent upon other domains. Such constructors usefully take the form of domain operators mapping domains into domains. Such operators are also generally associated with specific collections of morphisms for relating these constructions to other domains.

Most of the domain operators are constructions on domains which, in their turn, have a categorical basis and are usually related to functors between certain categories (see Section 2.2); those of particular interest are given as follows. The category CPO has as its objects all (small) cpo's (i.e. complete partial orders with least element) with morphisms all Scott continuous functions between domains. The obvious compositions and identities are used. The category CPO_1 is like CPO but with strict Scott continuous functions for morphisms.

Note that both of the categories defined above take the class of all (small) domains as their objects and sub-classes of the

Scott continuous functions as their morphisms.

To give precise types to morphisms, we introduce the function spaces. Let A and B be any domains. Then:-

$$[A \rightarrow B] = \{f: A \rightarrow B \mid f \text{ is Scott continuous} \}$$

This is ordered pointwise, for any $f, g: [A \rightarrow B]$, by:-

$$\vdash f \sqsubseteq g \iff \forall x:A. f(x) \sqsubseteq g(x)$$

In some sense, each function is viewed as a vector with one coordinate for each argument. The least element is the function Ω where, for any $x \in A$, $\Omega(x) = \perp_B$. In LCF, the uniform notation of $\perp_D$ is used for least elements of domain D and so Ω corresponds to $\perp_{A \rightarrow B}$. It turns out that $[A \rightarrow B]$ is a domain just as long as B is a domain.

The collection of all strict continuous functions from A to B is denoted by $[A \rightarrow_1 B]$ and is a sub-domain of $[A \rightarrow B]$. Both of the above domain constructions form bi-functors (contravariant in the first argument) $: \mathbf{CPO}^{\text{OP}} \times \mathbf{CPO} \rightarrow \mathbf{CPO}$. The morphism part is a functional:-

$$\text{Fun} : [[A' \rightarrow A] \rightarrow [[B \rightarrow B'] \rightarrow [[A \rightarrow B] \rightarrow [A' \rightarrow B']]]]$$

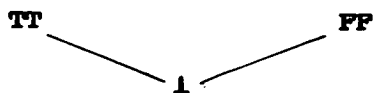
given by

$$\vdash \text{Fun } f \, g \, h \, a \equiv (g \circ h \circ f) \, a$$

Generally speaking, the functions of interest that are used later on are Scott-continuous. So from now on, the statement that $f : [A \rightarrow B]$ will be abbreviated to the statement that $f : A \rightarrow B$, unless otherwise indicated.

Some basic constant domains are now introduced. The trivial domain, dot, consist of $\{1\}$ with the trivial identity ordering. The two part domain, 1, (or I) consists of the set $\{1, T\}$. (T is known as top with the ordering $1 \sqsubseteq T$).

The standard domain of (internal) truth values, denoted by tr consists of $\{1, TT, FF\}$ ordered by:-



Note that $1 \neq TT \neq FF$ and indeed that $TT \not\leq FF$ and $FF \not\leq TT$.

These properties serve as the basis for the LCF rule for Case analysis and the Contradiction (or reductio ad absurdum):-

$$\begin{array}{l} \vdash (x = TT) \supset \text{fm} \\ \vdash (x = FF) \supset \text{fm} \\ \vdash (x = 1) \supset \text{fm} \end{array}$$

$$\text{Cases rule} \quad \frac{\quad}{\vdash \text{fm}}$$

This schema provides a primitive, although sufficient, means for proof by case analysis. There are the rules of contradiction (or absurdity) which assert that if a manifestly absurd result is proven then any property is valid. Specifically, we have:-

$$\text{Contradiction} \quad \frac{\vdash \text{tm}_1 \leq \text{tm}_2 : \text{tr}}{\vdash \text{fm}}$$

where both tm_1 and tm_2 are distinct truthvalued constants (i.e. one of TT , FF or $1:\text{tr}$) where $\text{tm}_1 \neq 1$. Together, these rules ensure that the truth values have their standard interpretation. Some general purpose functions involving tr are introduced as follows, for each domain D :-

$$\begin{array}{l} \partial : D \rightarrow \text{tr} \\ \text{COND} : \text{tr} \rightarrow D \rightarrow D \rightarrow D \end{array}$$

with the intended meaning:-

$$\begin{array}{l} \partial(x) = \begin{cases} TT & \text{if } x \neq 1_D \\ 1:\text{tr} & \text{otherwise} \end{cases} \\ \text{COND } t \ x_1 \ x_2 = \begin{cases} x_1 & \text{if } t = TT \\ x_2 & \text{if } t = FF \\ 1_D & \text{otherwise} \end{cases} \end{array}$$

In PPLAMBDA, these are axiomatised in the following way:-

$$\begin{array}{l} \vdash \forall x:D. \partial(x) \leq TT \\ \vdash \forall x:D. \partial(x) = 1 \supset x = 1_D \\ \vdash \forall x_1 \ x_2:D. \text{COND } TT \ x_1 \ x_2 = x_1 \\ \vdash \forall x_1 \ x_2:D. \text{COND } FF \ x_1 \ x_2 = x_2 \\ \vdash \forall x_1 \ x_2:D. \text{COND } 1:\text{tr} \ x_1 \ x_2 = 1_D \end{array}$$

We permit the notation $(t \Rightarrow x_1 \mid x_2)$ as an abbreviation for

COND (t) (x₁) (x₂).

PPLAMBDA does not pre-define the usual (internal) propositional functions on the (internal) truth values. They, of course, can be introduced (see Section 2.1.2 below). Indeed, the parallel or can be introduced (as the infix operator paror) viz:-

- paror - :tr → tr → tr

with the axioms:-

⊢ ∀t :tr. TT paror t ≡ TT

⊢ ∀t₁ t₂:tr. (t₁ ⊆ FF) ⊃ (t₁ paror t₂) ≡ t₂

Full (cartesian) product operator (- # -) of two domains A and B can be modelled set theoretically by:-

(A # B) = { (a,b) ∈ (A × B) | a ∈ A & b ∈ B }

with the componentwise ordering, for any a, a' ∈ A and b, b' ∈ B

(a, b) ⊆ (a', b') ⊃ (a ⊆ a') & (b ⊆ b')

The least element, $1_{A\#B}$ is equal to $\langle 1_A, 1_B \rangle$. There are several useful continuous functions associated with the full product:-

PAIR :A → B → A # B

FST :A # B → A

SND :A # B → B

with the PPLAMBDA axioms:-

⊢ ∀p:A # B. PAIR (FST p) (SND p) ≡ p

⊢ ∀x:A y:B. FST(PAIR x y) ≡ x

⊢ ∀x:A y:B. SND(PAIR x y) ≡ y

The infix notation (x, y) is permitted to abbreviate PAIR x y.

The domain operator (- # -) can be viewed as a functor:

CPO² → CPO with the morphism part:-

Prod:(A → A') # (B → B') → ((A # B) → (A' # B'))

defined by:-

$$\vdash \text{Prod}(f, g)(a, b) \equiv (f(a), g(b))$$

The smash product operator $(- \otimes -)$ of two domains A, B can be modelled by:-

$$(A \otimes B) = \{ (a, b) \in (A \# B) \mid \partial(a) \equiv \partial(b) \}$$

with the ordering and least element as for $(A \# B)$. This domain can be axiomatically determined using the following morphisms:-

$$\begin{aligned} (- \otimes -) &: A \rightarrow B \rightarrow (A \otimes B) \\ P1 &: (A \otimes B) \rightarrow A \\ P2 &: (A \otimes B) \rightarrow B \end{aligned}$$

The morphism $\otimes$ represents "smash" pairing and is intended to have the meaning:-

$$a \otimes b = \begin{cases} (a, b) & \text{if } \partial(a) \equiv \text{TT} \equiv \partial(b) \\ (\perp_A, \perp_B) & \text{otherwise} \end{cases}$$

Hence, if either "argument" is undefined then so is the result. This is formally axiomatised in PPLAMBDA as follows:-

$$\begin{aligned} &\vdash \forall p: A \otimes B. (P1(p) \otimes P2(p)) \equiv p \\ &\vdash \forall x: A \ y: B. P1(x \otimes y) \equiv \partial(y) \Rightarrow x \mid \perp_A \\ &\vdash \forall x: A \ y: B. P2(x \otimes y) \equiv \partial(x) \Rightarrow y \mid \perp_B \end{aligned}$$

The domain operator $(- \otimes -)$ can be viewed as a bi functor : $\text{CPO}_1^2 \rightarrow \text{CPO}_1$, with the morphisms part

$$\text{SmProd} : (A \rightarrow_1 A') \# (B \rightarrow_1 B') \rightarrow (A \otimes B \rightarrow_1 A' \otimes B')$$

defined as:-

$$\vdash \text{SmProd}(f, g)(sp) \equiv (f(P1 \ sp) \otimes g(P2 \ sp))$$

The explicit use of the smash projections $P1$ and $P2$ is made since $\otimes$ is NOT injective, due to both left and right strictness.

The coalesced sum operator $(- + -)$ of two domains A, B can be modelled set theoretically as follows:-

$$(A + B) = \{1\} \times (A \setminus \{\perp_A\}) \cup \{2\} \times (B \setminus \{\perp_B\}) \cup \{1\}$$

where the ordering satisfies the following properties:-

- (a) $\forall x \in (A + B). 1 \in x$
 (b) $\forall a_1, a_2 \in A \setminus \{1_A\}. \langle 1, a_1 \rangle \in \langle 1, a_2 \rangle \iff a_1 \in_A a_2$
 (c) $\forall b_1, b_2 \in B \setminus \{1_B\}. \langle 2, b_1 \rangle \in \langle 2, b_2 \rangle \iff b_1 \in_B b_2$

The basic idea is that if $x \in y$ then either x is undefined or both x and y belong to the same summand (i.e. they have the same tag) and that their respective untagged values are comparable with respect to the appropriate domain.

This domain operator can be axiomatised within PPLAMBDA using the following functions:-

INL : $A \rightarrow (A + B)$
 INR : $B \rightarrow (A + B)$
 OUTL : $(A + B) \rightarrow A$
 OUTR : $(A + B) \rightarrow B$
 ISL : $(A + B) \rightarrow \text{tr}$
 ISR : $(A + B) \rightarrow \text{tr}$

with the following axioms:-

$\vdash \forall s:A + B. \partial(\text{ISL}(s)) \equiv \partial(s)$
 $\vdash \forall a:A. \text{ISL}(\text{INL}(a)) \equiv \partial(a)$
 $\vdash \forall b:B. \text{ISL}(\text{INR}(b)) \equiv \partial(b) \Rightarrow \text{FF} \mid \text{TT}$

 $\vdash \forall a:A. \text{OUTL}(\text{INL}(a)) \equiv a$
 $\vdash \forall b:B. \text{OUTR}(\text{INR}(b)) \equiv b$

 $\vdash \forall a:A. \text{OUTR}(\text{INL}(a)) \equiv 1_B$
 $\vdash \forall b:B. \text{OUTL}(\text{INR}(b)) \equiv 1_A$

 $\vdash \forall s:A + B. (\text{ISL}(s) \Rightarrow \text{INL}(\text{OUTL}(s)) \mid \text{INR}(\text{OUTR}(s))) \equiv s$
 $\vdash \forall s:A + B. \text{ISR}(s) \equiv \text{ISL}(s) \Rightarrow \text{FF} \mid \text{TT}$

The coalesced sum operator $(- + -)$ can be viewed as a bi-functor : $\text{CPO}_1^2 \rightarrow \text{CPO}_1$ with morphism part:-

$\text{Sum} : (A \rightarrow_1 A') \# (B \rightarrow_1 B') \rightarrow ((A + B) \rightarrow_1 (A' + B'))$

defined by:-

$\vdash \text{Sum}(f, g)(s) \equiv \text{ISL}(s) \Rightarrow \text{INL}(f(\text{OUTL}(s))) \mid \text{INR}(g(\text{OUTR}(s)))$

The "lifting" operator -1 on a domain A can be defined set theoretically as:-

$A_1 = (\{0\} \times A) \cup \{1\}$

with the ordering satisfying the properties:-

$$\begin{aligned} \forall x \in A_1. \quad 1 \in x \\ \forall a_1, a_2 \in A. \quad \langle 0, a_1 \rangle \in \langle 0, a_2 \rangle \quad \Leftrightarrow \quad a_1 \in a_2 \end{aligned}$$

The idea is that if $x \in y$ then either $x = 1$ or both x and y are tagged with 0 and each are related elements in A .

This is formalised in PPLAMBDA with the assistance of the following functions:-

$$\begin{aligned} \text{UP} &: A \rightarrow A_1 \\ \text{DOWN} &: A_1 \rightarrow A \end{aligned}$$

with the axioms:-

$$\begin{aligned} \vdash \forall a:A. \quad \partial(\text{UP}(a)) &= \text{TT} \\ \vdash \forall a:A. \quad \text{DOWN}(\text{UP}(a)) &= a \\ \vdash \forall b:B. \quad \partial(b) = \text{TT} \supset \text{UP}(\text{DOWN}(b)) &= b \end{aligned}$$

The domain operator, -1 , can be viewed as a functor $\text{CPO} \rightarrow \text{CPO}_1$ with morphism part:-

$$\text{Lift} : (A \rightarrow B) \rightarrow (A_1 \rightarrow_1 B_1)$$

defined by:-

$$\vdash \text{Lift } f \text{ ua} = \partial(\text{ua}) \Rightarrow \text{UP}(f(\text{DOWN}(\text{ua}))) \mid 1_{B_1}$$

The lifting operator notation is often used to stand for the natural injection of sets into (flat) domains. If S is any set then $S_1 = (\{0\} \times S) \cup \{1\}$ with the ordering satisfying the obvious axioms.

The domain operators presented above generally have further categorical significance in that each embodies a particular kind of universal construction (see, for example, [MB] p129). This particular significance is briefly alluded to below (see also [SmythPlotkin82] for more details).

The trivial domain, dot, is the zero (i.e. initial and terminal) object in the category CPO_1 . The function space $[A \rightarrow B]$ is the exponential object of B by A in the (Cartesian closed) category CPO , and $[A \rightarrow_1 B]$ is the corresponding exponential in the closed category CPO_1 . The full product $(A \# B)$ is the categorical product of A with B in CPO and $(A \otimes B)$ is the corresponding categorical product in CPO_1 . Finally, $(A + B)$ is the categorical sum in CPO_1 .

2.1.2 Standard PPLAMBDA theories.

In the cases studies given in Chapters 3 and 4, various standard results and auxiliary definitions will be needed. This basic material is collected together and formalised in LCF within a small collection of interrelated theories, which are now discussed briefly below. Appendix 3 contains a full, unformatted listing of these theories. Figure 2.1 contains a diagram of their inheritance relationship and Figure 2.2 contains a sample of theorems from these theories.

LCF does not have a standard smash product domain operator. As this proves to be crucial later on, a theory called SMASH is given in which this type operator is defined which follows the presentation of smash product as given above. Unfortunately, the exact concrete syntax presented in Appendix 2 could not be used as given. Also, the axiom S1 and facts F1 - F10 from the theory SMASH are always included within the standard simpset BASICSS.

The theory KERNEL then builds upon this theory, and is used to record standard results concerning pure PPLAMBDA (based upon SMASH).

The theory BASIC contains no constants and merely acts to join all its sub-theories. The theories NAT and NATFUN are an axiomatisation of a flat domain of natural numbers and some basic arithmetic functions such as addition and multiplication. The theory MORPH defines the polymorphic function composition combinator by $(f \circ g) \equiv \lambda x. f(g(x))$. None of NAT, NATFUN or MORPH

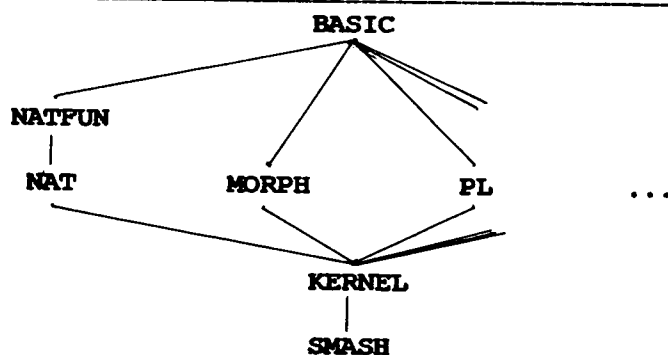


FIGURE 2.1

A selection of theorems proven in the SMASH, KERNEL and PL theories.

Theory SMASH.

```

'S1'  ]-  $\forall p: (\alpha \otimes \beta). (P1\ p) \otimes (P2\ p) = p$ 
'S2'  ]-  $\forall a: \alpha\ b: \beta. P1(a \otimes b) = \partial(b) \Rightarrow a \mid 1$ 
'F1'  ]-  $(1 \otimes 1) = 1$ 
'F3'  ]-  $\forall a: \alpha\ b: \beta. \partial(a) = TT \supset P2(a \otimes b) = b$ 
'F7'  ]-  $\forall b: \beta. (1 \otimes b) = 1$ 
'F10' ]-  $\forall a: \alpha\ b: \beta. \partial(a \otimes b) = (\partial(a) \Rightarrow \partial(b) \mid 1)$ 
'F11' ]-  $\forall x\ y: (\alpha \otimes \beta). (P1\ x = P1\ y) \ \& \ (P2\ x = P2\ y) \supset x = y$ 

```

Theory **KERNEL.**

```

'condUU'      ]-  $\forall t:tr\ x:y:\alpha. \partial(t \Rightarrow x \mid y) \equiv t \Rightarrow \partial(x) \mid \partial(y)$ 
'DEFxTT'      ]-  $\forall t:tr\ x:\alpha. \partial(x) \Rightarrow TT \mid t \equiv \partial(x)$ 
'DEFxUU'      ]-  $\forall t:tr\ x:\alpha. \partial(x) \Rightarrow \perp \mid t \equiv \perp$ 
'BooleanCond' ]-  $\forall t:tr. t \Rightarrow TT \mid FF \equiv t$ 
'TTDEft'      ]-  $\forall t:tr. t \Rightarrow TT \mid TT \equiv \partial(t)$ 
'DEFDEF'      ]-  $\forall x:\alpha. \partial(\partial(x)) \equiv \partial(x)$ 

```

Theory PL.

```

'notTT'  ]- not(TT) = FF
'notFF'  ]- not(FF) = TT
'notUU'  ]- not(1)  = 1

'TTor'   ]-  $\forall t:tr. TT \text{ or } t = TT$ 
'FFor'   ]-  $\forall t:tr. FF \text{ or } t = FF$ 
'UUor'   ]-  $\forall t:tr. 1 \text{ or } t = 1$ 

'TTand'  ]-  $\forall t:tr. TT \text{ and } t = t$ 
'FFand'  ]-  $\forall t:tr. FF \text{ and } t = FF$ 
'UUand'  ]-  $\forall t:tr. 1 \text{ and } t = 1$ 

'orAssoc'      ]-  $\forall t1\ t2\ t3:tr.$ 
                   $(t1 \text{ or } t2) \text{ or } t3 = t1 \text{ or } (t2 \text{ or } t3)$ 

'andAssoc'     ]-  $\forall t1\ t2\ t3:tr.$ 
                   $(t1 \text{ and } t2) \text{ and } t3 = t1 \text{ and } (t2 \text{ and } t3)$ 

'andAnalysis'  ]-  $\forall t1\ t2:tr.$ 
                   $(t1 \text{ and } t2 = TT) \supset (t1 = TT) \ \& \ (t2 = TT)$ 

'orAnalysis'   ]-  $\forall t1\ t2:tr.$ 
                   $(t1 \text{ or } t2 = FF) \supset (t1 = FF) \ \& \ (t2 = FF)$ 

'AndComm'      ]-  $\forall t1\ t2\ t3:tr.$ 
                   $\partial(t1) = TT \ \& \ \partial(t2) = TT \ \supset$ 
                   $t1 \text{ and } (t2 \text{ and } t3) = t2 \text{ and } (t1 \text{ and } t3)$ 

```

FIGURE 2.2

play any further role within the case studies; their inclusion shows how a (standard) body of facts could have been arranged to share common parts and to combine facts together. By including the single top theory BASIC within a theory, all of its component theories and their ancestors can be made available.

The theory PL contains the definitions of various propositional constants. For example, the truth value connectives "and", "or" and "not" are defined here using the conditional function as follows:-

```
(- and -) :tr → tr → tr
(- or -) :tr → tr → tr
  not :tr → tr
```

```
'and' ]- ∀t1 t2:tr. (t1 and t2) ≡ (t1 ⇒ t2 | FF)
'or'  ]- ∀t1 t2:tr. (t1 or t2) ≡ (t1 ⇒ TT | t2)
'not' ]- ∀t1:tr. not(t1) ≡ (t1 ⇒ FF | TT)
```

Various properties of these connectives are developed (using Boolean case analysis) in this theory which will be used later on.

```
'andAssoc' ]- ∀t1 t2 t3:tr.
              (t1 and t2) and t3 ≡ t1 and (t2 and t3)
'andRefl'  ]- ∀t1:tr. t1 and t1 ≡ t1
```

It will be generally more useful not to use the above axioms within simplification but instead to use axioms similar to these for "and":-

```
'TTand' ]- ∀t:tr. TT and t ≡ t
'FFand' ]- ∀t:tr. FF and t ≡ FF
'UUand' ]- ∀t:tr. 1 and t ≡ 1
```

Applying these rules during simplification disturbs the structure of the overall expression less and does not introduce further occurrences of the conditional. Thus, other simplification rules relating to propositional expressions such as these (e.g. De Morgans laws, associativity) can be applied in conjunction with these. Similar rules are also proven for the other operators.

2.1.3 Properties of isomorphisms in CPO.

In later sections, a number of simple properties about isomorphisms between domains are needed. These lemmas have also been formally proven within the theory `KERNEL`. Suppose that α and β are domains with $f : [\alpha \rightarrow \beta]$ and $g : [\beta \rightarrow \alpha]$.

Lemma 2.1

'gUU' $\vdash \forall g \ f. (\forall a:\alpha. g(f(a)) = a) \supset g(\perp_\beta) = \perp_\alpha$

Proof

By monotonicity of f and g . Clearly $\perp_\beta \sqsubseteq f(\perp_\alpha)$. So, applying g to both sides, we get $g(\perp_\beta) \sqsubseteq g(f(\perp_\alpha))$. However, from the above, $g(f(\perp_\alpha)) = \perp_\alpha$. Hence, $g(\perp_\beta) \sqsubseteq \perp_\alpha$. Clearly, $\perp_\alpha \sqsubseteq g(\perp_\beta)$ and so, by anti-symmetry of $\sqsubseteq$, $g(\perp_\beta) = \perp_\alpha$.

QED

Corollary 2.2

If f and g form an isomorphism pair then both functions are strict.

Proof Apply the above lemma twice. QED

Lemma 2.3

'DEFf'

] - $\forall f \ g.$

$(\forall a:\alpha. g(f(a)) = a) \ \& \ (\forall b:\beta. f(g(b)) = b) \supset$
 $\forall a:\alpha. \partial(f(a)) = \partial(a)$

Proof

The proof proceeds by cases on the definedness of an arbitrary value a from α .

Suppose that $\partial(a) = \perp$. Then $a = \perp$ and so, by the above lemma, $f(a) = \perp_\beta$ and then $\partial(f(a)) = \perp$, giving the result here.

Now suppose that $\partial(a) = \text{TT}$. We now proceed by cases upon the value of $f(a)$. If $\partial(f(a)) = \text{TT}$ then the result follows trivially.

On the other hand, if $\partial(f(a)) = 1$ then $f(a) = 1$. Hence, applying g to both sides gives $g(f(a)) = g(1)$. Using the isomorphism and strictness properties of f and g shows that $a = 1$. However, $\partial(a) = \text{TT}$ and so $1 = \partial(1) = \partial(a) = \text{TT}$. This is absurd and also completes the proof.

QED

2.1.4 Domain Equations.

In denotational semantics, a natural method of introducing domains is to define them by the so-called "domain equation" technique. Informally, a domain equation is a relationship between domains of the form:-

$$A \cong R$$

where R is some expression involving domains and domain operators, and possibly containing A itself. The "equation" asserts that the domain A (the "abstraction") is defined to be (continuously) isomorphic, or homeomorphic^{*}, to the domain R , (the "representation"). It is mathematically expedient (in this framework, at least) to require solutions of such equations only up to isomorphism. Moreover, the use of (continuous) isomorphisms suggests a natural formalisation of domain equations within PPLAMBDA.

For example, given the dot domain, it is possible to introduce the other standard constant domains.

$$\begin{aligned} 1 &\cong \text{dot}_1 \\ T &\cong 1 + 1 \end{aligned}$$

More interesting, is the possibility of having recursive domain equations. Indeed, one of the original objectives of domain theory was to give a rigorous treatment of just how domains may themselves be recursively defined. The need for recursive definitions of domains arises naturally when constructing formal definitions of programming languages (see [Gordon79] or [BjornerJones82]).

* Two (topological) spaces are said to be homeomorphic if they are continuously bi-jjective, with continuous inverse.

The initial challenge for domain theory was to give a construction of a non-trivial model of the (untyped) lambda calculus. Such a model involves finding a nontrivial domain D which is isomorphic to the full Scott continuous function space $[D \rightarrow D]$. In other words, this is:-

$$D \cong [D \rightarrow D]$$

It is crucial to ensure continuity of the bijections; it is not possible to find any (non-trivial) partial order P which is monotonically isomorphic to its monotonic endofunction space, ordered pointwise.

Solutions of the above equation are examples of reflexive domains (i.e. domains that satisfy domain equations involving their own function space e.g. $D \cong A + B \times [D \rightarrow C]$ for some domains A, B, C). However, the work presented herein does not require such domains and are not specifically dealt with any further.

For the most part, the kind of data that is used consists of various shapes of finite or infinite tree structure. Hence, the use of the domain equation technique here is more in the spirit of the work of [LehmannSmyth78] and [SmythPlotkin82]. It is also related to the ADJ group's use of continuous algebras (see [ADJ77]) to give pure tree domains (i.e. anarchic algebras).

We continue by specifying the basic class of domain equations needed in more formal terms. First of all, the statement that a domain A satisfies a domain equation $A \cong R$ is equivalent to the existence of an isomorphism $\theta: A \leftrightarrow R: \theta^{-1}$. There is a naming convention for such isomorphisms; absA: $R \rightarrow A$ (i.e. the "abstraction" function) and repA: $A \rightarrow R$ (i.e. the "representation" function) with the usual axioms:-

$$\begin{aligned} \vdash \forall a: A. \text{absA}(\text{repA}(a)) &= a \\ \vdash \forall r: R. \text{repA}(\text{absA}(r)) &= r \end{aligned}$$

The class of domain equations that are needed have the following, general form:-

$$A \cong B_1 + B_2 + \dots + B_n \quad ; n \in \{1, 2, 3, \dots\}$$

where each summand domain B_i is a (possibly "lifted") product of

one or more domains (which may include A itself). The products involved can be either the full or smash product operator (where factors of the product may also be lifted).

Now, a natural and very general mathematical approach to "equation solving" is to formulate such questions in terms of the existence and uniqueness of fixed points of appropriate function-like objects. So, in keeping with this approach, a domain operator T is introduced, roughly, as follows:-

$$T(A) = B_1 + B_2 + \dots + B_n$$

So, the domain equation can now be expressed in fixed-point form as:-

$$A \cong T(A)$$

where, as suggested above, solutions correspond to particular choices of domain A with a pair of (continuous) isomorphisms

$\text{abs}_A: [T(A) \rightarrow A]$ and $\text{rep}_A: [A \rightarrow T(A)]$

The crux of the problem is to determine when solutions exist and, when there is more than one, which solution is intended. Since the domain operator T is formed as a combination of standard domain operators (like $(- + -)$, $(- \# -)$, etc.) and basic domains (like 1), many (non-isomorphic) solutions will usually exist.

One answer to this question is to take the "least" (or initial) solution within an appropriate category. The motivation is that, for such "least" solutions, each element can be given directly in terms of a well-founded "expression", or as limits of such "expressions". It will then turn out that "structural induction" is valid for these solutions. Because of the completeness property of domains, these least solutions to domain equations can contain non-trivial limit points, corresponding to "infinite" elements. The existence of such limit points will depend upon the form of the domain equation; this is discussed in Section 2.1.7.

The remainder of this section discusses the notion of "leastness" used above, sketching a category theoretic proof of their existence (due to Plotkin and Smyth) and in addition indicates the connection to "well-foundedness" of elements and

"structural induction" schemes. A thorough and more precise account of the theory of recursive domain equations may be found in [SmythPlotkin82] and [LehmannSmyth78].

The first step is to say how domains can approximate each other, taking proper account of their partial order structures. To do this, the category of embeddings, $\mathbf{CPO}^E$, is introduced. This has objects all (small) domains, like $\mathbf{CPO}$, but with embeddings as morphisms. An embedding is a continuous function $\phi : [A \rightarrow B]$ such that $\exists \psi : [B \rightarrow A]$ such that:-

- $$\begin{aligned} (1) \quad & \forall a:A. \quad \psi(\phi(a)) \sqsubseteq a && (\text{i.e. } \psi \circ \phi \sqsubseteq \text{id}) \\ (2) \quad & \forall b:B. \quad \phi(\psi(b)) \sqsubseteq b && (\text{i.e. } \phi \circ \psi \sqsubseteq \text{id}) \end{aligned}$$

The function $\psi : [B \rightarrow A]$ is called a projection and is, in fact, unique. Following [SmythPlotkin82], suppose that $\chi : [B \rightarrow A]$ is also a projection for ϕ (i.e. $\chi \circ \phi = \text{id}_A$ and $\phi \circ \chi \sqsubseteq \text{id}_B$). Then, $\chi = \text{id}_A \circ \chi = (\psi \circ \phi) \circ \chi = \psi \circ (\phi \circ \chi) \sqsubseteq \psi \circ \text{id}_B = \psi$. Hence $\chi \sqsubseteq \psi$. Similarly, $\psi \sqsubseteq \chi$, and so, $\chi = \psi$, as required. Since the projection is unique, it can be denoted by ϕ^R for the embedding ϕ .

The embeddings can be used to give a natural idea of approximation between domains. The domain D can be embedded into D' (written as $D \alpha D'$) iff there exists an embedding $\phi : [D \rightarrow D']$. The domain-theoretic notion of (continuous) embedding roughly corresponds to the set-theoretic notion of subset. Note that saying $D \alpha D'$ (i.e. D approximates D') is equivalent to saying that there is a morphism in $\mathbf{CPO}^E$ from D to D' . It turns out that α is a preorder on the class of domains (but NOT a partial order, not even up to isomorphism).

As promised above, we turn to a sketch of the existence proof of initial solutions of domain equations, due to Plotkin and Smyth. The basic idea is similar to that for solving recursive function equations.

The first step is to construct the sequence of domains $\langle A_i \rangle_{i \in \omega}$ given by $A_0 = \text{dot}$ and $A_{i+1} = T(A_i)$ for any $i \in \omega$.

Now T is an (co-variant) endofunctor of $\mathbf{CPO}^E$, since it is a composition of domain operators, each of which is an appropriate functor on $\mathbf{CPO}^E$. Hence, T possesses a morphism part determined by how T is composed. This morphism part is traditionally also

denoted by T and is a (continuous) functional of type $:(A \rightarrow B) \rightarrow (T(A) \rightarrow T(B))$.

Accordingly, the domain sequence $\langle A_i \rangle_{i \in \omega}$ may be associated with the sequence $\langle f_i: A_i \rightarrow A_{i+1} \rangle_{i \in \omega}$ defined by $f_0 = \text{id}_{A_0 \rightarrow A_1}$ and $f_{i+1} = T(f_i)$. Each of the f_i 's are embeddings, since T is an endofunctor of CPO^E and f_0 is trivially an embedding.

The sequence $\langle A_i, f_i: A_i \rightarrow A_{i+1} \rangle_{i \in \omega}$ of domains and embeddings is an example of an approximation chain of domains. Now, it turns out that, for such chains in CPO^E , there is an analogue to the notion of "least upper bound". Each chain can be associated with a class of constructs called cones. Each cone consists of a domain, called the apex, and a countable sequence of embeddings, from each chain element into the apex, which satisfies a certain commuting condition. This condition ensures that the entire chain embeds in a consistent way into the apex.

Now, given any approximating chain in CPO^E , the class of all cones over this chain forms a category in a natural way. It then turns out that this category will always possess initial objects (i.e. initial cones). Roughly speaking, the apex domain of an initial cone can be visualised by forming the "union" of the chain elements (taking care to identify all corresponding points) and then completing it to form a domain by adding proper limit points as necessary. A more precise construction is to take:-

$$\{ \langle d_n \rangle_{n \in \omega} \in \prod_{k \in \omega} D_k \mid \forall i \in \omega. d_i = \theta_i^R(d_{i+1}) \}$$

ordered pointwise, for the chain $\langle D_i, \theta_i: D_i \rightarrow D_{i+1} \rangle_{i \in \omega}$

So, returning to domain equations and their "least" solutions, consider the category of cones over the chain $\langle A_i, f_i \rangle_{i \in \omega}$ defined above. Let A^* be the apex domain of an initial cone in this category. Now it can be shown that, assuming that T is a covariant functor and that it preserves initial cones (i.e. that it is ω -continuous), then (i) $A^* \cong T(A^*)$ and (ii) if $D \cong T(D)$ then $A^* \alpha D$. (In general T will have these properties due to its expression as a composition of basic domain operators). Hence, A^* is an appropriate "least", or initial, fixed point of the domain operator T and may be denoted as $\mu X.T(X)$ (since initial cones are

determined up to isomorphism).

Note that each of the A_i (for $i > 0$) are domains whose defined elements are seen to roughly correspond to certain tree-like values of depth $(i - 1)$. (This is illustrated by the example given below). Hence, taking the ω -colimit of the chain $\langle A_i, f_i \rangle_{i \in \omega}$ is then roughly equivalent to taking the "union" of the A_i 's and completing it by adding limit points as necessary. Such limit points will correspond to the least upper bounds of ascending ω -chains of elements, or "finite terms", from the A_i 's. Because of the continuity of the isomorphism between A and $T(A)$, this ensures that limits of limits are again limits. That is, if $\langle a_{ij} \rangle_{j \in \omega}$ is an ω -chain, for each $i \in \omega$, then:-

$$\bigsqcup_{i \in \omega} \left(\bigsqcup_{j \in \omega} a_{ij} \right) = \bigsqcup_{k \in \omega} a_{n_k n_k} \text{ for some } n_k$$

We now turn to a simple example of a domain of (finite and infinite) lists of a given domain of data, A . The domain equation is:-

$$L \cong 1 + (A \# L)$$

The corresponding domain operator, T , is:-

$$T(D) = 1 + (A \# D)$$

Hence, $L = \mu D.T(D)$

Let $\text{abs}_L: T(L) \rightarrow L$ and $\text{rep}_L: L \rightarrow T(L)$ be a fixed pair of isomorphisms; their existence is ensured by the discussion above. Each summand of the domain operator T gives rise to a "constructor" function, or a basic constant:-

$$\begin{aligned} N &: 1 \\ C &: A \# L \rightarrow L \end{aligned}$$

defined by:-

$$\begin{aligned} N &\equiv \text{abs}_L(\text{INL}(T)) \\ C(f, l) &\equiv \text{abs}_L(\text{INR}(f, l)) \end{aligned}$$

These can be used to construct elements of L corresponding to either summand in the equation..

Using the "representation" function, repL , other useful functions can be introduced. For example, there are the standard discriminator functions, one for each summand:-

$\text{IsN} : L \rightarrow \text{tr}$
 $\text{IsC} : L \rightarrow \text{tr}$

given by:-

$\text{IsN}(l) = \text{ISL}(\text{repL}(l))$
 $\text{IsC}(l) = \text{ISL}(\text{repL}(l)) \rightarrow \text{FF} \mid \text{TT}$

Furthermore, there are the usual selector functions:-

$\text{Hd} : L \rightarrow A$
 $\text{Tl} : L \rightarrow L$

where

$\text{Hd}(l) = \text{FST}(\text{OUTR}(\text{repL}(l)))$
 $\text{Tl}(l) = \text{SND}(\text{OUTR}(\text{repL}(l)))$

In general, there is one selector function for each factor of each summand in the domain equation.

Now, we note that elements in $T^{i+1}(\underline{\text{dot}})$ belong, by definition of T , to $1 + (A \# T^i(\underline{\text{dot}}))$. It is clear that $T^{i+1}(\underline{\text{dot}})$ consists roughly of elements corresponding to those given by ground "constructor" expressions (including l_A and l_L) of depth i . For example, $C(f_1, C(f_2, N))$ and $C(f_3, l_L)$ each correspond to elements of $T^3(\underline{\text{dot}})$ for any particular values of f_1, f_2 and f_3 from A .

Hence, L consists of all those elements given by "constructor" expressions of finite depth as well as any elements given as lubs of any proper ω -chains of such expressions. In this particular example, non-trivial limit points do exist. Let x be any defined element of A and consider the following sequence $\langle l_i \rangle_{i \in \omega}$ in L .

$l_0 = l_L$
 $l_{i+1} = C(x, l_i)$

Now, clearly $l_L \in C(x, l_L)$ and so by induction on i , $l_i \in l_{i+1}$. Note that $\partial(x, l_L) = \text{TT}$ (since x is defined), and so $C(x, l_L)$ is defined. Moreover, C is a composition of injective functions and so is also injective. So, in particular, $l_i \neq l_{i+1}$ for any $i \in \omega$. Hence, $\langle l_i \rangle_{i \in \omega}$ is a proper ω -chain in L , and so:-

$$\bigsqcup_{i \in \omega} l_i = l_x$$

is a proper "infinite" limit point. Moreover, notice that if the function $G: L \rightarrow L$ is given by $G(l) = \text{Cons}(x, l)$ then:-

$$\text{FIX}(G) = \bigsqcup_{i \in \omega} G^i(l_L) = \bigsqcup_{i \in \omega} l_i = l_x$$

Hence,

$$G(l_x) = C(x, l_x) = l_x$$

Finally, turning to the question of "structural induction" schemes, note that the usual list structural induction schema is valid:-

$$\frac{\vdash P[l_L] \quad \vdash P[N] \quad \vdash \forall l:L. P[l] \supset \forall x:A. P[C(x, l)]}{\vdash \forall l:L. P[l]}$$

where $P[l]$ is any admissible predicate depending upon $l:L$.

The validity of this schema for elements that are finitely expressible in terms of constructor functions follows by induction on the depth of "expressions". The admissibility of the predicate P then ensures validity for all proper limit points, also. Hence, the result follows since the "constructors" are continuous.

The schema above fails to be valid for non-well-founded solutions to the domain equation for lists, because the predicate P could then be untrue for some non-standard, or unreachable, point. However, a weaker case analysis schema will always be valid for every solution of the domain equation. The form of this rule is as above (with the corresponding types and constructors) but without any induction hypotheses. The validity of the cases schema follows from simple properties of the disjoint coalesced sum of domains, and the existence of discriminator functions. Consequently, there is no restriction upon meaning of the predicate P for this schema (i.e P does not have to be admissible).

In the general case, for any domain equation of appropriate form, the corresponding structural induction and case analysis schemes will be valid, by similar arguments to those sketched above.

2.1.5 Extended Domain Equations.

The theory of domain equations can be easily extended to cover the case where several domains are being defined in a mutually recursive fashion. The general situation can be stated as follows (where $n \geq 1$):-

$$\begin{aligned} D_1 &\cong T_1(D_1, D_2, \dots, D_n) \\ D_2 &\cong T_2(D_1, D_2, \dots, D_n) \\ &\vdots \\ D_n &\cong T_n(D_1, D_2, \dots, D_n) \end{aligned}$$

Such a system of equations is taken to define the initial, or "least", simultaneous fixed point within the product category $(\mathbf{CPO}^E)^n$ for suitable domain operators T_i . See [SmythPlotkin82] for further details.

Polymorphic, or parameterised, domain equations can also be dealt with theoretically. Informally, a "polymorphic" domain equation introduces a domain operator (written in postfix notation) that, when applied, gives a domain which is an initial fixed point of the instantiated domain equation. For example, consider the following:-

$$(\alpha)L \cong 1 + (\alpha \# (\alpha)L)$$

This defines a polymorphic "domain" of list; if D is any particular domain, then $(D)L$ gives a domain which satisfies the equation:-

$$(D)L \cong 1 + (D \# (D)L)$$

More generally, suppose that F is a domain operator of arity 1 which is defined via a domain equation. In general, the right hand side of this definition could depend upon both the parameter domain and the result. Hence, this is represented by a 2 argument type operator H used as follows:-

$$(\alpha)F \cong H(\alpha, (\alpha)F)$$

The domain operator F is then determined by putting

$$(D)F = R \text{ where } R = \mu X. T'(X) \text{ and } T'(X) = H(D, X)$$

Categorically, solving polymorphic domain equations corresponds to solving domain equations in an appropriate functor category. See [SmythPlotkin82] for further details.

2.1.6 Axiomatising Domain Equations in LCF.

Having informally sketched some of the theoretical background to the solution of domain equations, the question of how "least" solutions are formalised in LCF is addressed.

The property of being a solution of a domain equation is easily stated by introducing the appropriate isomorphism pair. As has been seen above, all appropriate constructor, selector and discriminator functions are definable using them (for an arbitrary solution).

However, the intended solution is any for which the appropriate structural induction rule is valid. Such induction rules, in their most general form, are schemas and cannot be stated directly (in LCF) as part of an axiomatisation.

In Edinburgh LCF, all induction rules have to be derived from the basic Computational Induction rule. So, any (finite) axiomatisation of the "least" solution of a domain equation has to provide enough information for the corresponding structural induction schema to be derivable from this basic one.

The standard method, originally due to Dana Scott, is to assert the "well-foundedness" of each element in the required domain. It strongly exploits LCF's higher type capabilities and makes essential use of least fixed points.

The idea is to introduce an auxiliary (second-order) function known as the "copy functional". As in the case of the isomorphisms, there is a naming convention for it. If A is the domain being defined, then the "copy functional for A " is named "copy A " and has type $:(A \rightarrow A) \rightarrow A \rightarrow A$.

The form of its definition depends, in a systematic way upon the form of the domain equation being used. Recall from Section 2.1.4 that the domain operator T is, in fact, an endofunctor of $\mathbf{CPO}^E$ and so possesses a morphism part (which we also name T). Now, assuming that $D = \mu X. T(X)$ with (fixed) isomorphisms $\text{abs}_D: T(D) \rightarrow D$

and $\text{repD}: D \rightarrow T(D)$, the "copy functional for D" can be defined as:-

$$\text{copyD} (f) (a) = (\text{absA} \circ T(f) \circ \text{repA}) (a)$$

Now, for definiteness, consider the domain equation for lists defined above in Section 2.1.3. The appropriate domain operator is $T(X) = 1 + (A \# X)$, with corresponding morphism part defined by:-

$$\begin{aligned} \vdash \forall f: X \rightarrow Y \quad tx: T(X). \\ T(f)(tx) = \\ \text{ISL}(tx) \Rightarrow (\text{INL} \circ \text{id}_1 \circ \text{OUTL})(tx) \\ \quad \mid (\text{INR} \circ (\lambda p. (\text{FST}(p), f(\text{SND}(p)))) \circ \text{OUTR})(tx) \end{aligned}$$

This simplifies down to:-

$$\begin{aligned} \vdash T(f)(tx) = \\ \text{ISL}(tx) \Rightarrow \text{INL } T \\ \quad \mid \text{INR}(\langle \text{FST}(\text{OUTR}(tx)), f(\text{SND}(\text{OUTR}(tx))) \rangle) \end{aligned}$$

So, from the general definition of copy functionals given above, the copy functional for domain L is:-

$$\begin{aligned} \text{copyL} (f) (l: L) \\ = \text{absL} (T(f)(\text{repA } l)) \\ = \text{IsN}(l) \Rightarrow N \mid C(\text{Hd}(l), f(\text{Tl}(l))) \end{aligned}$$

using the definitions of the discriminator and selector functions, "N", "Hd" and "Tl" given earlier.

Now, suppose that $F = \text{FIX copyL}$. Clearly, this satisfies the following equation:-

$$F(l) = N(l) \Rightarrow N \mid C(\text{Hd}(l), F(\text{Tl } l))$$

It can now be seen that F recursively breaks its argument down, and "reconstructs" it again. Now if l is finitely expressible in terms of "constructor" functions and constants, then, clearly, $F(l) = l$, by induction on the depth of l . On the other hand, if l is a proper limit point, then $\bigsqcup_{i \in \omega} l_i = l$ for some (proper) ω -chain $\langle l_i \rangle_{i \in \omega}$ of finitely-expressible elements. So, $F(l) = F(\bigsqcup_{i \in \omega} l_i) = \bigsqcup_{i \in \omega} F(l_i) = \bigsqcup_{i \in \omega} l_i = l$

Hence, we have shown that, since L consists entirely of the reachable, well founded elements, that $\text{FIX}(\text{copyL}) = F = \text{id}_L$. Therefore, to complete the axiomatisation of L, the definition of copyL is given in terms of the standard list manipulation functions

and the required well-foundedness property is stated as:-

$$\vdash \forall l:L. \text{FIX}(\text{copyL})(l) = l$$

Now, in general, given a domain equation (of appropriate form) the form of the corresponding "copy functional" can be easily expressed as a single lambda definition involving the constructor, discriminator and selector functions for the domain algebra being defined. The use of a single lambda-definition ensures that the proper care is taken when using strict (i.e. non-injective) "constructor" functions. The well-foundedness of the required domain is assured by using a fixed point axiom of the form above.

In section 5.1.1, an LCF package is discussed which automates the axiomatization technique given here. The axiomatisation contains sufficient explicit information that enables instances of the corresponding "structural induction" scheme to be systematically derived. A general purpose "structural induction" tactic generator is given in a second package which performs such a derivation to construct the appropriate induction tactic.

2.1.7 Limit Points and Domain Equations.

By definition, proper limit points are elements expressible as the least upper bound of some proper ω -chain. Hence, the existence of such limit points (in initial solutions of domain equations) directly depends upon the existence of defined, finite, partial elements. Such partial elements arise when using constructor functions that are not strict in some argument. For example, $l \sqsubseteq \text{Cons}(x, l_l)$ for any defined $x \in F$, as seen in Section 2.1.3. Significantly, non-strictness here occurs in an argument whose type is equal to that being defined. This permits partial elements to be nested to arbitrary depth, giving rise to proper ω -chains and so proper limit points.

The extent to which a constructor function is non-strict depends upon the form of the summand in the domain equation to which it corresponds. Generally speaking, non-strict arguments arise from the use of full Cartesian product or "lifting" domain operators in the expression of the summand.

2.1.8 Polymorphism.

The idea of polymorphism is a natural one; a "polymorphic type" corresponds to a (large) function mapping domains into domains. Similarly, a "polymorphic value" is a family of values, one for each instance of its polymorphic type. Note that this form of type parameterisation is taken with respect to the class of all domains, and not just for those domains which can be explicitly named.

Hence, theorems concerning polymorphic values (i.e. polymorphic theorems) remain valid no matter which specific types are explicitly named; the validity of a theorem is independent of the population of namable domains available at any particular stage. Therefore, a polymorphic theorem may be regarded as a family of theorems, all asserting the same property for every possible choice of domain.

However, there are some subtle points relating to polymorphism and type operators in general. Consider the three rather similar polymorphic theories, given in Figure 2.3 overleaf. In each case, a constant $\square$ and binary operation $\oplus$ is introduced. Additionally, three assertions are then stated which say that $\oplus$ is left strict and that $\square$ is both a left and right identity element for $\oplus$. The sole difference between the presentation of the theories lie in the degree of type parameterisation they contain.

The first theory has no type parameterisation, and there are many possible choices of specific domain M and corresponding operators for which the given axioms are valid.

For the second theory, there is some type parameterisation in the form of a domain function N (of arity 1), and this also possesses many (polymorphic) models. That is, there are choices of the domain function N such that, for each domain D , there is an assignment of constants $\square : N(D)$ and binary operation $\oplus : N(D)^2 \rightarrow N(D)$ which makes the axioms valid. For example, consider any model of the first theory, and then a choice of constant $\square : D$ and (continuous) binary operation $\oplus : D^2 \rightarrow D$ such that all of the axioms are satisfied.

However, when we come to the third theory, which has the most general form of type parameterisation, we find that it has NO

(a) constant type :M

$$\begin{aligned} \perp & :M \\ \otimes & :M \# M \rightarrow M \end{aligned}$$

$$\begin{aligned} \text{'a1'} & \vdash \forall x:M. \perp \otimes x \equiv \perp \\ \text{'a2'} & \vdash \forall x:M. \perp \otimes x \equiv x \\ \text{'a3'} & \vdash \forall x:M. x \otimes \perp \equiv x \end{aligned}$$

(b) type operator :N (of arity 1)

$$\begin{aligned} \perp & :N(\alpha) \\ \otimes & :N(\alpha) \# N(\alpha) \rightarrow N(\alpha) \end{aligned}$$

$$\begin{aligned} \text{'b1'} & \vdash \forall x:N(\alpha). \perp \otimes x \equiv \perp \\ \text{'b2'} & \vdash \forall x:N(\alpha). \perp \otimes x \equiv x \\ \text{'b3'} & \vdash \forall x:N(\alpha). x \otimes \perp \equiv x \end{aligned}$$

(c) - no type or type operator -

$$\begin{aligned} \perp & : \alpha \\ \otimes & : (\alpha \# \alpha) \rightarrow \alpha \end{aligned}$$

$$\begin{aligned} \text{'c1'} & \vdash \forall x:\alpha. \perp \otimes x \equiv \perp \\ \text{'c2'} & \vdash \forall x:\alpha. \perp \otimes x \equiv x \\ \text{'c3'} & \vdash \forall x:\alpha. x \otimes \perp \equiv x \end{aligned}$$

FIGURE 2.3

models at all! This particular theory asserts that for every domain, D , there is a choice of constant $\perp:D$ and (continuous) binary operation $\otimes:D^2 \rightarrow D$ which satisfy the axioms. However, the following theorem shows that this is impossible:-

Theorem 2.4

There exists a non-trivial domain A such that if $\otimes : A^2 \rightarrow A$ is a continuous function such that, for some $\perp \in A$

- (1) $\forall x:A. \perp \otimes x \equiv x$
- (2) $\forall x:A. x \otimes \perp \equiv x$

then $\perp \equiv 1$, the least element of A .

Proof

Let $A = \{ \emptyset \} \cup \{ s \subseteq \omega \mid s \text{ is infinite} \}$. Clearly A is a domain, whose least element is $\emptyset$, when ordered by subset inclusion.

The argument will now proceed by using reductio ad absurdum and so suppose that $\square \neq 1$.

Now, since $\otimes$ is continuous and therefore monotonic, we have:-

$$\forall x y:A. (x \subseteq \square) \supset (x \otimes y) \subseteq (\square \otimes y) = y$$

and similarly:-

$$\forall x y:A. (y \subseteq \square) \supset (x \otimes y) \subseteq (x \otimes \square) = x$$

Therefore,

$$(x \subseteq \square) \ \& \ (y \subseteq \square) \supset (x \otimes y \subseteq x) \ \& \ (x \otimes y \subseteq y)$$

Now, since $\square \neq \emptyset$, there exists a pair of (non-trivial) ω -chains

$\langle p_i \rangle_{i \in \omega}$ and $\langle q_i \rangle_{i \in \omega}$ such that:-

$$(1) \bigsqcup_{i \in \omega} p_i = \square = \bigsqcup_{i \in \omega} q_i$$

$$(2) \forall i \in \omega. (p_i \cap q_i) \text{ is a finite subset of } \omega$$

(where p_i and q_i are taken as subsets of ω).

This is because, if $\square \neq \emptyset$ then $\square$ is some infinite subset of ω .

Assume that $\square = \{a_0, a_1, a_2, a_3, \dots\}$ where $a_i \in \omega$. A suitable choice of the ω -chains $\langle p_i \rangle_{i \in \omega}$ and $\langle q_i \rangle_{i \in \omega}$ having the desired properties is now given, as follows:-

$$\text{Put } p_0 = \{ a_{2k} \mid k \in \omega \} \text{ and } q_0 = \{ a_{2k+1} \mid k \in \omega \}$$

and then by induction on $i \in \omega$, we define:-

$$\begin{aligned} p_{i+1} &= p_i \cup \{ a_{2i+1} \} \\ q_{i+1} &= q_i \cup \{ a_{2i} \} \end{aligned}$$

Clearly, for each i , $p_i \cap q_i = \{ a_0, a_1, \dots, a_{2i-1} \} \notin A$, since it is a finite set, and moreover:

$$\bigsqcup_{i \in \omega} p_i = \{ a_0, a_1, a_2, \dots \} = \square = \bigsqcup_{i \in \omega} q_i$$

Now, consider the value of $(p_i \otimes q_i) \in A$; by our previous

observations $(p_i \otimes q_i) \subseteq p_i$ and also $(p_i \otimes q_i) \subseteq q_i$. Therefore,

$(p_i \otimes q_i) \in A$ and $(p_i \cap q_i)$ is a finite set. Hence we can only

conclude that $(p_i \otimes q_i) = \emptyset$, for each $i \in \omega$. But then, by continuity of $\otimes$, we have that:-

$$\begin{aligned}
\emptyset &= \bigsqcup_{i \in \omega} (p_i \otimes q_i) = (\bigsqcup_{i \in \omega} p_i) \otimes (\bigsqcup_{i \in \omega} q_i) \\
&= \square \otimes \square \\
&= \square
\end{aligned}$$

This directly contradicts the assumption that $\square \neq \emptyset$, and completes this proof.

QED

Corollary 2.5

Since A is non-trivial, axiom 'c1' cannot be satisfied.

Proof

Suppose that $\square : A$ and $\otimes : A^2 \rightarrow A$ are such that all the axioms 'c1', 'c2' and 'c3' are satisfied. So, by the above theorem we have that $\square \equiv \emptyset \equiv 1$. So, by 'c1', $1 \equiv 1 \otimes x \equiv \emptyset \otimes x \equiv x$, for any x showing that A is trivial, which is clearly false, and this gives the result.

QED

2.1.9 Computational intuition underlying domain theory.

The central aspect of domain theory, or Scott's theory of computation, is the role played by partial orderings of various kinds. It is clear that only certain sorts of partial orderings are useful, applicable and support "computational intuition". But which sort of partial orderings? Is it merely some technical device which just happens to work for the construction of a particular kind of (applied) mathematical model? Or is there a rational intuition concerning computation that motivates the choice of partial ordering? In short, what is the insight into the nature of computation which is given by Scott's concept of approximation?

Much has been written concerning domain theory and its applications (for example [ScottStrachey71], [Scott76], [Scott82], [Stoy77], [Milner72c], [Bird76], [de Bakker80] to mention a few) but the underlying computational intuition has not always been stated.

The issue lies not with the validity of the mathematics developed, but in the choice of mathematics and, in particular, with its "external meaning" in the context of its application to

computation. The starting point lies in the seminal paper by Scott and Strachey where they say:-

"An intuitive way of reading the relationship $x \sqsubseteq y$ is to say that x approximates y The statement $x \sqsubseteq y$ does not mean that x is very near y , but rather that x is a poorer version of y , that x is only partially specified and that it can be improved to y without changing any of the definite features of x ."

From "Toward a Mathematical Semantics for Computer Languages" by Dana Scott and Christopher Strachey, Page 32, Proc. Symposium on Computers and Automata, Polytechnic Inst. of Brooklyn, Vol. 21, 1971.

So, according to the above, the partial ordering says how some values can be thought to be more specific refinements of other values. Others, such as [Stoy77] p. 80, have used the metaphor that the ordering represents increasing "information"; $x \sqsubseteq y$ means that all we know about x is certainly true of y and that more could be known about the properties of y than is known about the properties, or characteristics, of x . This idea of "information-ordering" has been further extended with the development of Scott's theory of "information systems" in [Scott82].

The intuition discussed here is similar to, and was inspired by, the exposition given within [Wadsworth78]. For us, the partial ordering is concerned with the observable characteristics of values. The statement that $x \sqsubseteq y$ says that the observable features of x are included amongst those for value y . This means that at least as much detailed "structure" can be seen in y as can be seen in x .

This does not necessarily imply a concept of "time". Whenever $x \sqsubseteq y$ holds, the value x is not necessarily "calculated" before or even during the "calculation" of y . It is understood that the relationship $\sqsubseteq$ on a domain has the same mathematical status as any other binary relation. Its significance is that it gives one mathematical model of one notion of observable approximation.

The least element, $\perp$, has a simple, straight-forward meaning; it is simply an object about which no definite structure could ever be observed, except perhaps that it is an element of D ! This fits

in with the operational idea that $\perp_D$ stands for the "result" of a "non-terminating" computation. What this roughly means is that, from the outside, no definite output is observed from a process that silently acts forever.

The monotonicity of "interesting" functions is now easily argued. Suppose that $x \sqsubseteq y$ and that $f : A \rightarrow B$ is a (possibly computable) function of interest. Since $x \sqsubseteq y$, the value of f at y (i.e. $f(y)$) depends on at least as much structure as can be observed from inspecting x as can be observed from inspecting y . Hence, the observable characteristics of $f(x)$ must lie among those for $f(y)$, which says that $f(x) \sqsubseteq f(y)$.

The motivation for the continuity of "interesting" functions comes from at least two sources. The first source is the essentially finitary character of computation - each observable characteristic of the data delivered as output from a (computable) function could only have been determined by analysis of some finite collection of observable characteristics of the argument. Hence, the output can only contain all those observable characteristics which are produced by applying the function to any suitable approximation of the argument.

The second motivation is related to the technical requirements for a "least fixed point" semantics for recursively defined functions. The use of "fixed points" is mathematically very natural, in that this gives a general formulation for many mathematical existence problems.

Suppose, in general, that $f : A \rightarrow B$ such that $f(x) \sqsubseteq e[f, x]$ where $e[f, x]$ is some lambda expression possibly depending upon f and x . Hence, $f \sqsubseteq \lambda x:A. e[f, x]$. Now, consider the functional $F : (A \rightarrow B) \rightarrow (A \rightarrow B)$ such that $F(g) \sqsubseteq \lambda x:A. e[g, x]$ which, by lambda abstraction, gives $F \sqsubseteq \lambda g. \lambda x. e[g, x]$. Clearly, we now have that $F(f)(a) \sqsubseteq e[f, a] \sqsubseteq f(a)$ for any $a:A$, and so, by extensionality, $F(f) = f$. Therefore, f is a fixed point of the functional F .

This immediately raises two issues - when does any fixed point of F exist, and when they do, which one is intended? The answers to both of these questions are motivated by informal computational considerations. In some sense, $F(g)(a)$ behaves rather like f

applied to argument a - except that where the function f would have been used, the function g is used instead. Clearly $F(1)(a) \sqsubseteq f(a)$ since $F(1)(a)$ gives, as it were, the same observable output as one "iteration" of the definition of f . Similarly, $F(F^{n-1}(1))(a) \sqsubseteq F^n(1)(a) \sqsubseteq f(a)$, for any $n \in \omega$. Now, since each observable characteristic of the value $f(a)$ is computed, it is reasonable to suppose that any such characteristic is contributed at some finite stage. This implies that any output value is always completely given within ω iterations; that is, for any $a \in A$:-

$$f(a) = \bigsqcup_{i \in \omega} F^n(1)(a)$$

and so, by lambda abstraction, $f = \bigsqcup_{i \in \omega} F^n(1)$. However, $F(f) = f$,

and, so by substituting both sides for f , we get that:-

$$F(\bigsqcup_{n \in \omega} F^n(1)) = \bigsqcup_{n \in \omega} F^n(1)$$

But now, by monotonicity of the functional F , we have that

$$\bigsqcup_{n \in \omega} F^n(1) = \bigsqcup_{n \in \omega} F^{n+1}(1)$$

and so we get that:-

$$F(\bigsqcup_{n \in \omega} F^n(1)) = \bigsqcup_{n \in \omega} F^{n+1}(1)$$

This is an instance, for the functional F and at argument 1 , of the continuity property for functions. Of course, Scott's least fixed point theorem then shows that the continuous function f as defined above exists for continuous F and that f is its least fixed point. Hence, continuity ensures that the semantics of recursive functions corresponds to the operational intuition of unwinding or unfolding a recursive definition to give an infinite computation tree.

The interest in continuous functions therefore stems from the least fixed point theorem and its characterisation of the least fixed point using finite iterates.

2.2 Basic Category Theory

This section is a review of basic concepts from the general theory of categories. Much of this material can be found in some form in standard algebra texts such as [MB].

A category C is an algebraic structure consisting of a class of objects, $|C|$ and a class of morphisms, $\vec{C}$ with the following (partial) functions:-

$\text{dom} - : \vec{C} \rightarrow |C|$ — the domain function

$\text{ran} - : \vec{C} \rightarrow |C|$ — the range function

$\text{id} - : |C| \rightarrow \vec{C}$ — the "identity morphism" function

$(- \circ -) : \vec{C}^2 \rightarrow \vec{C}$ — the (partial) composition function

The notation " $f: A \rightarrow B$ " means that f is a morphism of C

(i.e. $f \in \vec{C}$), such that $\text{dom}(f) = A$ and $\text{ran}(f) = B$, where $A, B \in |C|$.

The composition function is partial to the extent that if

$f: A \rightarrow B$ and $g: B \rightarrow C$ then $(g \circ f)$ is defined and $(g \circ f): A \rightarrow C$. For each $A \in |C|$, there is the identity morphism $\text{id}_A: A \rightarrow A$. The composition function is associative (when defined) with left and right identities given by id_A for appropriate objects A .

A functor $F: A \rightarrow B$ (i.e from a category A to category B) is a morphism between categories in the sense that it is given by a pair of functions that preserve the basic categorical structure. Formally:-

$$F = \langle |F|: |A| \rightarrow |B|, \vec{F}: \vec{A} \rightarrow \vec{B} \rangle$$

such that:-

(a) for every A -morphism $g: a \rightarrow b$, $\vec{F}(g)$ is a B -morphism

$$\text{where } \vec{F}(g): |F|(a) \rightarrow |F|(b)$$

(b) if $g: a \rightarrow b$ and $h: b \rightarrow c$ are A -morphisms then

$$\vec{F}(h \circ g) = \vec{F}(h) \circ \vec{F}(g)$$

(c) for every A -object a ,

$$\vec{F}(\text{id}_a) = \text{id}_{|F|(a)} : |F|(a) \rightarrow |F|(a)$$

It is traditional to call both $|F|$ and $\vec{F}$ by the same name, F .

2.2.1 Concrete Categories.

A category C is concrete with respect to a base category B iff there exists a functor $F : C \rightarrow B$ such that for every C -morphism $g : a \rightarrow b$, $F(g) = g : F(a) \rightarrow F(b)$ i.e. every C -morphism can be viewed as a B -morphism of a certain kind.

Many examples of concrete categories arise by taking the objects of the base category B and endowing them with extra "structure" to give the C -objects. The C -morphisms are then these B -morphisms which "preserve", in a well-defined way, this additional structure. The functor F mentioned above is often known as a "forgetful functor" since it usually maps C -objects back to B -objects by "forgetting" the additional "structure".

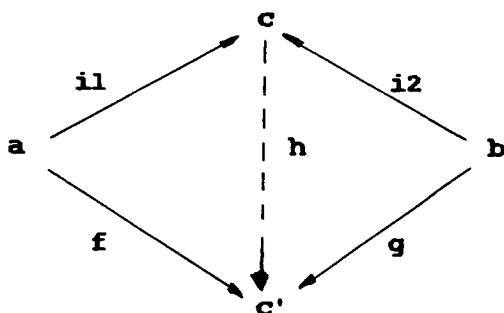
A relevant example of a concrete category is **POSET**, the category of all (small) partially-ordered sets whose morphisms are the order-preserving (or monotone) functions. Each poset is a relational structure of the form $\langle A; \leq_A \rangle$ where $\leq_A$ is the partial ordering relation. An appropriate forgetful functor is $F : \mathbf{POSET} \Rightarrow \mathbf{SET}$, where $F(\langle A; \leq_A \rangle) = A$ and for monotonic functions $F(f : \langle A; \leq_A \rangle \rightarrow \langle B; \leq_B \rangle) = f : A \rightarrow B$. In other words, each poset corresponds to its "carrier" set and each **POSET** morphism is clearly a function.

When the base category is not mentioned, it is traditional to take it to be **SET**, a (large) category consisting of (small) sets with total functions for morphisms. However, the discussion contained herein is principally concerned with Scott domains. Hence, our convention is to assume the category **CPO** instead whenever the base category is omitted.

2.2.2 Commuting Diagrams and Universal constructions.

A diagram (for a category C) is a (directed) graph whose nodes correspond to C -objects and whose edges correspond to C -morphisms; the source and target nodes of edges correspond to the domain and range objects of the corresponding C -morphisms. Diagrams are said to commute whenever every pair of morphisms, represented by paths (of length ≥ 1) with common beginning and ending points, are equal. Many categorical concepts can be concisely formulated by stating that certain diagrams commute.

For example, a general notion of categorical sum or co-product, can be formulated thus:- let a, b be C -objects. A sum of a and b is a C -object, c , with morphisms $i_1 : a \rightarrow c$ and $i_2 : b \rightarrow c$ such that for any c' with morphisms $f : a \rightarrow c'$ and $g : b \rightarrow c'$, there exists a unique morphism $h : c \rightarrow c'$ such that the following diagram commutes:-



This is an example of a universal definition. The morphism h is called the fill-in (or universal) morphism, and the most significant aspect of such universal definitions is the existence and the uniqueness of this morphism. As observed in [Burstall80], such morphisms frequently correspond to computationally useful constructions. A very simple example of this is to observe that the usual set-theoretical disjoint union is a categorical sum in the category SET .

Given two sets A, B , their disjoint union is $A + B = \{0\} \times A \cup \{1\} \times B$, with injection morphisms $i_1(a) = (0, a)$ and $i_2(b) = (1, b)$. Suppose that $f : A \rightarrow C$ and $g : B \rightarrow C$. The fill in morphism $h : (A + B) \rightarrow C$ has the property that $h((0, a)) = f(a)$ and $h((1, b)) = g(b)$, corresponding to the traditional concept of "case analysis" in programming languages.

A general characteristic of universal definitions is that the specified structure will be unique up to isomorphism within the category C . For this reason, it is permissible to speak of, for example, "the categorical sum" rather than "a categorical sum". Also, the existence of such universal constructions generally depends upon the internal characteristics of the category - for example, a given category may or may not possess co-products for every pair of objects. When it does, the category is said to possess co-products. See [Rydeheard81] for a further exploration of these ideas in a computational framework.

2.2.3 Free and Initial objects.

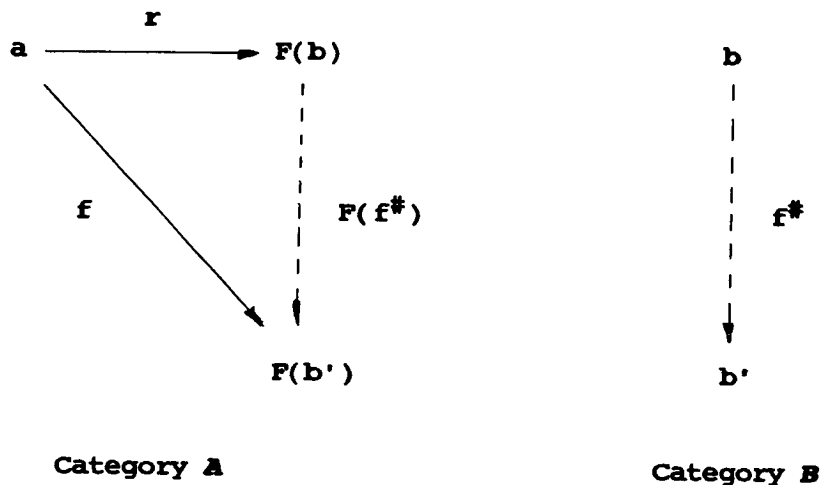
In traditional algebra, there is the concept of a freely-generated algebraic structure with respect to a collection of "generators" (or "variables"). For example, in the mathematical theory of groups, the free group on a collection of generators (with no "relations") consists of all formal expressions built from the given generator symbols (with the binary operation and inverse) reduced with respect to the laws for a group. If further "relations" (ie. extra equations between the generators) are given then these are also taken into account.

The basic idea is to construct the appropriate collection of terms and to specify the least congruence consistent with the desired (equationally specified) properties. The required "free algebra" is then formed by taking the corresponding quotient set endowed with the "natural" operator structure. This is widely known as the "free term algebra" construction or the "Herbrand model" amongst mathematical logicians. A lucid, introductory discussion of the concepts of free and initial algebra may be found in [BurstallGoguen82].

It turns out that the "free algebra" can be specified abstractly by the use of categorical methods without recourse to the explicit construction sketched out above. Moreover, the categorical definition applies to general categories, not just to categories of algebras. Hence, the "free algebra" idea is an instance of the much wider categorical concept of "free object"

(upon a given object).

The basic set-up is as follows:- let $\mathcal{A}$ and $\mathcal{B}$ be categories with a "forgetful functor" $F:\mathcal{B} \rightarrow \mathcal{A}$. Let a be an $\mathcal{A}$ -object. A free-object upon the object a is a $\mathcal{B}$ -object b and a morphism (called the "inclusion of generators") $r : a \rightarrow F(b)$ such that the following holds. For any $\mathcal{B}$ -object b' and $\mathcal{A}$ -morphism (or valuation) $f : a \rightarrow F(b')$, there exists a unique $\mathcal{B}$ -morphism $f^* : b \rightarrow b'$ such that $f = F(f^*) \circ r$. This can be described succinctly in terms of the following commuting diagram:-



For example, suppose that $\mathcal{A}$ is **SET** and $\mathcal{B}$ is **GROUP**, the category of all (small) groups. The object " a " above then corresponds to some given collection of generators, the object b is the free group generated by a and $F(b)$ is the set of elements comprising the group b . Now, let b' be any group and suppose that f is any valuation function from the generator set, a , to $F(b')$, the set of elements comprising the group b' . Now, by the freeness property, this valuation f can be extended uniquely to a group homomorphism $f^* : b \rightarrow b'$ that behaves the same as f on group elements corresponding to the generators. In some sense, f^* extends the valuation, f , from the generators to an arbitrary (finite) expression involving the generators and group operators.

The concept of free object is, in fact, an instance of a still more general categorical concept of adjunctions (See [MB], p517). For our purposes, the definition given above is sufficiently

abstract for deriving the appropriate correctness criteria for our case studies. Our main application of these concepts is to give a definition principle for basic data types specified algebraically. This is further discussed in Section 2.3.1 below.

The above definition can be regarded as a model-theoretic constraint. One way of giving this proof-theoretic substance is to note that, in considering categories of algebras, the "free term algebra" is the free object (with the obvious inclusion of generators). This will satisfy the appropriate structural induction principle (which must also include a base case for those "terms" corresponding to the generators).

An initial object, i , for a category C is such that for every C object a , there exists a unique morphism $h : i \rightarrow a$. This morphism is called the "initiality" morphism for the initial object i .

For example, in CPO_1 , where all morphisms are strict, an initial object is the one-point domain dot, where the "initiality" morphism is the obvious strict function. However, there is no initial object in the category CPO , since there are always at least two distinct continuous functions from a cpo D into a (non-trivial) cpo D' . The application of initiality to give basic domains of data along with their operations (i.e. initial algebras) is discussed below.

2.3 Continuous Algebras.

A (multi-sorted) algebra intuitively consists of a specified collection of sets (called the carriers) and a collection of basic functions (called the operators) of specified type which satisfy certain axiomatic properties (often expressed equationally). Much work has been done on the algebraic approach to specification (e.g. [BurstallLandin69], [BurstallGoguen80a], [ADJ77], [Guttag75], [BroyWirsing81]) and here we review some of the basic definitions and assumptions needed later on.

When working with PPLAMBDA, all axiomatic descriptions are taken in the context of complete partial orders with Scott continuous functions (i.e. with respect to the (concrete) category CPO). This context is therefore specified in the following

definitions, although they could be based upon any concrete category C (with respect to SET), such as CPO_1 . So, in the terminology of [ADJ77], the algebraic structures taken here are the continuous algebras.

In formalising algebraic structures, the notion of signature plays a major role. The signature is a syntactic device for explicitly stating the names of sorts and associating the names of operators with their argument and result types. Formally, a signature Σ is a pair $(S, \tau: Op \rightarrow (S^* \times S))$ where S is a set of sort names, Op is a set of operator names and the "typing" function, τ , assigns each operator name to its type. This is represented by a pair whose first component is a tuple representing the individual components of the operators' domain of definition and the other component represents its codomain.

Now, given a sort-name, s , from Σ , and a (possibly countable) collection of "variables" V , the well-typed continuous Σ -terms of sort s over V can be formed by mutual recursion and is usually denoted by $CT_{\Sigma,s}(V)$. A Σ -algebra presentation is a pair $\langle \Sigma, Ax \rangle$ consisting of the signature Σ , and Ax , a collection of logical sentences (involving Σ terms) that represent the axiomatic properties to be satisfied. Such axioms, (or basic observations), usually consist of universally-quantified equalities between pairs of well-typed Σ -terms of equal type - i.e. equations. A (Scott-continuous) Σ -algebra is an association of a domain (called the carriers) with each sort name and a corresponding type-preserving association of (continuous) functions with operator names. If A is a Σ -algebra, then s^A stands for the domain associated with the Σ -sort name $s \in S$ by the algebra A . Similarly, σ^A stands for the operation associated with the Σ operator name $\sigma \in Op$ by the algebra A . Using this basic notation for interpretations, the type-preservation property of Σ -algebras can be stated as:-

$$\forall \sigma \in Op. \sigma^A \in [(s_1^A \# s_2^A \# \dots \# s_k^A) \rightarrow s^A]$$

$$\text{where } \tau(\sigma) = ((s_1, s_2, \dots, s_k), s) \quad (k > 0)$$

When $k = 0$ then $\tau(\sigma) = ((), s)$ and $\sigma^A \in s^A$.

Suppose that $P = \langle \mathcal{L}, Ax \rangle$ is a $\mathcal{L}$ -algebra presentation. A $\mathcal{L}$ -algebra is said to satisfy P (and then called a $\mathcal{L}$ - P algebra) iff every sentence in Ax is true under the interpretation specified by A . Note that every $\mathcal{L}$ algebra satisfies the presentation $\langle \mathcal{L}, \emptyset \rangle$ (i.e. where there are no axioms to constrain the interpretation). This is called the anarchic presentation, since there are no "laws" that are required to hold.

Suppose that A and B are $\mathcal{L}$ -algebras. A $\mathcal{L}$ -algebra homomorphism $\underline{h}$ from A to B is an S -indexed family of continuous morphisms $\langle h_s : s^A \rightarrow s^B \rangle_{s \in S}$ such that, for every operator symbol $\sigma \in Op$:-

$$\begin{array}{ccc}
 s_1^A \# s_2^A \# \dots \# s_k^A & \xrightarrow{\sigma^A} & s^A \\
 \downarrow h_{s_1} \# h_{s_2} \# \dots \# h_{s_k} & & \downarrow h_s \\
 s_1^B \# s_2^B \# \dots \# s_k^B & \xrightarrow{\sigma^B} & s^B
 \end{array}$$

where $\tau(\sigma) = ((s_1, s_2, \dots, s_k), s)$ ($k > 0$)

When $k = 0$ then $\tau(\sigma) = ((), s)$ and $h_s(\sigma^A ()) = \sigma^B ()$.

and, for $f: D_1 \rightarrow D_2$, $g: D_3 \rightarrow D_4$, we have $(f \# g): (D_1 \# D_3) \rightarrow (D_2 \# D_4)$ such that $(f \# g)(a, b) = (f(a), g(b))$.

Moreover, if both $\mathcal{L}$ algebras A , B satisfy the presentation $P = \langle \mathcal{L}, Ax \rangle$ then $\underline{h}$ is said to be a $\mathcal{L}$ - P homomorphism. The class of all $\mathcal{L}$ - P algebras can be given a natural and straightforward categorical structure with the above notion of homomorphism (i.e composition is pairwise functional composition). This category is denoted by $ALG_{\mathcal{L}}(P)$.

Suppose that μ is a sentence of the logical calculus which only involves $\mathcal{L}$ -terms. Such sentences are called $\mathcal{L}$ -sentences. Any $\mathcal{L}$ -algebra A then fully determines the truth value of such sentences. So, for any $\mathcal{L}$ -sentence μ , μ satisfies A iff μ is true under the interpretation given by A . This is stated symbolically by writing $A \models \mu$. If a $\mathcal{L}$ -sentence μ satisfies every $\mathcal{L}$ -algebra then it is said to be valid and this is expressed by writing $\models \mu$.

Given any $\mathcal{L}$ -algebra, A , consider the set, T , of all $\mathcal{L}$ -sentences that satisfy A . Such a set is maximally consistent in so much as that any sentence $\mu \notin T$ is not true in A . More generally, if R is any sub-class of $\mathcal{L}$ -algebras, the collection, T , of all $\mathcal{L}$ -sentences

which satisfy every Σ -algebra in R , has the property that if $\mu \not\models T$ then $\exists$ a Σ -algebra A in R such that μ is not true in A (i.e. $A \not\models \mu$). Such (complete) sets of Σ -sentences are called Σ -theories. Given a Σ -algebra presentation $P = \langle \Sigma, Ax \rangle$, this gives rise to a Σ -theory in a natural way by first taking the class $ALG_{\Sigma}(P)$ and then forming the corresponding Σ -theory of these algebras.

This correspondence between Σ -theories and their Σ -algebras is a simple example of a Galois connection. This is further discussed in, for example, [BurstallGoguen80b] and [Cohn65].

2.3.1 Free and Initial Σ -algebras.

Let P be the anarchic Σ -algebra presentation. Then the category $ALG_{\Sigma}(P)$ has a free (continuous) Σ -algebra over a given domain of generators, D . This may be constructed by following the standard "free-term-algebra" construction to give the collection of finite terms of each sort in Σ , with (typed) "generators" from D . These collections are then partially-ordered in the obvious way i.e. if t_1, t_2 , are two terms of the same sort, s , then $t_1 \sqsubseteq_s t_2$ iff t_2 only differs from t_1 at occurrences of "1" or by generator elements from D , ordered appropriately. These posets are "completed" to give domains, using the lub-preserving continuous completion as described in [CourcelleRaoult80]. If D is a flat domain then the simpler ideal-completion due to McNeille (see [Birkhoff67]) suffices. The appropriate Σ -algebra is obtained by defining the operators in the usual "syntactic" way for finite terms and then extending the operators to limit points by the continuity condition.

The initial (continuous) Σ -algebra can be obtained by taking the free Σ -algebra given above, over the trivial generator domain, dot. Intuitively, the initial Σ -algebra consists of values corresponding to (ground) Σ -terms, possible involving 1. The unique Σ -homomorphism (i.e. the "initiality" morphism) from the initial Σ -algebra to a given Σ -algebra A then corresponds to the "evaluation" of Σ -terms with respect to the interpretation defined by A . Note that this unique morphism is precisely determined by "structural induction" over the Σ -terms and the interpretation of

the operators given by A.

2.3.2 Anarchic Σ -algebras and domain equations.

There is a simple syntactical characterisation of anarchic free and initial Σ algebras over CPO in terms of the solutions of certain domain equations. Given a signature $\Sigma = \langle S, \tau \rangle$, a (mutually-recursive) system of domain equations is constructed, one equation for each sort in S . The least solution of this system then gives both the carrier and the operators of the initial Σ -algebra, i.e. the initial Σ -algebra itself.

The basic idea is to ensure that there is a unique representative of each finite Σ -term within each domain (since there are no equations to identify terms). The proper "infinite" terms are then taken care of as a result of taking the least solution. The operators of the initial Σ -algebra will then correspond to the constructor functions derived from each summand of the domain equations.

More precisely, suppose that $\Sigma = \langle S, \tau \rangle$ where $\tau: \text{Op} \rightarrow (S^* \times S)$. Suppose that $s \in S$ and consider $C_s = \{ \sigma \in \text{Op} \mid \pi_2(\tau(\sigma)) = s \}$ i.e. C_s consists of all those operator names whose codomain type is equal to s . The required domain equation for sort s is formed by including a (lifted) summand for each element of C_s that consists of the (full) Cartesian product of each component sort of the codomain type (or dot if it is empty). Hence, the domain equation for sort s has the form:-

$$s \cong C1_1 + C2_1 + \dots + Cn_1$$

where $Ci = (s_1 \# s_2 \# \dots \# sk_i)$

and where $\tau(\sigma_i) = ((s_1, s_2, \dots, sk_i), s)$, for each $\sigma_i \in C_s$. (taking the empty product to be dot, as required).

This construction gives the anarchic initial Σ -algebra (relative to CPO). To obtain an anarchic free Σ -algebra over a domain of generators G , include in each domain equation, for each sort, an extra (non-lifted) summand for G . Hence the form for each domain equation is now:-

$$s \cong G + C1_1 + C2_1 + \dots + Cn_1$$

where the C_i 's are as before.

Note that by substituting dot for G , the initial Σ -algebra is again obtained. The idea is to introduce values in each sort to stand for the "generators" given by G . Conversely, each generator can give rise to a value in each sort, via the corresponding constructor.

Similiar constructions to these also give the appropriate anarchic initial and free Σ -algebras in CPO_1 (i.e. with strict constructors) by using the smash product instead of full Cartesian product and omitting the domain lifting operator $-_1$ for product (i.e. non dot) summands.

2.3.3 The initiality functional.

It is illustrated how the "initiality" morphism, based upon a signature Σ , gives an analogous functional to the "primitive recursion" iteration functional over the natural numbers. This is one idea forming a principal theme of the pioneering paper of [BurstallLandin69]. There it is shown how an easily defined second-order functional, called Extend, produces an evaluation function for classes of algebraic expressions, given an interpretation of the particular operators. This general technique is applied in the case studies in Chapters 3 and 4 to formalise simple freeness criteria in PPLAMBDA of a number of very simple algebraic data types.

For example, consider the following simple signature, Σ , with a single sort, Nm , and operators:-

$Z : Nm$
 $S : Nm \rightarrow Nm$

The anarchic initial Σ -algebra (with respect to CPO) is given by the least solution of the domain equation:-

$$Nm \approx \underline{dot}_1 + Nm_1$$

Now any Σ -algebra consists of a domain D , constant $z : D$ and operation $sc : [D \rightarrow D]$. The "initiality" morphism from the initial Σ -algebra, Nm , to D depends not only upon the carrier, but also upon the assignment of operators and constants. This dependency

can be easily formalised by giving a polymorphic, second-order functional that, when instantiated, gives the appropriate "initiality" morphism. The polymorphic type of this morphism is used to determine the dependency on the carrier and the (possibly functional) parameters describing the operations of the $\mathcal{L}$ -algebra. This "initiality" functional, $\text{InitNm}:(D \# [D \rightarrow D]) \rightarrow [Nm \rightarrow D]$ is the least continuous functional satisfying the following (homomorphism) properties:-

$$\begin{aligned} \vdash \text{InitNm}(z, \text{sc})(1_{Nm}) &= 1_D \\ \vdash \text{InitNm}(z, \text{sc})(Z) &= z \\ \vdash \text{InitNm}(z, \text{sc})(S(n)) &= \text{sc}(\text{InitNm}(z, \text{sc})(n)) \end{aligned}$$

This functional can be easily given a standard recursive definition:-

$$\text{InitNm}(z, \text{sc}) \equiv \text{FIX}(\lambda f. \lambda n. \text{IsZ}(n) \Rightarrow z \mid \text{sc}(f(P(n))))$$

where the function $\text{IsZ} : Nm \rightarrow \text{tr}$ is the discriminator function for the constant Z , and $P : Nm \rightarrow Nm$ is the corresponding selector function for the constructor function $S : Nm \rightarrow Nm$.

The domain Nm is diagrammed in Figure 2.4 below. Notice that the chain of finite partial elements $S(1)$, $S^2(1)$, $S^3(1)$... includes the proper limit point:-

$$S_\infty = \bigsqcup_{i \in \omega} S^i(1_a)$$

Note that $S(S_\infty) = S_\infty$. By the continuity of InitNm and the given

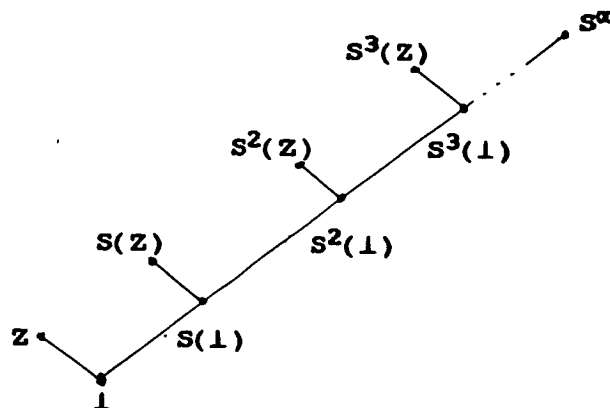


FIGURE 2.4

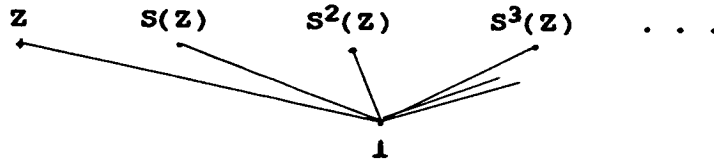


FIGURE 2.5

operation $sc \in [D \rightarrow D]$, we have that:-

$$\text{InitNm}(z, sc)(S_\omega) = \bigsqcup_{i \in \omega} sc^i(1_D) = \text{FIX}(sc)$$

Moreover, $\text{InitNm}(z, sc)(S^n(Z)) = sc^n(z)$, for each $n \in \omega$. This simply illustrates how InitN gives a general-purpose function "iterator" where z is the "initial value", sc is the "step function" and $S^n(Z)$ (a "unary representation" of the number n) gives the number of iterations to be made. Of course, since Nm contains non-trivial partial elements, the value of only doing some partially-completed number of iterations is also well-defined.

If the initial $\sqcup$ -algebra with respect to CPO_1 (i.e. strict operations) was taken instead by using the following domain equation:-

$$\text{Nm}_1 \cong \text{dot}_1 + \text{Nm}_1$$

then the "initiality" functional has identical recursive form as before since this depends solely upon the form of the signature involved. As can be seen from the diagram of this domain in Figure 2.5 above, there are no non-trivial finite partial elements in the domain and hence no proper infinite points. Such a domain serves as an adequate model of the natural numbers.

Another data type defined by initiality whose corresponding "initiality" functional has obvious application is the familiar "list" data type (containing both finite and infinite lists). This data type may be defined using the following signature, Σ_L , where A is some given domain of atoms.

$$\begin{array}{ll} E : L & \text{(the "empty list")} \\ \text{Cs} : (A \# L) \rightarrow L & \text{(the "cons" operation)} \end{array}$$

The corresponding domain equation for the initial Σ_L -algebra is:-

$$L \cong \underline{\text{dot}}_1 + (A \# L)_1$$

Any L -algebra consists of a domain D , a constant $e : D$ and a function $cs : [(A \# D) \rightarrow D]$. The appropriate "initiality" functional $\text{InitL} : (D \# ((A \# D) \rightarrow D)) \rightarrow L \rightarrow D$, is the least continuous function satisfying the following homomorphism condition:-

$$\begin{aligned} \vdash \text{InitL}(e, cs)(1_L) &= 1_D \\ \vdash \text{InitL}(e, cs)(E) &= e \\ \vdash \text{InitL}(e, cs)(Cs(a, l)) &= cs(a, \text{InitL}(e, c)(l)) \end{aligned}$$

As before, this can be given a standard lambda definition (thereby also demonstrating computability):-

$$\text{InitL}(e, cs) = \text{FIX}(\lambda f. \lambda l. \text{IsE}(l) \Rightarrow e \mid cs((H\ l), f(T\ l)))$$

where, as before, $\text{IsN} : L \rightarrow \text{tr}$ is the usual "empty list" discriminator, and $H : L \rightarrow A$ and $T : L \rightarrow L$ are the corresponding selector functions for the "cons" function, Cs .

In general, the systematic use of "initiality" functionals can be used to encapsulate and pre-package standard "primitive recursive" forms for many data types. In this way, the use of explicit recursion can be concentrated into a number of standard forms. Functions of interest are then defined whenever possible as derived functions, i.e. as functional combinations of "initiality" functionals, constructor functions and other functions defined in this manner; some simple examples are given below:-

$$\begin{aligned} \text{Append} : (L \# L) &\rightarrow L \\ \text{Append}(l_1, l_2) &= \text{InitL}(l_2, Cs)(l_1) \end{aligned}$$

$$\begin{aligned} \text{Cl} : (A \# L) &\rightarrow L \\ \text{Cl}(a, l) &= \text{Append}(l, Cs(a, E)) \end{aligned}$$

$$\begin{aligned} \text{Rev} : L &\rightarrow L \\ \text{Rev}(l) &= \text{InitL}(E, \text{Cl})(l) \end{aligned}$$

$$\begin{aligned} \text{Map} : (A \rightarrow A) &\rightarrow (L \rightarrow L) \\ \text{Map}(f)(l) &= \text{InitL}(E, \lambda(a, ll). Cs(f(a), ll))(l) \end{aligned}$$

In this style of programming, the emphasis is placed on giving the application and composition of a few general-purpose functions which might then have efficient low level machine-oriented implementations. However, the complexity of programming would then be located in the constructing these composite functions.

This, in turn, promotes the idea of giving a simple programming language to assist the expression of these instances. Such a language would permit the declaration of data types which introduce the appropriate operators as well as giving the corresponding "initiality" functional. In a sense, Burstall's language HOPE [HOPE80] and also Goguen's language OBJ [GoguenTardo77] do this already by permitting functions to be defined by separate cases, indexed upon the "constructor" functions. Such definitions rely upon the injectiveness of the constructors to ensure their well-formedness.

2.4 Continuous Quotients.

We have informally discussed various notions of Σ -algebra in the context of Scott's domain theory. The existence of free and initial (continuous) Σ -algebras have been discussed in the anarchic case (where there are no equations or other axiomatic properties).

In general, the more interesting case is when some non-trivial equations or axioms are required to hold. Within the traditional (set-theoretical) framework given by Universal Algebra, initial and free Σ -algebras satisfying these axioms will exist and possess a standard, universal construction, by quotienting the free term algebra by the least congruence satisfying the equations (see [Cohn65], [Gratzer79] for example). However, the situation is not as simple in the domain theoretic setting.

The difficulty is simply that the implied congruence may be inconsistent with the Scott ordering upon the initial anarchic continuous Σ -algebra. Even when it is consistent, the space may not be ω -complete with respect to the ordering inherited from this underlying Σ -algebra.

The first problem may be solved by taking the least congruence that is consistent with the underlying ordering of terms. This

amounts to including equalities implied by the ordering amongst those implied directly from the given equations. It is also equivalent to ensuring that the stated operators (satisfying the equations) are necessarily monotonic.

As shown in, for example, the Appendix of [Milner77], the standard (set-theoretic) term algebra construction with the above congruence will give both the initial and free (monotonic) Σ -algebras with respect to the underlying category of POSET. In this case, each term will have finite depth since completeness of the carrier posets is not required for such categories of algebras.

The second problem is solved using McNeill's "completion by ideals" upon the free and initial Σ -algebra with respect to the category POSET mentioned above. This gives the required algebras since every monotonic function can be uniquely extended to a corresponding continuous function over the completed spaces. Moreover, since there are no proper limits (i.e. only finite terms) in the carriers before completion, the "completion by ideals" gives necessary and sufficient limit points, thereby ensuring the existence and uniqueness of the appropriate continuous Σ -homomorphisms. For further details, see [Milner77].

The work presented in Levy and Maibaum [LevyMaibaum82] gives another interesting approach to the problem of continuous quotients. They investigate certain natural conditions for when the standard quotient construction directly gives the initial continuous Σ -algebra.

As observed above, the quotient of the term algebra can fail to give even a monotonic Σ -algebra for certain congruences. So attention is restricted to those congruences which preserve lubs of ω -chains - the so-called continuous Σ -congruences. In addition, the following notion of a continuous normalising function is introduced.

Let $\approx$ be a continuous Σ -congruence on CT_{Σ} (the domain of all (continuous) ground Σ -terms) and suppose that $f : CT_{\Sigma} \rightarrow CT_{\Sigma}$ is continuous. Then f is a continuous normalizing function for $\approx$ iff

- (a) $\forall t_1 t_2 \in CT_{\Sigma}. t_1 \approx t_2 \supset f(t_1) = f(t_2)$
- (b) $\forall t \in CT_{\Sigma}. f(t) \approx t$

It is immediate that f is idempotent (i.e. that $f(f(t)) = f(t)$), since $f(f(t)) \approx f(t)$ and that $t_1 \approx t_2 \Rightarrow f(t_1) \approx f(t_2)$. Hence the $\approx$ -congruence is completely characterised as the kernel of the normaliser function, f .

The main result of [LevyMaibaum82] is that if $\approx$ is a continuous $\approx$ -congruence which also possesses a continuous normaliser function then the usual quotient model, i.e. $CT_{\mathcal{L}}/\approx$, is an initial continuous $\approx$ -algebra satisfying the congruence.

The partial ordering given to this initial $\approx$ -algebra is inherited directly from the underlying term algebra. This is obviously equivalent to "lifting" the ordering using a continuous (and hence monotonic) normalisation function.

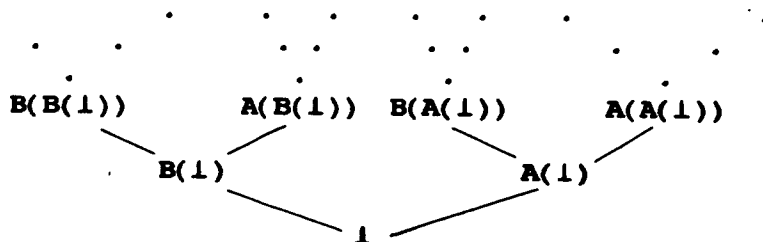
It is also shown (by an example) that there exists a $\approx$ -algebra and a continuous congruence $\approx$ which does not possess a continuous normalisation function. Moreover, the natural quotient algebra is not, in this case, a continuous $\approx$ -algebra and so cannot be initial in the required category.

The converse of the main result given above is easily shown to be false, by considering the following simple example:-

Let $\mathcal{L}$ have a single sort, S , with the operators $A : S \rightarrow S$ and $B : S \rightarrow S$. The anarchic initial $\approx$ -algebra is given as the least solution of the domain equation:-

$$S \approx S_1 + S_1$$

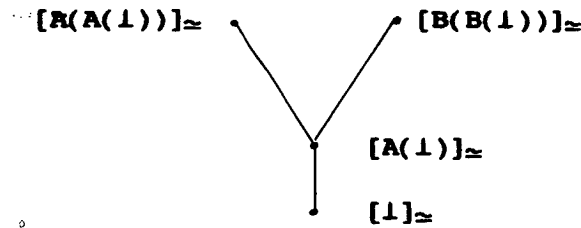
A finite sketch of part of this domain is:-



Now, consider the least congruence $\approx$ defined by the equations:-

$$\begin{aligned} A(1) &\approx B(1) \\ A(A(t)) &\approx A(B(t)) \\ B(A(t)) &\approx B(B(t)) \end{aligned}$$

The quotient of S by $\approx$ is a simple finite poset viz:-



where, as usual, $[x]_{\approx} = \{ y \in S \mid x \approx y \}$.

Clearly, $\approx$ is a continuous congruence, and the resulting Σ -algebra is clearly continuous (since there are no proper limits in the quotient space upon which the operators could be non-continuous). Since the quotient algebra is always initial in the full category of Σ -algebras, the above is also the initial continuous Σ -algebra satisfying the congruence. However, we have the following lemma:-

Lemma 2.6

There is no monotonic normalisation function $f : S \rightarrow S$ such that:-

$$f(t_1) \equiv f(t_2) \nRightarrow t_1 \approx t_2.$$

Proof

Suppose that $f(A(1)) \equiv A(1)$. Now:- $[A(1)]_{\approx} \subseteq [B(A(1))]_{\approx}$ and so $A(1) \equiv f(A(1)) \subseteq f(B(A(1)))$. But $f(B(A(1))) \in [B(A(1))]_{\approx}$ which implies that $B(1) \subseteq f(B(A(1)))$. This is a contradiction, since there is no element c in S such that $A(1) \subseteq c$ and $B(1) \subseteq c$. A similar contradiction can be proven in the case that $f(A(1)) \equiv B(1)$ which completes the proof.

QED

The technique of giving a continuous normalisation function is illustrated in [LevyMaibaum82] using two computer science oriented examples:- finite and infinite lists (to model circular lists) and a simple algebra of recursive functions.

2.4.1 Equivalence predicates.

When a continuous Σ -algebra (of a basic data type) is given by some equational axioms, it is reasonable to expect that such equalities should be effectively decidable within any

implementation. For example, suppose that a data type D is specified using an equational λ -algebra presentation. An implementation of the algebra is then, ideally, an explicit model constructed using more primitive, given constructions. (the above statement should be read modulo "implementation restrictions" e.g. only lists less than a certain (machine-dependant) length may actually be realisable in practice). The need for a decidable equality predicate is made clear if, for example, it is required to process arbitrary lists, or sets, of values from D . The algorithms generally need to compare elements of the representing data structures in a way which respects the stated laws. For example, to delete all occurrences of a given item from a given list, this will entail inspecting each item on the list, checking if it is equal to the given item and if so, removing it. Similar activities are undertaken when processing finite sets of items from D .

It may be argued that the point of using domain theory at all is to provide denotations of process-like entities, e.g. functions. In such cases, it is quite reasonable not to require continuous equality predicates upon such entities. Usually, these denotations are required for quite specific purposes. For example, the only thing one expects to use a function denotation for is to apply it to an argument. It might be considered odd if it were used to index an array! The problem, of course, is if the thing being used as an index is really a function denotation (usually some sort of infinite object) or if it is some "coded" finite representation of it, suitable for machine evaluation.

The main point is that it should also be possible to consider manipulations of non-process oriented data (e.g. lists of numbers or records involving numbers and characters) in the same logical framework as used for discussing process-oriented data (e.g. function denotations). The need to consider both is made more acute if the problem of the correctness of the algorithms used for performing these manipulations is considered.

The starting point is the observation that in order to manipulate basic data it is often necessary to test for equality of elements. Moreover, this equality predicate is intended to be

total, i.e. the result is defined whenever each argument is defined. Hence, the predicate must characterise an equivalence relation over the representation. Finally, there will usually be some requirement that an equivalence predicate represents a congruence with respect to certain operators in some desired algebra.

2.4.2 General properties of equivalence predicates.

In this section, the concept of equivalence predicate is formally introduced. A number of properties are given, in PPLAMBDA, about arbitrary equivalence predicates.

These are all contained within an LCF theory called EQFUN. This theory is, in turn, based upon the theory BASIC, discussed previously in Section 2.1.2. Note that the theories PL and KERNEL are also inherited.

Let EQ have the type $:\alpha \rightarrow \alpha \rightarrow \text{tr}$. EQ is an equivalence predicate iff it satisfies the following axiomatic properties:-

- 'EQ1']- $\forall x y:\alpha. \partial(\text{EQ } x \ y) = \partial(x) \text{ and } \partial(y)$
- 'EQ2']- $\forall x:\alpha. \text{EQ } x \ x = \partial(x)$
- 'EQ3']- $\forall x y:\alpha. \text{EQ } x \ y = \text{EQ } y \ x$
- 'EQ4']- $\forall x y z:\alpha.$
 $(\text{EQ } x \ y = \text{TT}) \ \& \ (\text{EQ } y \ z = \text{TT}) \supset (\text{EQ } x \ z = \text{TT})$

The first axiom states that the test for equivalence is defined whenever both arguments are defined. The second axiom states that we can test for self-equality precisely when the item in question is known, or defined, and as such corresponds to reflexivity of EQ. The third and fourth properties state that EQ is, in some sense, symmetric and transitive.

The derived general properties are given in the following 3 lemmas.

Lemma 2.7 (Bistrictness)

- 'EQUUX' .]- $\forall x:\alpha. \text{EQ } 1 \ x = 1$
- 'EQXUU' .]- $\forall x:\alpha. \text{EQ } x \ 1 = 1$

(on the assumption that the function EQ satisfies the definedness property given by 'EQ1')

Proof

Only the proof of lemma 'EQ1UU' is given since the proof of 'EQxUU' is similar. The proof proceeds by definedness case analysis on the value of "(EQ x 1)" and use of contradiction.

Suppose that $(EQ\ x\ 1) \neq 1$ i.e. that $\partial(EQ\ x\ 1) = TT$. Now, by the assumption 'EQ1' we have that $\partial(EQ\ x\ 1) = (\partial(x) \text{ and } \partial(1)) = (\partial(x) \text{ and } 1)$. Consider the definedness of "x". If $\partial(x) = TT$ then $(\partial(x) \text{ and } 1) = (TT \text{ and } 1) = 1$; otherwise if $\partial(x) = 1$ then $(\partial(x) \text{ and } 1) = (1 \text{ and } 1) = 1$. So, in either case, we have that $(\partial(x) \text{ and } 1) = 1$. This is a contradiction, and so the original assumption must also have been false; by reductio ad absurdum, or the contradiction rule, the desired conclusion can be drawn.

Turning now to the case where $\partial(EQ\ x\ 1) = 1$ we can immediately conclude that $(EQ\ x\ 1) = 1$. Since in both cases, $(EQ\ x\ 1) = 1$, we have obtained the desired result.

QED

The next lemma is a useful generalisation of the transitivity property (given by 'EQ4').

Lemma 2.8 (Generalised transitivity)

'GenTrans'

...]- $\forall x\ y\ z:\alpha\ t:\text{tr}.$

$$(EQ\ x\ y) = TT \ \& \ (EQ\ y\ z) = t \ \supset \ (EQ\ x\ z) = t$$

(on the assumption that 'EQ1', 'EQ3' and 'EQ4' hold. It does not depend upon "reflexivity", 'EQ2').

Proof

Assume that, for arbitrary $x, y, z:\alpha$ and $t:\text{tr}$ we have that $(EQ\ x\ y) = TT$ and $(EQ\ y\ z) = t$. Now, applying " ∂ " to both sides of both assumptions we get:-

1. $\partial(EQ\ x\ y) = \partial(TT) = TT$. Now, from 'EQ1' we have that $TT = (\partial(x) \text{ and } \partial(y))$. Hence, using the 'AndAnalysis' rule from theory PL, $\partial(x) = TT$, and also $\partial(y) = TT$.

2. Similarly, $\partial(\text{EQ } y \ z) = \partial(t) = \partial(y) \text{ and } \partial(z)$, and so $\partial(t) = (\text{TT and } \partial(z)) = \partial(z)$, i.e. $\partial(t) = \partial(z)$.

These two properties help shorten the case analyses which follow. We continue with a truth value case analysis on the value of "t".

Suppose that $t = \text{TT}$. Then $(\text{EQ } y \ z) = \text{TT}$ and so by ordinary transitivity, property 'EQ4', we get that $(\text{EQ } x \ z) = \text{TT} = t$, which completes this case.

Suppose that $t = \text{FF}$. We now give a second truth value case analysis on the value of $(\text{EQ } x \ z)$.

Suppose that $(\text{EQ } x \ z) = \text{TT}$. Then by the symmetry property, 'EQ3', we get that $(\text{EQ } z \ x) = \text{TT}$. So, since $(\text{EQ } x \ y) = \text{TT}$ we get, using 'EQ4', $(\text{EQ } z \ y) = \text{TT}$. By another application of 'EQ3', we get that $(\text{EQ } y \ z) = \text{TT}$. However, this contradicts the assumption that $(\text{EQ } y \ z) = t = \text{FF}$; hence, we may formally conclude from this absurdity the desired result.

Suppose that $(\text{EQ } x \ z) = \text{FF}$. This immediately gives $(\text{EQ } x \ z) = t$, since we have assumed that $t = \text{FF}$.

Suppose that $(\text{EQ } x \ z) = 1$. Now, apply " ∂ " to both sides to give $\partial(\text{EQ } x \ z) = \partial(1) = 1$. So, using 'EQ1' again we get $\partial(x) \text{ and } \partial(z) = 1$. But from previous assumptions $\partial(x) = \text{TT}$ and that $\partial(z) = \partial(t)$. So, substituting to get:- $1 = \partial(x) \text{ and } \partial(z) = \text{TT and } \partial(t) = \partial(t) = \partial(\text{FF}) = \text{TT}$, since we have assumed that $t = \text{FF}$. This contradiction concludes this particular case. In each of the above, we have shown that the assumption $t = \text{FF}$ implies that $(\text{EQ } x \ z) = \text{FF}$.

Suppose now that $t = 1$. From previous assumptions we get that $\partial(z) = \partial(t) = \partial(1) = 1$ and hence that $z = 1$. So, by Lemma 2.7, we have that $(\text{EQ } x \ z) = \text{EQ } x \ 1 = 1 = t$, concluding this proof.

QED

The last general property given here concerning equivalence predicates is the so called "Self-Congruence" property. This states that for points $a_1, a_2, b_1, b_2: \alpha$, if $(\text{EQ } a_1 \ a_2) = \text{TT}$ and $(\text{EQ } b_1 \ b_2) = \text{TT}$ then the equation $(\text{EQ } a_1 \ b_1) = (\text{EQ } a_2 \ b_2)$ holds. This relationship is illustrated with the diagram:-



The equivalences represented by the dotted lines are either both true or both false. Note that these equivalences can not be undefined, by 'EQ1'.

Lemma 2.9 (Self-Congruence)

'SelfEQCong'

...]- $\forall$ a1 a2 b1 b2: α .

(EQ a1 a2) = TT & (EQ b1 b2) = TT $\supset$

(EQ a1 b1) = (EQ a2 b2)

(on the assumption that 'EQ1', 'EQ3' and 'EQ4' hold).

Proof

This proof proceeds by using several applications of the symmetry property 'EQ3' and the 'GenTrans' lemma above.

Let (EQ a1 b1) = t, where t is some arbitrary truth value. Now, from 'EQ3', we have that (EQ a2 a1) = (EQ a1 a2) = TT. By combining this with the above assumption using the 'GenTrans' lemma we get that (EQ a2 b1) = t. By applying 'EQ3' twice more, (EQ b1 a2) = (EQ a2 b1) = t, and also (EQ b2 b1) = (EQ b1 b2) = TT. So, applying the 'GenTrans' again we reach (EQ b2 a2) = t and then applying 'EQ3' we have that (EQ a2 b2) = (EQ b2 a2) = t. Hence, we have shown that (EQ a1 b1) = t = (EQ a2 b2), for any arbitrary truth value t.

QED

My original proof of this lemma used a complicated case analysis. Inspection of this proof led to the formulation of the 'GenTrans' lemma. (In effect, its proof was embedded in a "FF" case.) The above proof was found by noticing that the effective use of 'GenTrans' in the original proof did not strongly depend on the use of the value "FF".

None of the above lemmas depended upon the "reflexivity" property, 'EQ2'. So, for example, they hold for the following function $f:\alpha \rightarrow \alpha \rightarrow \text{tr}$ where $(f\ x\ y) = (\partial(x) \text{ and } \partial(y)) \Rightarrow \text{FF} \mid \perp$. Clearly, f satisfies properties 'EQ1', 'EQ3' and 'EQ4' but not

'EQ2', and so is not an equivalence predicate. The property 'EQ4' holds vacuously since for no values $a, b: \alpha$ is $(f\ a\ b)$ ever equal to "TT".

Each of the above lemmas contain the arbitrary function indeterminate $EQ : \alpha \rightarrow \alpha \rightarrow tr$, and have been predicated upon some of the assumptions 'EQ1' to 'EQ4'. Trivially, each of these lemmas are equivalent to a theorem with no assumptions i.e. empty sequent, and with no free variables i.e. a sentence in PPLAMBDA. For example, the lemma 'EQUUX' is equivalent to the sentence:-

$$\begin{aligned}]- \forall EQ: \alpha \rightarrow \alpha \rightarrow tr. \\ (\forall x\ y: \alpha. \partial(EQ\ x\ y) \equiv \partial(x)\text{and}\partial(y)) \supset (\forall x: \alpha. EQ\ 1\ x \equiv 1) \end{aligned}$$

The practical advantage of such "higher-type" theorems is that they may be used by instantiating types of EQ, specialising EQ and then discharging antecedents either by proving them independently or permitting them as justified assumptions.

2.4.3 Effective congruences.

The equivalence (or equality) predicates satisfying the axioms above will, generally speaking, be given a recursive definition of some kind. Such definitions are stated with respect to the underlying representation algebra. Any equivalence predicates satisfying the above axioms and which is recursively defined is said to be effective.

In addition, there is usually some requirement that the equivalence predicate represents a congruence with respect to certain operators. The general formulation of this kind of property is straightforward and is briefly illustrated by an example here. Suppose that $EQ : D \rightarrow D \rightarrow tr$, is an equivalence predicate for some domain D , and that $F : (D \# D) \rightarrow D$ is a (total) operation on D for which EQ is a congruence. This property can be formally expressed in PPLAMBDA by:-

$$\begin{aligned} \vdash \forall d_1\ d_2\ d_3\ d_4: D. \\ (EQ\ d_1\ d_2 \equiv TT) \ \& \ (EQ\ d_3\ d_4 \equiv TT) \supset \\ (EQ\ (F(d_1, d_3))\ (F(d_2, d_4))) \equiv TT \end{aligned}$$

This condition will usually be required for each basic operator of

the particular algebra in question.

2.4.4 The relation represented by an equivalence predicate.

Let D be some given domain. The binary relation, $\Xi \subseteq D^2$, which a given equivalence predicate $EQ : D \rightarrow D \rightarrow \text{tr}$ is taken to represent is defined by:-

$$\forall x, y : D. (x \Xi y) \iff (\partial(x) = \partial(y)) \ \& \ (EQ \ x \ y = \partial(x))$$

With this definition, we have the following immediate observation:-

Lemma 2.10

- (1) $\forall x : D. (1 \Xi x) \supset x = 1$
- (2) $\forall x, y : D. \partial(x) = \text{TT} \ \& \ x \subseteq y \supset (x \Xi y)$

Proof

- (1) Suppose that $1 \Xi x$. Hence, $\partial(1) = \partial(x)$
and so $x = 1$.

- (2) Suppose that $\partial(x) = \text{TT} \ \& \ x \subseteq y$.
So, $\text{TT} = \partial(x) \subseteq \partial(y)$ and also we have that
 $\text{TT} = (EQ \ x \ x) \subseteq (EQ \ x \ y)$, by monotonicity.
Hence $(x \Xi y)$.

QED

Also, we show the following (expected) result:-

Lemma 2.11

Ξ is an equivalence relation on D .

Proof

Ξ is reflexive since if $x \in D$ then $(x \Xi x) \iff \partial(x) = \partial(x)$
& $(EQ \ x \ x = \partial(x)) \iff (EQ \ x \ x = \partial(x))$, which is equivalent to
axiom EQ2.

Ξ is symmetric since if $x, y \in D$ then

$$\begin{aligned} (x \Xi y) &\iff (\partial(x) = \partial(y)) \ \& \ (EQ \ x \ y = \partial(x)) \\ &\iff (\partial(y) = \partial(x)) \ \& \ (EQ \ y \ x = \partial(x)) \end{aligned}$$

$$\begin{aligned} & \# (\partial(y) = \partial(x)) \ \& \ (EQ \ y \ x = \partial(y)) \\ & \# (y \Xi x) \end{aligned}$$

Ξ is transitive, since for any $x \ y \ z \in D$ such that $(x \Xi y)$ and $(y \Xi z)$ we have:-

$$\begin{aligned} (a) \quad & (x \Xi y) \ \# \ (\partial(x) = \partial(y)) \ \& \ (EQ \ x \ y = \partial(x)) \\ (b) \quad & (y \Xi z) \ \# \ (\partial(y) = \partial(z)) \ \& \ (EQ \ y \ z = \partial(z)) \end{aligned}$$

Now, from this we get that $\partial(x) = \partial(z)$ (by transitivity of $=$) and we now proceed by cases on the definedness of x . If $x = 1$ then, clearly, $x = y = z = 1$, and so $(x \Xi z)$. On the other hand, suppose that $\partial(x) = TT$. Then certainly $(EQ \ x \ y) = TT$ and also $(EQ \ y \ z) = \partial(y) = TT$. Therefore, by "transitivity" of EQ we have that $(EQ \ x \ z) = TT = \partial(x)$, so completing this proof.

QED

Taking these two lemmas together, it is clear that the quotient set, $Q = D/\Xi$ is well-defined. Moreover, the first lemma shows that Q can be given the flat domain ordering with $\{1\}$ as the least element. This ordering respects the original ordering on D in the sense that the natural mapping $d \mapsto [d]_{\Xi}$ is continuous. Finally this partial ordering of Q is the least such with this property (since it is flat).

This shows that by giving a (possibly non-flat) domain D an equivalence predicate satisfying the axioms EQ1 - EQ4, a flat domain is obtained as the quotient. This consequence is desirable when effectiveness requirements are taken into account.

2.5 Monoids.

Monoids are a very simple example of an algebraically defined structure. A monoid consists of a set, A , with a binary operation denoted by $\otimes: A^2 \rightarrow A$ and a distinguished element, $\square \in A$, both of which satisfy the following axioms:-

$$\begin{aligned} & \forall a \in A. \ \square \otimes a = a \\ & \forall a \in A. \ a \otimes \square = a \\ & \forall a \ b \ c \in A. \ (a \otimes b) \otimes c = a \otimes (b \otimes c) \end{aligned}$$

So, the element $\square$ is both a left and right identity for the

operation $\otimes$ which is taken to be associative. Such (very) simple structures are also known as semigroups and virtually every mathematical structure of any significance has some sort of semigroup structure (e.g. Categories, Groups, Rings, Modules, Fields, etc).

However, a more convenient, computationally realisable definition of "monoid" will be useful in this domain theoretic setting. This is the (left-strict) continuous monoid which consists of some domain D , a continuous binary operation $\otimes : [D^2 \rightarrow D]$ and an element $1:D$ which forms a monoid as described above, as well as satisfying the following left-strictness axiom:-

$$\vdash \forall d:D. 1 \otimes d = d$$

This axiom is justified mainly on pragmatic grounds; in giving an "implementation" of the binary operation, one or other of its arguments will be "evaluated" first (assuming sequential reduction). By convention, it is assumed that this operation is left-strict, as the theory of right-strict continuous monoids is equivalent. From now on, the term "monoid" is taken to mean "left-strict continuous monoid", unless otherwise indicated.

We shall also need the notion of bi-strict (continuous) monoid which is a left-strict continuous monoid that, in addition, satisfies the right-strictness axiom:-

$$\vdash \forall d:D. d \otimes 1 = d$$

A commutative monoid is, for us at least, a bi-strict continuous monoid satisfying the axiom of commutativity:-

$$\vdash \forall d_1 d_2:D. (d_1 \otimes d_2) = (d_2 \otimes d_1)$$

Monoids have various natural applications in computer science. For example, many examples of operations that involve the composing together of some kind of data structure frequently have a monoid-like structure (e.g union of finite sets, concatenation of lists, addition of numbers, composition of tactics and functions).

However, our main concern later is with "free monoids" of various kinds. In a sense, the universal property of free monoids shows that all of the monoids mentioned above could be modelled symbolically, so far as their monoid structure is concerned.

2.5.1 Categories of monoids.

In the domain theoretic setting for PPLAMBDA, our interest is focussed upon the Scott continuous left-strict monoids introduced above. The class of all such (small) monoids naturally forms a category of continuous algebras, **MONOID**, with respect to **CPO**. From Section 2.3, each morphism, $m: [D_1 \rightarrow D_2]$, of this category is a Scott continuous function for some domains D_1, D_2 , such that:-

$$\vdash m(\square_1) = \square_2$$

$$\vdash \forall d_1 d_2: D_1. m(d_1 \otimes_1 d_2) = m(d_1) \otimes_2 m(d_2)$$

where $\langle \square_i, \otimes_i \rangle$ forms a left-strict continuous monoid on D_i , for $i \in \{1, 2\}$. Note that the morphisms are NOT necessarily strict here. The category **MONOID**₁ is the proper sub-category of **MONOID**, all of whose morphisms are strict monoid morphisms. The composition in either case is functional composition, and has the obvious identity morphisms.

2.5.2 An induction principle for monoids.

Assume that $\langle \square, \otimes \rangle$ is a left strict monoid on carrier D , and that there is an injection function $U: A \rightarrow M$, which satisfies the following schema:-

$$\begin{array}{l} \vdash fm[1] \\ \vdash fm[\square] \\ \vdash \forall a: A. fm[U(a)] \\ \vdash \forall m_1 m_2: M. fm[m_1] \ \& \ fm[m_2] \supset fm[m_1 \otimes m_2] \\ \hline \vdash \forall m: M. fm[m] \end{array}$$

where $fm[m]$ is an admissible formula in the free variable m . This schema is called the Monoid Induction Principle (or MIP).

The schema may or may not be valid for particular choices of monoid and "injection" function. For example, take the arithmetical monoid, $(1, (- * -))$ monotonically extended to the flat domain of non-zero numbers, with the two injection functions defined by $U_1(i) = p_i$ (i.e. the i th prime) and $U_2(i) = 3^i$. By the Prime Factorisation theorem, the MIP is valid for the given monoid with the first injection function. However, it is clearly invalid for the given monoid with the second injection.

Chapter 3

Monoid Case Studies

All three case studies presented in this chapter are variations upon a common theme. Each study shows that a given construction, formulated in PPLAMBDA is a "free object" in a certain category of algebras, with respect to a natural choice of injection, or "Unit", function. The link between the studies lies in the variation of certain underlying categories, which takes the form of a corresponding variation in certain strictness assumptions on the class of valuation morphisms available, and the binary operation. The case-studies also illustrates how strictness assumptions can affect the choice of the underlying domains and also shows their effect upon detailed, formal argument.

The theory structure and their development in each of the studies follows the same basic pattern, which is described here for the first case study. A domain operator for a particular kind of list is defined, with the standard collection of primitive list operations (including the "empty list", Nil) in a theory called L. This theory is built upon the theory BASIC (introduced in Section 2.1.2), which inherits both the theories KERNEL and SMASH. Next, a theory of simple list functions, called LFUN, is introduced based upon the theory L. Both the list concatenation function, denoted by infix $(- @ -)$, and the unit function Unit are introduced here, and $\langle \text{Nil}, @ \rangle$ is shown to form a monoid. The final theory introduced in each case study is built on LFUN and is called LFREE. In this theory, a second order functional, FM, is defined which, when suitably applied, produces the unique strict extension morphism making the appropriate freeness diagram commute. The purpose of introducing this functional is to formalise the freeness criterion in PPLAMBDA.

The overall theory diagram for all three case studies is given in Figure 3.1 below. In order to ease comparisons between the theories of each case study, appropriate theory names have been

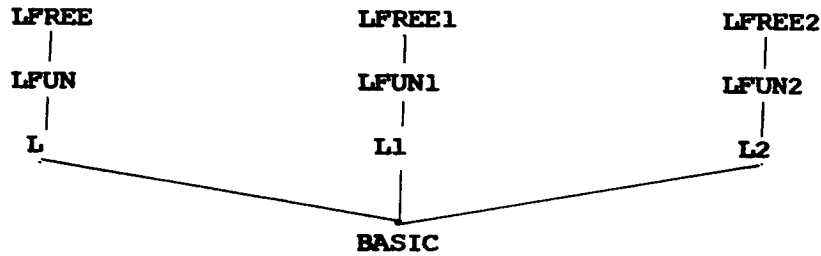


FIGURE 3.1

given. Theories developed within the second case study have names postfixed by "1"; those for the third case study are postfixed by "2".

In each of the case studies, the monoid morphisms are always taken to be strict. In the terminology of Section 2.5.1, we are working in the category MONOID_1 , when dealing with monoids. However, each case study is concerned with the variation of strictness assumptions of the valuation function, or whether the binary operation is just left-strict or whether it is bi-strict.

The first study is concerned with the free (left-strict) monoid where the valuation functions are arbitrary CPO morphisms and where the "Unit" function used is non-strict. The second study discusses the free (left-strict) monoid, but where the valuation functions and the "Unit" function are all CPO_1 morphisms (i.e. strict). Finally, the third study discusses the free (bi-strict) monoid* where, as for the second study, all morphisms are strict. In addition, the binary operation is strict in each argument independently.

3.1 The First Monoid Case Study.

In this first case study, no special strictness properties are assumed here for arbitrary functions, except that each monoid's binary operation is left-strict (and hence strict). In the following three sections, an informal presentation of the

* A free bi-strict monoid for a generating domain D exists, where the valuation morphisms are arbitrary CPO morphisms and the "Unit" function is non-strict. The carrier of this monoid is a least solution of the domain equation $L \approx \text{dot}1 + (D_1 \otimes L)$. However, this was not formally studied here.

"freeness" proof is given. This is then followed by its formal counterpart in PPLAMBDA.

The direction and structure of the informal argument presented will correspond to analogous (tactical) components of the formal proof. The formal statement of the final results is motivated and, most importantly, shown to correspond to the properties they are purported to formalise.

During the informal presentation, various operators and functions are omitted from formulae as they tend to distract attention by introducing irrelevant detail into proofs. Most of the functions omitted are artifacts of the formalisation adopted and as such do not essentially contribute to the basic argument. In particular, the isomorphism pair, operators dealing with domain summation, and such like are not mentioned. When formal arguments are given, these details will be included as and where necessary.

3.1.1 A domain of lists.

Let "List" be a domain operator with a single argument, such that, for each domain α , the domain $(\alpha)\text{List}$ is a "well-founded" solution of the domain equation:-

$$(\alpha)\text{List} \cong \text{dot}_1 + (\alpha \# (\alpha)\text{List})_1$$

The axiomatisation of this domain equation follows the technique discussed in Section 2.1.6, and so the mathematical details are omitted here. The formal axiomatisation is generated using the LCF package discussed in Section 5.1.1. From the discussion in Section 2.1.7, the domain $(\alpha)\text{List}$ contains both partial and total elements, and so contains, in general, non-trivial limit points (i.e. infinite lists).

Each of the primitive list processing functions introduced here are defined in terms of various injection, selection and pairing operations and the isomorphism pair that represents the domain equation for lists. As such, these definitions can be safely omitted here, as they are given formally in Section 3.2.1.

Nil : $(\alpha)\text{List}$
 (- :: -) : $\alpha \rightarrow (\alpha)\text{List} \rightarrow (\alpha)\text{List}$.

Nil denotes the "empty list" and $(- :: -)$ denotes the list constructor (or "Cons") operation which places a given element onto the front of a given list. The Nil value corresponds to the single proper value in the left summand of the domain equation for $(\alpha)\text{List}$, whereas the Cons operation corresponds to values from the right summand.

In order to be able to "select" components from lists and to define Boolean predicate functions on them, 3 more primitive polymorphic functions are introduced:-

```
Head : (α)List → α
Tail : (α)List → (α)List
Null : (α)List → tr
```

From their formal definitions, the usual properties concerning these standard list manipulating functions can be derived, for example:-

```
] - Null(Nil) = TT
] - Head(Nil) = 1
] - ∀l. Null(l) = FF  >  l = (Head l) :: (Tail l)
] - ∀l. Null(l) = TT  >  l = Nil
```

Note that:-

```
] - ∀ a l . ∂(a :: l) = TT
```

follows because the right summand of the domain equation is lifted. Hence, the Cons operation is not strict in either argument.

Finally, the usual structural induction schema over (polymorphic) lists is valid. Let $F[l]$ be any PPLAMBDA formula which admits induction in the freely occurring variable "l" of type $(\alpha)\text{List}$. The structural induction schema is now:-

$$\frac{] - F[l] \quad] - F[\text{Nil}] \quad] - \forall l_1. F[l_1] \supset \forall a. F[a :: l_1]}{] - \forall l. F[l]}$$

Note that structural induction is valid even when infinite lists are present. This phenomenon is a simple consequence of continuity as was illustrated in Section 2.1.4.

3.1.2 Concatenation and Unit

The concatenation function over lists is declared to have type:-

$$(- @ -) : (\alpha)\text{List}^2 \rightarrow (\alpha)\text{List}$$

and is recursively defined as usual by:-

$$(l_1 @ l_2) = (\text{Null } l_1) \Rightarrow l_2 \mid (\text{Head } l_1) :: ((\text{Tail } l_1) @ l_2)$$

where both l_1 and l_2 have type $(\alpha)\text{List}$. Note that the above definition asserts that $(- @ -)$ is any continuous function which satisfies the above equation. However, in this case, there is exactly one solution, as could be explicitly shown by a structural induction proof.

The usual properties of concatenation are stated in the following lemma.

Lemma 3.1

- (a) $\vdash \forall l. l @ l = l$
- (b) $\vdash \forall l. \text{Nil} @ l = l$
- (c) $\vdash \forall a \ l_1 \ l_2. (a :: l_1) @ l_2 = a :: (l_1 @ l_2)$
- (d) $\vdash \forall l. l @ \text{Nil} = l$
- (e) $\vdash \forall l_1 \ l_2 \ l_3. (l_1 @ l_2) @ l_3 = l_1 @ (l_2 @ l_3)$

Proof

The first three parts are straightforward and follow by substituting directly into the definition and then simplifying using list identities.

The last two parts are not as simple to prove; they both use structural induction to enable the definition of concatenation to be applied, and also use the first three properties above as results.

QED.

These lemmas together show that, for any domain D , $\langle \text{Nil}, @ \rangle$, is a monoid. For the specification of a free monoid to be meaningful, a particular "Unit" function is required. This is introduced below by the declaration:-

$$\text{Unit} : \alpha \rightarrow (\alpha)\text{List}$$

and defined by the equation:-

$$\vdash \forall a. \text{Unit}(a) = (a :: \text{Nil})$$

Observe that $\partial(\text{Unit } a) = \partial(a :: \text{Nil}) = \text{TT}$, and so Unit is NOT strict. Moreover, Unit is also an injective (polymorphic) function, since if $\text{Unit}(a_1) = \text{Unit}(a_2)$ for some $a_1, a_2 : \alpha$, then $\text{Head}(\text{Unit}(a_1)) = \text{Head}(\text{Unit}(a_2))$ which immediately gives $a_1 = a_2$.

The next observation concerning the Unit function plays a crucial role in the forthcoming arguments.

Lemma 3.2 (The "ConsUnit" Lemma)

$$\vdash \forall a \text{ l. } (a :: l) = \text{Unit}(a) @ l$$

Proof (by computation)

$$\text{Unit}(a) @ l = (a :: \text{Nil}) @ l = a :: (\text{Nil} @ l) = (a :: l)$$

QED

The "ConsUnit" Lemma above is crucial, since it links two different presentations of algebraic structures over the same domain - viz - lists which can be given either in terms of the "element-wise" list constructor Cons or in terms of Unit and Append .

This gives the possibility of using the Monoid Induction Principle introduced in Section 2.5.2 for $(\alpha)\text{List}$ in terms of Nil , Unit and $@$. This is confirmed with the following lemma which shows that two hypotheses of the Monoid Induction Principle implies the "Cons" case in the standard structural induction principle over lists. Since the remaining two parts of the monoid principle correspond exactly to the remaining parts of the standard induction schema, the validity of the monoid principle is demonstrated.

Lemma 3.3

Let $F[l]$ be any admissible property of $(\alpha)\text{Lists}$, and suppose that

- (1) $\vdash \forall a : \alpha. F[\text{Unit}(a)]$
- (2) $\vdash \forall l_1 \ l_2 : (\alpha)\text{List}. F[l_1] \ \& \ F[l_2] \supset F[l_1 @ l_2]$.

Then, we have that $\vdash \forall l. F[l] \supset \forall a. F[a :: l]$

Proof

Proceeding informally, let a and l be any elements of α and $(\alpha)\text{List}$ respectively, and assume that $F[l]$ is true. Now, by (1) above, $F[\text{Unit}(a)]$ holds, and so, via (2), we can conclude that $F[\text{Unit}(a) @ l]$ is true. But, by the Unit Lemma, this is equivalent to $F[a::l]$.

QED

The informal proof above forms the basis of a schematic tactic called MONOIDTAC for the application of Monoid Induction for lists. This tactic is discussed during the formal proofs later on and also in Section 5.3.

3.1.3 The Freeness Functional.

To demonstrate that the monoid construction given in the previous section is free with respect to Unit, a second-order functional, FM, is introduced which, when suitably instantiated, produces a (strict) monoid morphism extending a given valuation function. The technique used here is due to Burstall and Landin (see Section 2.3.3).

Let α be the domain of generators, and suppose that β is the carrier of the algebra (Z, P) , then:-

$$\text{FM} : \underbrace{(\beta \# (\beta^2 \rightarrow \beta))}_{\text{algebra on carrier } \beta} \rightarrow \underbrace{(\alpha \rightarrow \beta)}_{\text{valuation of generators}} \rightarrow \underbrace{((\alpha)\text{List} \rightarrow \beta)}_{\text{monoid extension morphism}}$$

and is recursively defined by the equation:-

$$\begin{aligned} &]- \forall Z:\beta \quad P:\beta^2 \rightarrow \beta \quad f:\alpha \rightarrow \beta \quad l:(\alpha)\text{List} . \\ & \quad (\text{FM } (Z, P) \ f) \ (l) \equiv \\ & \quad \text{Null}(l) \Rightarrow Z \mid P(f(\text{Head } l), (\text{FM } (Z, P) \ f) (\text{Tail } l)) \end{aligned}$$

Notice that the definition of FM does not depend upon any special properties of the tuple (Z, P) , except that it has the required signature. A consequence of this is that assumptions concerning (Z, P) and, indeed, f will need to be explicitly mentioned as the occasion arises.

Turning now to simple computational properties of FM, it is easily seen, by case analysis, that:-

$$\begin{aligned}
(\text{FM } (Z,P) f) (1) &\equiv 1 \\
(\text{FM } (Z,P) f) (\text{Nil}) &\equiv Z \\
(\text{FM } (Z,P) f) (a::l) &\equiv P(f(a), (\text{FM } (Z,P) f) l)
\end{aligned}$$

All of these properties are valid for arbitrary $Z:\beta$, $P:\beta^2 \rightarrow \beta$, $f:\alpha \rightarrow \beta$, $a:\alpha$ and $l:(\alpha)\text{List}$. The next proposition shows that FM does indeed extend the generator valuation function, f .

Lemma 3.4

Assuming that, for any $x:\beta$, $P(x,Z) \equiv x$, then, $(\text{FM } (Z,P) f)$ extends f , or more formally:-

$$\forall a:\alpha. f(a) \equiv (\text{FM } (Z,P) f) (\text{Unit}(a))$$

Proof (by calculation)

Let a be any element in α , and so, $(\text{FM } (Z,P) f) (\text{Unit } a) \equiv (\text{FM } (Z,P) f) (a::\text{Nil}) \equiv P(f(a), (\text{FM } (Z,P) f) (\text{Nil})) \equiv P(f(a), Z) \equiv f(a)$.

QED

As a trivial corollary of this lemma, if (Z,P) form a monoid, with carrier β , then $(\text{FM } (Z,P) f)$ extends f , via the "Unit" function. The next lemma is the main step in showing that $(\text{FM } (Z,P) f)$ is a strict monoid morphism (assuming appropriate relationships between Z and P , of course).

Lemma 3.5

Assuming that, for any $x, y, z: \beta$,

$$\begin{aligned}
P(1,x) &\equiv 1 \\
P(Z,x) &\equiv x \\
P(P(x,y),z) &\equiv P(x,P(y,z))
\end{aligned}$$

then, for any $l1\ l2 :(\alpha)\text{List}$,

$$\begin{aligned}
(\text{FM } (Z,P) f) (l1 @ l2) \\
\equiv P((\text{FM } (Z,P) f)(l1), (\text{FM } (Z,P) f)(l2))
\end{aligned}$$

Proof (By structural list induction on $l1$)

To shorten formulae, abbreviate " $(\text{FM } (Z,P) f)$ " by FM for the duration of this proof.

Suppose that $l1 \equiv 1$. So, $\text{FM}(l1 @ l2) \equiv \text{FM}(1 @ l2) \equiv \text{FM}(1) \equiv 1$. But then also $P(\text{FM}(l1), \text{FM}(l2)) \equiv P(\text{FM}(1), \text{FM}(l2)) \equiv P(1, \text{FM}(l2)) \equiv 1$.

Suppose that $l1 \equiv Nil$. So, $FM(l1 @ l2) \equiv FM(Nil @ l2) \equiv FM(l2) \equiv P(Z, FM(l2)) \equiv P(FM(Nil), FM(l2)) \equiv P(FM(l1), FM(l2))$.

Suppose that $l1 \equiv (a::l')$, with the induction hypothesis:-
 $\forall l2 . FM(l' @ l2) \equiv P(FM(l'), FM(l2))$.

So, $FM(l1 @ l2) \equiv FM((a::l') @ l2) \equiv FM(a::(l' @ l2)) \equiv P(f(a), FM(l' @ l2)) \equiv P(f(a), P(FM(l'), FM(l2)))$, using the induction hypothesis. However, we also have that $P(FM(l1), FM(l2)) \equiv P(FM(a::l'), FM(l2)) \equiv P(P(f(a), FM(l')), FM(l2)) \equiv P(f(a), P(FM(l'), FM(l2)))$. By combining these two calculations, we reach the desired result.

QED

This lemma, together with previous results, shows that the function $(FM (Z,P) f)$ is a strict monoid morphism whenever (Z,P) forms a monoid on carrier β . So, assuming that (Z,P) is a monoid on β then, for any $f : \alpha \rightarrow \beta$:-

- (1) $(FM (Z,P) f) (1_\alpha) \equiv 1_\beta$
- (2) $(FM (Z,P) f) (Nil) \equiv Z$
- (3) $(FM (Z,P) f) (l1 @ l2) \equiv P((FM (Z,P) f) (l1), (FM (Z,P) f) (l2))$

Interestingly enough, neither of the previous two major lemmas needed the full power of the permissible assumption that (Z,P) constitutes a monoid over β . However, their conjunction does require this assumption in full.

The last part of the discussion here shows that there is only one strict monoid morphism from $(\alpha)List$ to β compatible with $(Nil, @)$ and (Z,P) . Two proofs of this lemma are given (labelled A and B), the first using standard list induction, and the second using the monoid induction principle, for the monoid $(Nil, @)$. Again, " $(FM (Z,P) f)$ " is abbreviated by FM during the course of this proof to shorten and simplify formulae. So proceeding thus:-

Theorem 3.6 (The Uniqueness Theorem)

Suppose that $G: (\alpha)\text{List} \rightarrow \beta$ where:-

- (1) (Z, P) is a monoid over β
- (2) $\forall a: \alpha . f(a) \equiv G(\text{Unit}(a))$
(i.e. G extends f , with respect to Unit)
- (3) G is a strict monoid morphism (from $(\text{Nil}, @)$ to (Z, P))

then, for any $l: (\alpha)\text{List}$, $(FM (Z, P) f) (l) \equiv G(l)$

Proof (A) (by structural list induction on l)

Suppose that $l \equiv l$. So, $FM(l) \equiv l \equiv G(l)$, because G is strict.

Suppose that $l \equiv \text{Nil}$. So, $FM(\text{Nil}) \equiv Z \equiv G(\text{Nil})$, because G is a monoid morphism.

Suppose that $l \equiv (a::l')$, with the induction hypothesis:-

$$FM(l') \equiv G(l')$$

So, by computation, $FM(ll) \equiv FM(a::l') \equiv P(f(a), FM(l')) \equiv P(f(a), G(l'))$, by using the induction hypothesis. But also we have that $G(a::l') \equiv G(\text{Unit}(a) @ l') \equiv P(G(\text{Unit}(a)), G(l')) \equiv P(f(a), G(l'))$, using the "Unit" lemma, and the assumption that G also extends f to all of $(\alpha)\text{List}$. Hence we have shown that $FM(a::l') \equiv G(a::l')$ for each $a: \alpha$ and $l: (\alpha)\text{List}$, and completes the first proof of the uniqueness theorem.

Proof (B) (by monoid induction on l)

The Base cases corresponding to putting $l \equiv l$ or $l \equiv \text{Nil}$ are trivial and follow exactly as before. The interesting cases are those that replace the "Cons" case.

Suppose that $l \equiv \text{Unit}(a)$, for $a: \alpha$. So, by Lemma 3.4, we have that $FM(\text{Unit}(a)) \equiv (FM(Z, P)) (f) (\text{Unit}(a)) \equiv f(a) \equiv G(\text{Unit}(a))$, since (Z, P) is a monoid on β . Also, $G(\text{Unit}(a)) \equiv f(a)$ (by the assumption that G extends f).

Suppose that $l \equiv (l_1 @ l_2)$ with the two induction assumptions:-

$$\begin{aligned} FM(l_1) &\equiv G(l_1) \\ FM(l_2) &\equiv G(l_2) \end{aligned}$$

So, by calculation, $FM(l_1 @ l_2) = P(FM(l_1), FM(l_2)) = P(G(l_1), G(l_2)) = G(l_1 @ l_2)$, using both induction hypotheses and that both FM and G are monoid morphisms.

QED

This theorem completes the informal presentation of the result that $(Nil, @)$ is a free (left-strict) monoid on $(\alpha)List$ with respect to $Unit: \alpha \rightarrow (\alpha)List$, with arbitrary valuation morphisms from CPO . The statement of this overall result is the conjunction of the following lemmas:-

- (1) $(Nil, @)$ is a monoid on $(\alpha)List$
- (2) if (Z, P) is a monoid on β , then
 - (a) $f \equiv (FM(Z, P) f) \circ Unit$, for any $f: \alpha \rightarrow \beta$
 - (b) $(FM(Z, P) f): (\alpha)List \rightarrow \beta$ is a monoid morphism from $(Nil, @)$ to (Z, P) .
 - (c) $(FM(Z, P) f)$ is the only such morphism.

3.2 Formalisation in PPLAMBDA.

In this section the informal development given in the last three sections is cast into the LCF framework. The formalisation presented tries to mirror the informal development of the proof structure, permitting the informal study to be used as a guide through the more detailed, formal presentation given below.

3.2.1 The Axiomatisation of Lists.

This section corresponds to the ground covered within Section 3.1.1 where the list domain operator and standard list manipulating functions are informally introduced. Much of this axiomatisation is generated by the package discussed in Section 5.1.1.

First of all, a type operator, "List", of arity 1, is introduced which corresponds to the domain operator "List" introduced in Section 3.1.1. Introduce an isomorphism pair "absList" and "repList" defined by:-

```
absList : (dot_1 + ( $\alpha$  # ( $\alpha$ )List)_1)  $\rightarrow$  ( $\alpha$ )List
repList : ( $\alpha$ )List  $\rightarrow$  (dot_1 + ( $\alpha$  # ( $\alpha$ )List)_1)
```

whose only essential properties are given by:-

```
'absList' ]-  $\forall$ abs. absList(repList(abs)) = abs
'repList' ]-  $\forall$ rep. repList(absList(rep)) = rep
```

This is sufficient to assert that $(\alpha)\text{List}$ denotes a domain that satisfies the given domain equation. However, it does not say that $(\alpha)\text{List}$ is "well-founded", i.e. that structural induction is valid. This is stated later on, using however, this assertion is more easily expressed once a few more constants are available and so it is deferred to the end of this section.

Figure 3.2 below gives a table of the PPLAMBDA definitions of the basic list manipulating constants. Their definitions make use of the basic properties of summand injection, selection, product projection and pairing. Note that the "Cons" function has been "curried", which involves reading terms of the form "(Cons a l)" in place of "Cons(a,l)". Note that the formal distinction between these two forms has been respected.

Lemma 3.7

```
'absListUU' ]- "absList(l) = l"
'repListUU' ]- "repList(l) = l"
```

Proof

Recall lemma 2.1, which is alpha-convertible to:-

```
] - "∀ g:(α → β) f:(β → α). (∀ x:α. g(f(x)) = x) ⊃ g(l) = l"
```

and is known as fact 'gUU' on theory KERNEL. To produce the lemma absListUU, first simultaneously instantiate types by:-

```
"α"  ⇒  ":(dot_1 + (α # (α)List))_1"
"β"  ⇒  ":(α)List"
```

```
Nil  : (α)List
Cons : α → (α)List → (α)List
Null : (α)List → tr
Head : (α)List → α
Tail : (α)List → (α)List
```

```
'Nil' ]- "Nil = absList(INL(UP( )))"
'Cons' ]- "∀ a l. Cons a l = absList(INR(UP<a,l>))"
'Null' ]- "∀ l. Null(l) = ISL(repList l)"
'Head' ]- "∀ l. Head(l) = FST(DOWN(OUTR(repList l)))"
'Tail' ]- "∀ l. Tail(l) = SND(DOWN(OUTR(repList l)))"
```

FIGURE 3.2

Next, specialise quantifiers twice, so that "g" becomes "absList" and "f" becomes "repList". The result is then:-

$$\text{]- } (\forall x. \text{absList}(\text{repList}(x)) = x) \supset \text{absList}(1) = 1$$

The antecedent is now alpha-convertible to axiom 'absList', and so by an application of Modus Ponens (i.e. MP), this gives Lemma 'absListUU'. A similar derivation produces 'repListUU', and is omitted here.

QED

Figure 3.3 below gives a fragment of ML text whose evaluation simulates the above argument. It is not my intention to cover each derivation in this much detail, but to give only typical arguments and some idea of the interaction with LCF required. This particular lemma is an example of the "forwards proof style" discussed in Section 1.4.1. The next result gives the definedness properties of the isomorphism pair:-

Lemma 3.8

'DEFabsList']- $\forall x. \partial(\text{absList } x) = \partial(x)$
 'DEFrepList']- $\forall y. \partial(\text{repList } y) = \partial(y)$

Proof

The proofs follow the same pattern, as for the previous lemma. An already known fact is type instantiated (as necessary), quantified variables specialised, and already known antecedents are eliminated, via Modus Ponens, to render the desired result. In this case, lemma 2.2 is used and is quoted below:

$$\text{]- } \forall g f. (\forall x. g(f x) = x) \ \& \ (\forall y. f(g y) = y) \supset \partial(f x) = \partial(x)$$

It is clearly sufficient to instantiate the above appropriately and then apply the isomorphism axiom to obtain either lemma.

QED

The previous two lemmas are necessary steps in proving basic properties of the primitive list operations from their domain-theoretic definitions.

```

let aty = ":( . )u + (*a # (*a)List)u"
and bty = ":(*a)List"
;;

let lemma = FACT 'KERNEL' 'gUU'
and absListax = AXIOM 'L' 'absList'
;;

let thm1 = INSTTYPE [":*" ,aty; ":",bty] lemma
in
let thm2 = (SPEC gtm (SPEC ftm thm1)
             where ftm = "absList: !aty → !bty"
             and   gtm = "repList: !bty → !aty")
in
    MP thm2 absListax
;;

```

FIGURE 3.3

Lemma 3.9

'DEFCons']- $\forall a \ 1. \ \partial(\text{Cons } a \ 1) \equiv \text{TT}$

Proof

This result is obtained by rewriting, or simplifying, the left-hand side term to deliver the term on the right hand side. The set of simplification rules, or simpset, needed contains the axiom 'Cons' and the fact 'DEFabsList' with the set of basic simplification rules, BASICSS. The derivation of 'DEFCons' now proceeds as follows:-

$\partial(\text{Cons } a \ 1) \equiv \partial(\text{absList}(\text{INR}(\text{UP}(a,1))))$	
$\equiv \partial(\text{INR}(\text{UP}(a,1)))$	(by axiom 'Cons')
$\equiv \partial(\text{UP}(a,1))$	(by fact 'DEFabsList')
$\equiv \text{TT}$	(by schema 'DEFCONV')
	(by schema 'DEFCONV')

QED

This derivation can be performed in LCF using the following ML text:-


```

% Get theorems for simpset %

let thm1 = [(AXIOM 'L' 'Cons'); (FACT 'L' 'DEFabsList')]
;;

% Make simpset from thm1 and basic simpset, BASICSS %

let ssl = itlist ssadd thm1 BASICSS
;;

% Invoke simplification of given term with simpset %

simplterm ssl "θ(Cons !a !l)"
  where a = "a:*a"
  and l = "l:(*a)List"
;;

```

The next lemma states a number of other basic properties:-

Lemma 3.10

```

'DEFNil'      ]- θ(Nil) ≡ TT
'NullUU'      ]- Null(l) ≡ l
'HeadUU'      ]- Head(l) ≡ l
'TailUU'      ]- Tail(l) ≡ l
'NullNil'     ]- Null(Nil) ≡ TT
'NullCons'    ]- ∀a l. Null(Cons a l) ≡ FF
'HeadCons'    ]- ∀a l. Head(Cons a l) ≡ a
'TailCons'    ]- ∀a l. Tail(Cons a l) ≡ l

```

Proof

Each of these Lemmas can be demonstrated by equational reduction with respect to some suitable simpset, and as such can be proven mechanically using a similar pattern of the proof for the previous lemma.

QED

None of the previous lemmas required structural induction over lists, and so are valid in domains which satisfy the domain equation, but which may not be well-founded. The well-foundedness of the list domain is now asserted, as promised. As already discussed in Section 2.1.6, the "copyList" functional is introduced:-

$\text{copyList} : ((\alpha)\text{List} \rightarrow (\alpha)\text{List}) \rightarrow (\alpha)\text{List} \rightarrow (\alpha)\text{List}$

defined as follows:-

'copyList']- " $\forall f:(\alpha)\text{List} \rightarrow (\alpha)\text{List} \quad l:(\alpha)\text{List}.$
 $\text{copyList } (f) (l) \equiv$
 $(\text{Null } l) \Rightarrow l \mid \text{Cons } (\text{Head } l) (f (\text{Tail } l))"$

The well-foundedness axiom is then:-

'FIXList']- " $\forall l. \text{FIX } (\text{copyList}) (l) \equiv l"$

The next few lemmas show how constructor, selectors and the discriminator functions are related, and as such could be used as a basis for a structural "case-analysis" rule over lists. The proofs given here are special in that they make strong use of the well-foundedness property and the functional copyList; the "case analysis" rule actually arises as a result of the use of the sum operator in the domain equation, and is in fact valid for every solution of that equation, as mentioned in Section 2.1.4.

Lemma 3.11

'coverList']- " $\forall l:(\alpha)\text{List}.$
 $(\text{Null } l) \Rightarrow \text{Nil} \mid \text{Cons}(\text{Head } l)(\text{Tail } l) \equiv l"$

Proof

First, apply the PPLAMBDA rule FIXPT to the term "copyList" to give:-

] - $\text{FIX copyList} \equiv \text{copyList } (\text{FIX copyList})$

Now apply the term "l" to both sides of the above equation, by using APTHM, to give:-

] - $\text{FIX copyList } l \equiv \text{copyList } (\text{FIX copyList}) l$

Next, simplify the above theorem (as a whole) via an application of SIMP with the definition of copyList and the axiom 'FIXLIST' as simplification rules. After an application of the symmetry rule SYM, the desired result is obtained.

QED

The above lemma forms a key step in the proofs of each of the two "decomposition" lemmas given below:-

Lemma 3.12

'NullIMPNil']- $\forall l. \text{Null}(l) \equiv \text{TT} \supset l \equiv \text{Nil}$
 'NullIMPCons']- $\forall l. \text{Null}(l) \equiv \text{FF} \supset$
 $l \equiv \text{Cons}(\text{Head } l)(\text{Tail } l)$

Proof

The proof of 'NullIMPCons' is given here, as the proof of 'NullIMPNil' is similar. Assume that " $\text{Null}(l) \equiv \text{FF}$ ", and use this equational assumption as a simplification rule. Recall, that from Section 1.4.2, if this is ever engaged during a simplification proof, the formula " $\text{Null}(l) \equiv \text{FF}$ " will appear among any hypotheses of the resulting theorem.

Returning to the present proof, take lemma 'coverList', and simplify it using the assumption and BASICSS, to give:-

.]- $\text{Cons}(\text{Head } l)(\text{Tail } l) \equiv l$

By applying SYM, then the hypothesis discharge rule DISCH to bring the sole hypothesis into the conclusion, the result is obtained, after universally closing up any free variables.

QED

This lemma completes the discussion of the LCF theory 'L' for this case study.

3.2.2 Formalising concatenation and Unit.

The next theory, 'LFUN', contains a formal development of the Unit and concatenation functions informally discussed in Section 3.1.2. and is based upon the previous theory, L. These functions are introduced by the PPLAMBDA axioms displayed in Figure 3.4 below.

The first lemmas to be proven below provide non-recursive calculation rules for concatenation. The simplifier loops if the original defining equation for concatenation is ever engaged directly as a simprule. This is because the lhs is a general instance of concatenation which matches the occurrences introduced by the rhs. Hence, this situation recurs for any recursively defined function, given by a single equation. The solution is to break the definition into several separate equations so that each left-hand-side cannot match on the right-hand-side. A further

```

Unit  : $\alpha \rightarrow (\alpha)\text{List}$ 
- @ - : $(\alpha)\text{List} \# (\alpha)\text{List} \rightarrow (\alpha)\text{List}$ 

'Unit'    ]- "Va: $\alpha$ . Unit(a) = Cons a Nil"

'Append'  ]- " $\forall l_1 l_2$ .
              ( $l_1 @ l_2$ ) =
              (Null  $l_1$ )  $\Rightarrow l_2$  |
              Cons (Head  $l_1$ ) ((Tail  $l_1$ ) @  $l_2$ )"
```

FIGURE 3.4

requirement for such a set of computation rules is that any ground term in which the function is applied can be rewritten, using the calculation rules, to a ground term not containing occurrences of the function (cf. sufficient completeness [Guttag75]).

Lemma 3.13

```

'UUApp'    ]-  $\forall l_2. l @ l_2 = l$ 
'NilApp'   ]-  $\forall l_2. \text{Nil} @ l_2 = l_2$ 
'ConsApp'  ]-  $\forall l_2. (\text{Cons } a \ l) @ l_2 = \text{Cons } a \ (l @ l_2)$ 
```

Proof

A temporary set of rules used is obtained by specialising the variable l_1 to order to produce the following theorems:-

```

(a) ]- " $\forall l_2. l @ l_2 =$ 
        (Null  $l$ )  $\Rightarrow l_2$  | Cons(Head  $l$ )((Tail  $l$ ) @  $l_2$ )"
```

```

(b) ]- " $\forall l_2. \text{Nil} @ l_2 = (\text{Null Nil}) \Rightarrow l_2$  |
        Cons(Head Nil)((Tail Nil) @  $l_2$ )"
```

```

(c) ]- " $\forall l_2. (\text{Cons } a \ l) @ l_2 = (\text{Null}(\text{Cons } a \ l)) \Rightarrow l_2$  |
        Cons (Head(Cons  $a \ l$  ))
        ((Tail(Cons  $a \ l$ )) @  $l_2$ )"
```

By simplifying the above equations using various properties of Null, Head and Tail (e.g. 'NullUU', 'HeadCons' etc), the desired equations stated above are obtained.

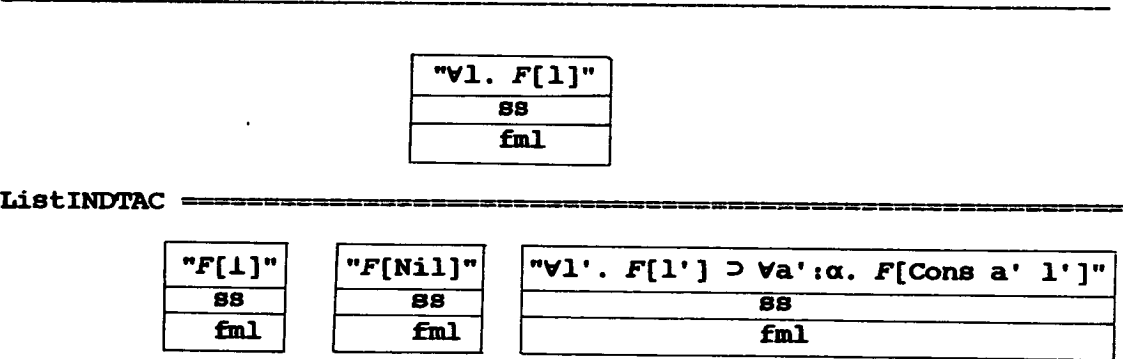
QED

These three equations are used as simplification (i.e. computation) rules for the concatenation function. The next couple of lemmas mark the first application of structural induction over lists, and also mark the first serious use of a tactic to generate a formal proof in this Chapter.

So, by way of preparation for the discussion of the lemmas themselves, a couple of list induction tactics, `ListTAC` and `ListTAC'`, are introduced. Both of these use schematically generated structural list induction tactic, called `ListINDTAC`, to generate appropriate subgoals. Each of the tactics `ListTAC` and `ListTAC'` then transform the inductive step case into a more convenient form in slightly different ways. The behaviour of `ListINDTAC` is diagrammed in Figure 3.5 below, using Cohn's tactic notation.

This is the simplest and most direct form of the list induction tactic. Note that it introduces 3 subgoals corresponding to the antecedents of the list induction rule. However, the inductive step case, while in the most general form, is not the form most appropriate for its application here. It is usually easier to prove the step goal, by "assuming" the induction hypothesis (ie. by placing it into the goal's assumption list) and then showing that the consequent goal follows from this collection of assumptions. Such a proof will, if induction was necessary, make use of the induction hypothesis in some way. Now, since it may turn out that the induction hypothesis can be conveniently used as a *simprule*, the two tactical variants are used to treat this induction hypothesis as a *simprule*.

The first task performed by `ListTAC` is to obtain the induction subgoals by applying `ListINDTAC`. It then applies `GENTAC` followed by `IMPTAC` to the step goal. This first specialises the leading quantified variable "l", (taking care to avoid any existing free



(where $F[l]$ is a PPLAMBDA formula, admissible in free variable "l")

FIGURE 3.5

variables in the goal's assumption list) leaving an implicative goal. The application of IMPTAC places the antecedent into the goal's assumption list, leaving the consequent as the resulting goal formula. ListTAC is expressed in ML as follows:-

```
let ListTAC =
  ListINDTAC THENL [IDTAC; IDTAC; (GENTAC THEN IMPTAC)]
;;
```

The overall effect of ListTAC is diagrammed below:-

" $\forall l. F[l]$ "
ss
fml

ListTAC

" $F[l]$ "
ss
fml

" $F[Nil]$ "
ss
fml

" $\forall a:\alpha. F[Cons\ a\ l']$ "
ss
" $F[l']$ " . fml

where l' is a variable that does not occur free in the formula fml.

The difference between the tactic's ListTAC and ListTAC' is that ListTAC' attempts to include the induction hypothesis in the step goal's simpset. The tactic can be defined in ML in a similar way to ListTAC by using IMPTAC' instead of IMPTAC:-

```
let ListTAC' =
  ListINDTAC THENL [IDTAC; IDTAC; (GENTAC THEN IMPTAC')]
;;
```

Recall from Section 1.4.4 that IMPTAC' adds the antecedent of the goal as an ASSUME'd simprule to the local simpset (if possible) as well as placing the antecedent in the assumption list. Both IMPTAC and IMPTAC' fail if the goal is not an implication of some sort. For completeness, the effect of the tactic ListTAC' is given below:-

" $\forall l. F[l]$ "
ss
fml

ListTAC'

" $F[l]$ "
ss
fml

" $F[Nil]$ "
ss
fml

" $\forall a:\alpha. F[Cons\ a\ l']$ "
. $\vdash$ " $F[l']$ " $\cup$ ss
" $F[l']$ " . fml

where l' is a variable that does not occur free in the formula fml .

Having thus added these tools for performing list induction proofs, we may now return to the discussion of properties of the concatenation function.

Lemma 3.14

'AppNil' $\vdash \forall l. l \in Nil \Rightarrow l = []$
 'Assoc' $\vdash \forall l_1 l_2 l_3. (l_1 \in l_2) \in l_3 \Rightarrow l_1 \in (l_2 \in l_3)$

Proof

Appeal to list induction, and then simplify the result, using the induction hypothesis in the step case. This proof recipe corresponds to the simple tactic:-

ListTAC' THEN SIMPTAC

The initial goal presented to the tactic consists, in each case, of the goal formula (ie the body of the lemma to be proven), some suitably chosen simpset, and an empty list of assumptions. In the case of the two lemmas to be proven above, the same simpset can be used. This simpset consists of the 3 computational equations for concatenation, formally proved as Lemma 3.13 together with BASICSS.

QED

Lemma 3.15 (The "ConsUnit" Lemma).

'ConsUnit' $\vdash \forall a l. Cons a l \Rightarrow (Unit a) \in l$

Proof

By simplification. This amounts to just applying the tactic SIMPTAC to a goal formed by the body of the lemma to be proven, an appropriate simpset^{*}, and an initially empty list of assumptions.

QED

3.2.3 Formalising the Freeness Functional.

Having completed the formal discussion of the basic monoid operations, we proceed to a detailed formulation of the freeness result; this section formalises the results of Section 3.1.3. A fresh LCF theory, called LFREE, is initiated and is based upon the

* The simpset consists of equational properties of Head, Tail etc, plus Lemmas 3.13 and 3.14, and the definition of Unit.

Proof

As for Lemma 3.13, instantiate the definition accordingly and then simplify each of these with a standard simpset for the list theory, 'L', containing properties of "Null", "Head" and "Tail".

QED

To state the next lemma concisely, another meta-language (ie syntactic) abbreviation is introduced; this states that, given terms of the form " $\uparrow z:\uparrow\theta$ " and " $\uparrow p:(\uparrow\theta \# \uparrow\theta \rightarrow \uparrow\theta)$ ", they are a (left-strict) monoid, on carrier " $\uparrow\theta$ ". Each occurrence of the term parameters z , p and type parameter θ , may be substituted to give the desired assertion. In this case, the object type parameter θ can be determined from the terms z and p given and so need not be passed explicitly. This assertion is constructed using an ML function IsMonoid :term # term $\rightarrow$ form list, defined below in Figure 3.6. Note that it explicitly checks that the types of the parameters correspond to a monoid signature.

It is a matter of convenience that IsMonoid produces a list of four equations rather than their conjunction, as it is easier to build a simpset containing these rules, produced in this form. Such

```

let IsMonoid(z:term, f:term) =
  let theta = typeof(z) in
    if (typeof(p) = ":\uparrowtheta # \uparrowtheta \rightarrow \uparrowtheta") then
      let a = "a:\uparrowtheta"
      and b = "b:\uparrowtheta"
      and c = "c:\uparrowtheta"
      in
        [ "Va. \uparrow p(1:\uparrowtheta,a) \equiv 1:\uparrowtheta"           %Left-strictness%
          ; "Va. \uparrow p(\uparrow z,a) \equiv a"                %Left-identity%
          ; "Va. \uparrow p(a,\uparrow z) \equiv a"                %Right-identity%
          ; "Va b c. \uparrow p(\uparrow p(a,b),c) \equiv \uparrow p(a,\uparrow p(b,c))" %Associativity%
        ]
      else (failwith 'Bad Signature')
  ;;

```

% Note that anti-quotation of variables a,b,c has been suppressed in the above for clarity. %

FIGURE 3.6

a simpset may be constructed using the following ML expression:-

```
(ssaddl o (map ASSUME) o IsMonoid) (z,p)
  where ssaddl(thl) = itlist ssadd thl BASICSS
;;
```

The function IsMonoid is applied to (z,p), producing a list of formulae. Next, each formula in the list is ASSUME'd, producing a :thm list. Finally, this is passed to the auxiliary function ssaddl forming the simpset from the :thm list, based upon BASICSS.

The next two lemmas will make use of the assumption that " $Z : \beta$ " and " $P : \beta^2 \rightarrow \beta$ " form a monoid and each of the component assumptions are made available as simprules.

Lemma 3.17

'FreeMonUnit' .]- " $\forall f a. !FMtm(\text{Unit } a) = f(a)$ "

Proof

The initial goal is:-

< " $\forall f a. !FMtm(\text{Unit } a) = f(a)$ ", ss , IsMonoid("Z","P") >

where the simpset ss contains the following rules:-

- The basic list-theory simpset as for Lemma 3.15.
- 'FreeMonNil', 'FreeMonCons'; the "Nil" and "Cons" cases of the FreeMonoid functional definition.
- each of the formulae in IsMonoid("Z","P") as permitted assumptions.

The tactic used is SIMPTAC. Now, the derivation then proceeds to apply the following simprules (in sequence):- 'Unit'; 'FreeMonCons'; 'FreeMonNil'; and then finally the "monoid" assumption of right identity, . $\vdash \forall x. P(x,Z) = x$.

Note that only the assumption of right identity appears in the hypotheses of the result, since it was the only simprule to be engaged with non-empty hypotheses.

QED

The next lemma formally shows that the FreeMonoid functional induces a monoid morphism, when given appropriate parameters, and formalises Lemma 3.5:-

Lemma 3.18

'FreeMonAppend'

...]- " $\forall l_1 l_2. \text{FMtm}(l_1 @ l_2) \equiv P(\text{FMtm}(l_1), \text{FMtm}(l_2))$ "

Proof

The initial goal follows the usual pattern; the goal formula corresponds to the body of the lemma above; the assumptions used are as for the previous lemma (that is, "Z" and "P" together form a monoid). The initial simpset contains the following rules:-

- The basic list-theory simpset as for Lemma 3.15.
- Each formula in the assumption list is ASSUME'd for use as a simprule (ie IsMonoid("Z","P")).
- The calculation rules for "Append" (ie 'UUApp', 'NilApp', 'ConsApp').
- The calculation rules for "FreeMonoid" (ie 'FreeMonUU', 'FreeMonNil', 'FreeMonCons').

The tactic applied is:-

ListTAC' THEN SIMPTAC

This first does list induction on the first quantified variable, l_1 , in the goal, and then simplifies with the resulting assumptions. Again, only some of the assumptions linking "Z" and "P" are used. These are the left-strictness of "P", that "Z" is a left identity for "P" and finally that "P" is associative. As in the previous lemma these assumptions will appear among the hypotheses of the result.

QED

Before the final lemma is discussed, another meta-linguistic syntactic abbreviation for a collection of assumptions is required. This abbreviation asserts that a given function is a strict monoid morphism (with respect to given monoids on the appropriate carriers).

Suppose that $f:\tau_1 \rightarrow \tau_2$ for some types τ_1 and τ_2 , and also let $Z_1:\tau_1$, $P_1:\tau_1^2 \rightarrow \tau_1$, $Z_2:\tau_2$, and $P_2:\tau_2^2 \rightarrow \tau_2$. Then, the assertion that $G:\tau_1 \rightarrow \tau_2$ is a strict morphism from (Z_1, P_1) to (Z_2, P_2) is true whenever the following equations hold:-

- (1) $G(1:\tau_1) \equiv 1$
- (2) $G(Z_1) \equiv Z_2$
- (3) $\forall x_1 x_2:\tau_1. G(P_1(x_1, x_2)) \equiv P_2(G(x_1), G(x_2))$

First of all, note that this definition does not (at least for well-formedness) depend on either algebra actually being a monoid. In fact, this definition is identical to that for the anarchic case. So, in particular, it is unnecessary to constrain the definition further to ensure that either algebra is a monoid

As for the previous abbreviation IsMonoid, an ML function is given for calculating the body of the assertion. This function is called IsHom : ((term # term) # term # term) $\rightarrow$ term $\rightarrow$ form list, and this is described in Figure 3.7 below. The parameters Z_1 , P_1 , Z_2 , P_2 and G all have meta-type term, and for successful application, must also have the object types implied by the well-formedness of each of the resulting formulae.

The remaining lemma to formally prove is the Uniqueness theorem, corresponding to Theorem 3.6. The two proofs of this

```

let IsHom ((Z1,P1) (Z2,P2)) (G) =
  (let ty1 = typeof Z1
   and ty2 = typeof Z2
   in
    let fm1 = "!(G(1:ty1) = 1:ty2)"
    and fm2 = "!(G(Z1) = Z2)"
    and fm3 = "\x1 x2:ty1.
                  !(G(P1(x1,x2) = P2(G(x1), G(x2)))"
    in
      [fm1; fm2; fm3]
  ) ? failwith 'Bad type for homomorphism'
;;

```

FIGURE 3.7

result given there are also repeated here; one using standard structural induction over lists and the other using monoid induction. However, in order to do this, a basic monoid induction tactic called MONOIDTAC is introduced below. The derivation performed by the proof component of MONOIDTAC is not special or particular to the (polymorphic) domain "List", and the same proof component is used for each of the List-like domains used in these case studies. The calculation used in the proof component is based on the proof of Lemma 3.3, which reduces monoid induction to the case of structural induction over lists:-

" $\forall l. F[l]$ "
ss
fml

MONOIDTAC =====
 [subgoal1; subgoal2; subgoal3; subgoal4]

where

subgoal	=	"F[1]"
		ss
		fml

subgoal2	=	"F[Nil]"
		ss
		fml

subgoal3	=	" $\forall a:\alpha. F[Unit(a)]$ "
		ss
		fml

subgoal4	=	" $\forall l_1 l_2. F[l_1] \ \& \ F[l_2] \supset F[l_1 @ l_2]$ "
		ss
		fml

Each quantified variable introduced into a subgoal is freshly chosen to avoid any free variables appearing in either the goal's formula or assumption list. Also, the induction step subgoal is stated in a form appropriate for arbitrary tactic processing. However, as for ListTAC and ListTAC', it is convenient here to strip off the two quantifiers (by using GENTAC) and move the antecedents into the assumption list, possibly adding them as simprules to the local simpset (by using IMPTAC'). This extra processing is easily arranged by using the tactic MONTAC' defined below:-

```

let MONTAC' = MONOIDTAC THENL
  [ IDTAC
  ; IDTAC
  ; IDTAC
  ; (REPEAT GENTAC) THEN (REPEAT IMPTAC')
  ]
;;

```

There is also a version of MONTAC' which does not add induction hypotheses to goal simpsets, called MONTAC. This is not used later on; its ML definition is as above for MONTAC', but with IMPTAC instead of IMPTAC'.

We now turn to the tactical proof(s) of the Uniqueness theorem. There are two proofs given here, corresponding to those for Theorem 3.6. The initial goals for each proof differ in that the simpset for the second proof contains one less simprule; both goals have the same goal formula and the same (eight) assumptions as stated below.

The set of simprules common to both proofs includes the basic simpset, BASICSS, together with each of the assumptions prescribed in the statement below, ASSUME'd to do duty as simprules. The final ingredients required for this simpset are the theorems 'FreeMonUU', 'FreeMonNil', 'FreeMonUnit' and 'FreeMonAppend'. The first two of these theorems are simple quantified equations and hence are straightforward simprules. However, the second pair of theorems (in sentential form) possess non-trivial antecedents. For example, the theorem 'FreeMonUnit' is:-

]- "VZ P f. ($\forall x. P(x, Z) \equiv Z$) $\supset \forall a:\alpha. !FMtm(\text{Unit } a) \equiv f(a)$ "

This can be used as a conditional simplification rule (see Section 1.4.2).

Theorem 3.19

Assume the following sets of formulae:-

- (1) IsMonoid("Z: β ", "P: $\beta^2 \rightarrow \beta$ ");
i.e. "Z" and "P" form a monoid on the carrier " β ".
- (2) IsHom(("Nil", "e"), ("Z", "P")) ("G:(α)List $\rightarrow \beta$ ");
i.e. "G" is a strict monoid morphism, from

("Nil", "e") to ("Z", "P").

(3) "f $\equiv \lambda a:\alpha. G(\text{Unit } a)$ ";

i.e. "G" extends "f", via "Unit".

Then, we have that:-

'FreeMonUnique']- "Vl. !FMtm(1) $\equiv G(1)$ "

Tactical proof (A)

Take the goal as described above, and include the 'ConsUnit' lemma (proven as Lemma 3.15) as a simprule. Apply the following tactic:-

SIMPTAC THEN ListTAC' THEN SIMPTAC

The goal is initially:-

"Vl. FreeMonoid (Z,P) (f) (1) $\equiv G(1)$ "

After applying simplification, it becomes:-

"Vl. FreeMonoid (Z,P) ($\lambda a. G(\text{Unit } a)$) (1) $\equiv G(1)$ "

The only simprule that is applicable is the assumption that "G extends f" expressed in the form "f $\equiv \lambda a. G(\text{Unit}(a))$ "; this formulation is used to permit f to be rewritten, even when f occurs as an argument to a function.

To prevent successive formulae becoming unwieldy in this proof, introduce FMtm' = "FreeMonoid (Z,P) ($\lambda a. G(\text{Unit}(a))$)". So, using this new abbreviation, the goal may be written as:-

"Vl. !FMtm'(1) $\equiv G(1)$ "

The goal now has a form suitable for list induction to be applied, via ListTAC'. As usual, this gives three subgoals, the first two of which correspond to the base cases and are easily dealt with by SIMPTAC. The third subgoal generated corresponds to the induction step; the induction hypothesis " $!FMtm'(1') \equiv G(1')$ " (for the variable " $1':(\alpha)\text{List}$ ") is included as an assumption and made into a simprule. This case also goes through by simplification, but by using a (non-deterministic) sequence of reductions which may be different to that given in the first proof of Theorem 3.6. One

such sequence is as follows:-

$$\text{"!FMtm' (Cons a l) = G(Cons a l)"}$$

The simplifier now applies the simprule 'ConsUnit' to both sides, simultaneously, giving:-

$$\text{"!FMtm' ((Unit a) @ l) = G((Unit a) @ l)"}$$

Now, by applying the conditional simprule 'FreeMonAppend' (by using monoid assumptions on "Z" and "P" to do so) giving:-

$$\text{"P(!FMtm'(Unit a), (!FMtm' l)) = G((Unit a) @ l)"}$$

Next, apply the morphism assumptions about G, and the conditional simprule 'FreeMonUnit' (plus a basic β -conversion from BASICSS) to obtain:-

$$\text{"P(G(Unit a), (!FMtm' l)) = P(G(Unit a), G(l))"}$$

Finally, apply the induction hypothesis (as a simprule) to give the (trivial) goal:-

$$\text{"P(G(Unit a), G(l)) = P(G(Unit a), G(l))"}$$

This completes the first tactical proof.

QED(A)

If the "G extends f" assumption had been expressed more straight-forwardly here by " $\forall a. f(a) = G(\text{Unit } a)$ " instead, it then turns out that the "induction then simplification" tactic does solve the goal. However, this will not be the case for the other two case studies; the tactical proof given above (with corresponding simpset) does also apply to the other studies to be presented. This is because there will be extra strictness conditions that need to be proven about "f". These can be proven directly from the assumed simprule " $f = \lambda a. G(\text{Unit } a)$ " only using simplification and the fact that, in those case studies, the "Unit" function is strict.

The initial simplification above was used, effectively, to substitute for "f" in the goal to ensure that "f" did not occur in the induction hypothesis. (This could also have been applied using

the basic tactic SUBSTAC, see [LCF], pp 140). This elimination of "f" was necessary for the following reason.

Suppose that no rewriting is done before the induction tactic is applied. In that case, there are matching occurrences of "f" within the induction hypothesis. These occurrences would have to be matched in order that the induction hypothesis could be applied as a *simprule*. Now, the G extends f assumption, used as a *simprule*, eliminates all occurrences of "f" appearing in the goal. But this would then block any application of the induction hypothesis. Hence, for the proof to succeed by simplification, it is necessary for the induction hypothesis to be used before the G extends f assumption. This kind of condition cannot be guaranteed when using simplification.

This situation arises because the *simpset* resulting from adding the induction hypothesis is not confluent, (ie the *simpset* is not Church-Rosser). See [Huet80] for a general discussion of the confluence of rewriting systems. It is known that the confluence of such systems is not a decidable property in general.

Finally, note that each assumption made was eventually used in this proof. The morphism assumption on "G" were used directly in each appropriate case generated by induction; for example the strictness of G is used in the "1" case. The monoid assumptions on "Z" and "P" were used indirectly to justify the application of the conditional rewrite rules 'FreeMonUnit' and 'FreeMonAppend'.

Tactical proof (B)

The goal for this proof is as before, except that the 'ConsUnit' lemma is not included within the *simpset*. The proof is generated by applying the tactic:-

MONTAC' THEN SIMPTAC

Initially the goal is:-

" $\forall l. !FMtm(l) = G(l)$ "

After applying monoid induction, via MONTAC', four subgoals are obtained. The first two subgoals are the standard base cases and go through in the usual way by simplification. The remaining

subgoals correspond to the "Unit" and the (inductive) concatenation cases. The Unit case is:-

$$\text{"Va. !FMtm(Unit a) = G(Unit a)}$$

So, using the conditional simprule 'FreeMonUnit' (after proving the antecedents) this produces:-

$$\text{"Va. f(a) = G(Unit a)}$$

But now, by applying the assumption that G extends f, this gives (after a β -conversion from BASICSS) the (trivial) goal:-

$$\text{"Va. G(Unit a) = G(Unit a)"}$$

The concatenation subgoal is:-

$$\text{"!FMtm(l_1 @ l_2) = G(l_1 @ l_2)"}$$

with the induction hypotheses " $\text{!FMtm}(l_1) = G(l_1)$ " and " $\text{!FMtm}(l_2) = G(l_2)$ ", available as simprules. Apply the 'FreeMonAppend' lemma to the lhs of the goal to get:-

$$\text{"P(!FMtm}(l_1), \text{!FMtm}(l_2)) = P(G(l_1), G(l_2))\text{"}$$

Finally, apply both induction hypotheses on the lhs to give the usual trivial goal. This completes the second tactical proof of the Uniqueness Theorem.

QED(B)

It may appear, at first sight, that the use of the 'ConsUnit' lemma has been avoided. This is not the case since it forms an essential part of the proof component for the basic monoid induction tactics, MONOIDTAC, from which MONTAC' is constructed.

The second proof has a simpler structure than the first. This is due to the simpler form that the concatenation and FreeMonoid functions possess when inductively defined over a monoid structure. In effect, we have derived the monoid-oriented definition (i.e. the 'FreeMonUnit' and 'FreeMonAppend' lemmas) from the original "Cons" oriented "one element at a time" definition (i.e. the 'FreeMonCons' lemma). The monoid induction principle then provides exactly the right pattern of arguments to permit precisely these lemmas to be used effectively. Note that the introduction of induction

hypotheses as *simprules* did not give rise to any lack of confluence, as had happened in the previous proof.

One of the issues addressed by these case studies concerns the observation that anarchic algebraic structures possessing freely generated term models, such as "lists" and "binary trees" are easily constructable (from a computational point of view). Other data-types such as "sets" and "multisets" can be derived from these by "quotienting". This case study has shown that it may occasionally be possible to use an anarchic algebra to represent another (non-anarchic) algebra directly, without any need for quotienting.

3.3 The Second Case Study.

This case study is concerned with the effect on the formalisation and proof structure given above when the valuation functions and the "Unit" function are assumed to be strict. This amounts to taking the construction with respect to CPO_1 .

It is perhaps surprising that the construction given in the first case study does not also satisfy the modified freeness criterion. Even if the Unit function used previously is "strictified", the domain of the free monoid given contains "irrelevant" elements. There exist strict continuous valuations, such that there is more than one monoid morphism making the appropriate diagram commute using this "strictified" unit function. To see this, consider the following example:-

Take the domain of generators to be one = $\{T, \perp\}$, and define the "strictified" unit function, "Unit': $\alpha \rightarrow (\alpha)List$ " as follows:-

$$\begin{aligned} \text{Unit}'(a) &\equiv (\partial(a) \Rightarrow \text{Unit}(a) \mid \perp) \\ &\equiv (\partial(a) \Rightarrow (\text{Cons } a \text{ Nil}) \mid \perp) \end{aligned}$$

Consider the (left-strict) monoid $(TT, \text{and}; (tr^2 \rightarrow tr))$ on carrier domain, tr . The function and is the conditional conjunction as defined in Section 2.1.2.

Define the strict valuation function $v: \text{one} \rightarrow tr$ by $v(T) = FF$, and consider the two functions $H1, H2: (\text{one})List \rightarrow tr$ defined by:-
 "H1 = FreeMonoid $\langle TT, \text{and} \rangle (v)$ " and also, "H2 = Null". Now, it turns out that both H1 and H2 are monoid morphisms from $\langle Nil, @ \rangle$ to

(TT,and); this result for H2 strongly relies on the fact that and is not right-strict. Moreover, we have that " $v \equiv (H1 \circ \text{Unit}') \equiv (H2 \circ \text{Unit}')$ " as required. Finally, note that H1 is not equal to H2, since we have that " $H1(\text{Cons } 1 \text{ Nil}) \equiv 1$ ", whereas " $H2(\text{Cons } 1 \text{ Nil}) \equiv \text{FF}$ ". Hence, the construction given in the first case study is not a free monoid construction with respect to CP01 (i.e. strict unit function and strict valuation morphisms).

This example shows that the question of whether or not some universal construction exists need not be as obvious as might be first thought, and that it is a fitting candidate for formal proof attempts.

A main objective of the remaining two studies is to chart the consequences for formalisation and formal proof, given various strictness requirements. These consequences are reported in terms of the differences between the present case study and the corresponding part of the first case study. Generally speaking, the differences are most apparent in the choice of list domain and the introduction of more assumptions concerning strictness generally. Because of these extra assumptions and properties, further case analyses on the definedness of terms will generally need to be introduced.

The hierarchical structure of theories remains the same as before although, for clarity, their names have been changed. So, in this case study, the list domain is axiomatised on theory L1, (based on theory BASIC); the unit and concatenation functions are defined in LFUN1 and the freeness functional in theory LPREE1.

However, we shall assume that quantities in the present study corresponding to quantities in the first study are given identical names here. This refers not only to the names of constants and lemmas, but also to tactics, schemas and abbreviations (e.g. MONOIDTAC, IsMonoid and FMtm). Any differences between such quantities are discussed as and where they arise.

3.3.1 The Theory L1.

The domain of lists needed here requires a "Cons" operation which is strict in its first but not its second argument. This is

necessary if the Unit function is to be strict and to be defined as in the first study:-

Unit(a) = Cons a Nil

So, an appropriate choice of a (polymorphic) domain equation is:-

$$(\alpha)\text{List1} \cong \underline{\text{dot}}_1 + (\alpha \otimes ((\alpha)\text{List1})_1)$$

This is axiomatised in the LCF theory L1, in a similar fashion to the previous domain operator using the LCF package described in section 5.1.1. So, as before, various basic constants are introduced which are as follows:-

```
absList1  : (dot1 + (α ⊗ ((α)List1)1)) → (α)List1
repList1  : (α)List1 → (dot1 + (α ⊗ ((α)List1)1))
copyList1 : ((α)List1 → (α)List1) → (α)List1 → (α)List1
Nil        : (α)List1
Cons       : α → (α)List1 → (α)List1
Head       : (α)List1 → α
Tail       : (α)List1 → (α)List1
Null       : (α)List1 → tr
```

The standard auxilliary functions eg. absList1, repList1 and copyList1, all have definition much as before. This includes the well-foundedness axiom:-

'FIXList1']- "Vx:(α)List1. FIX copyList1 x = x"

The differences in definitions that arise are mainly dictated by the form of the domain equation above. For example, the "Cons" operation makes use of strict pairing instead of standard Cartesian pairing and "lifts" its second curried argument. The definition of "Cons" here is:-

'Cons']- "Va l. Cons a l = absList1(INR<a ⊗ (UP l)>)"

Correspondingly, the definitions of "Head" and "Tail" use strict pair projections accordingly, and Tail needs to apply a "DOWN" function to "drop" the second component appropriately. These are:-

'Head']- "Vl. Head l = P1(OUTR(repList1 l))"

and also,

'Tail']- "Vl. Tail l = DOWN(P2(OUTR(repList1 l)))"

The definition of Null has the same form as previously.

There are, however, significant differences in theorems concerning lists. For example, consider the lemma 'TailCons'. In this theory of lists, it is not always true that "Tail(Cons a l) = l", purely because "Cons" is strict in its first argument. However, this can be proved on the assumption that "a" is defined. So, formally this is:-

'TailCons']- "Va l. $\partial(a) \equiv \text{TT} \supset \text{Tail}(\text{Cons } a \ l) = l$ "

This can be proven with the tactic:-

(REPEAT GENTAC) THEN IMPTAC' THEN SIMPTAC

This takes a suitable goal, strips off any leading quantifiers by repeated use of GENTAC; places the leading antecedent of the goal into its assumption list and adding it as a simprule to the simpset. Finally, simplification is applied to the remaining goal. (The tactic fails if the goal formula is not a possibly quantified implication.)

In this case, the theorem is proven by expanding the definitions of Tail and Cons in terms of primitive PPLAMBDA constants. The assumption that "a" is defined is used to simplify the antecedent of the conditional simprule for the second strict projection function "P2". The simpset used contains all of the definitions, plus BASICSS (enriched with standard properties of the strict projection and pairing functions).

The above lemma is a simple example of how definedness assumptions begin to permeate theorems whose analogous first case study theorems did not require any. A similar condition is required for the 'NullCons' lemma; that is

'NullCons']- "Va l. $\partial(a) \equiv \text{TT} \supset \text{Null}(\text{Cons } a \ l) \equiv \text{FF}$ "

The previous tactic deals with this goal as well.

The form of the 'HeadCons' theorem generalises from the first case study directly:-

'HeadCons']- "Va l. Head(Cons a l) = a"

It should also be possible to prove it directly with

simplification. But it happens that applying SIMPTAC to the appropriate goal is not successful. However, by taking the goal as above, and then removing the quantifiers enables SIMPTAC to solve the goal!

The initial failure to simplify the goal is due to a pragmatic restriction of the simplification algorithm when applying conditional simprules. To avoid trying to prove too many "unprovable" matches, the lhs of the consequent of a conditional simprule is only matched against subterms that are free in the entire term being rewritten. (See Appendix 8.1 of [LCF] for further details). In the original quantified form of 'HeadCons' each variable is bound, so blocking any conditional simplification. Hence, the tactic used to actually prove 'HeadCons' is:-

(REPEAT GENTAC) THEN SIMPTAC

(with the usual simpset of definitions and standard simprules included). Also note that, by simplification, there is the additional theorem:-

'UUCons']- " $\forall l. (\text{Cons } l \ l) = l$ "

confirming the strictness requirement upon the "Cons" operation. Note that, from the 'HeadCons' theorem, we have that

] - " $\forall a. \text{Head}(\text{Cons } a \ l) = a$ " also showing non-strictness in the second argument of "Cons". Incidentally, each of the case distinction laws analogous to 'NullIMPNil' and 'NullIMPCons' (i.e.

] - " $\forall l. \text{Null}(l) \equiv \text{FF} \supset l \equiv \text{Cons}(\text{Head } l)(\text{Nil } l)$ ") go through with no alteration. This is a consequence of the easily proven fact:-

'DEFNull']- " $\forall l. \partial(\text{Null } l) \equiv \partial(l)$ "

This completes the discussion of the theory L1.

3.3.2 The Theory LFUN1.

In this theory, the definitions of "Unit" and "Append" are given in the same way as for the first study. Since Append is defined recursively, this must be split up into the usual 3 calculation equations. The first two lemmas (analogous to 'UUApp'

and 'NilApp') go through as before, via simplification. However, the third computation rule, the 'ConsApp' lemma, requires a case analysis on the definedness of a variable.

'ConsApp']- " $\forall a \ l_1 \ l_2. (\text{Cons } a \ l_1) @ \ l_2 = \text{Cons } a \ (l_1 @ \ l_2)$ "

A new tactic is introduced, called GENDEF CASESTAC, which takes a quantified goal and performs a truthvalue case analysis upon the definedness of the quantified variable. This case analysis takes account of the knowledge that $\vdash \forall x:\alpha. \partial(x) \in \text{TT}$. This means that a truthvalue subgoal for the assumption that " $\partial(x) = \text{FF}$ " need not be explicitly generated since that assumption is already known to be contradictory. The tactic also adds each of the definedness assumptions to the corresponding assumption list; suitable simprules based on these assumptions are also included in each simpset. The behaviour of GENDEF CASESTAC is given in Figure 3.8 below.

The 'ConsApp' lemma is proven using the following tactic:-

GENDEF CASESTAC THEN SIMPTAC

A basic tactic is also needed for applying the structural induction principal for the present domain of lists. This tactic, called ListLINDTAC, is generated by using the LCF package described in Section 5.2.1.

However, this is not in its most useful form, and so another tactic is constructed which eliminates quantifiers, assumes the induction hypotheses and adds it to the simpset. It also does a definedness case analysis on the first "Cons" induction argument in

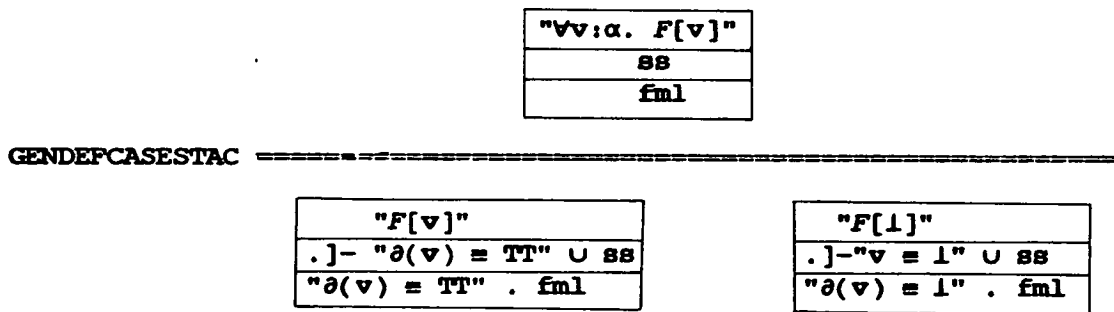


FIGURE 3.8

the step case. This more useful induction tactic is called `List1TAC'` and is defined in ML by:-

```
let List1TAC' =
  List1INDTAC THENL
  [IDTAC; IDTAC; (GENTAC THEN IMPTAC' THEN GENDEFCASESTAC)]
;;
```

This tactic behaves as follows:-

" $\forall l. F[l]$ "
ss
fml

`List1TAC'` =====
 [subgoal1; subgoal2; subgoal3; subgoal4]

where subgoal1 =

" $F[l]$ "
ss
fml

subgoal2 =

" $F[Nil]$ "
ss
fml

subgoal3 =

" $F[Cons\ l\ l']$ "
$\{.\}- "F[l']", .\}- "a = l"\} \cup ss$
" $F[l']$ " . " $a = l$ " . fml

subgoal4 =

" $F[Cons\ a\ l']$ "
$\{.\}- "F[l']", .\}- "\partial(a) \equiv TT"\} \cup ss$
" $F[l']$ " . " $\partial(a) \equiv TT$ " . fml

(where the variable " l' " is chosen not to occur freely in either the goal formula " $F[l]$ " or the assumption list, fml)

Since "Cons" is strict in its first argument, the third subgoal could always be reduced to the " l " case. In the application here, this goal is easily proven by simplification and no further special manipulation is applied to it here.

Both of the lemmas 'AppNil' and 'Assoc' are solved using the tactic:-

`List1TAC' THEN SIMPTAC`

using a standard collection of list identities and calculation rules for the simpset. The lemmas corresponding to 'UUApp',

'NilApp', 'AppNil' and 'Assoc', go through as before, and together show that ("Nil","@") is a monoid. The 'ConsUnit' lemma has the same form as before and is proven using simplification.

An extra lemma, stating the strictness of "Unit", is proven using simplification:-

'UnitUU']- "Unit(1) = 1"

This minor lemma is needed later on, since the definition of "Unit" cannot appear in the same simpset as the 'ConsUnit' lemma.

3.3.3 The Theory LFREEL.

We now move from LFUN1 to the theory LFREEL, where the freeness functional "FreeMonoid" is introduced. The form of this definition remains as before. Some notation is re-introduced analogously from the first case study. Put " $Z:\beta$ " and " $P:\beta^2 \rightarrow \beta^n$ ", and put FMtm = "FreeMonoid(Z,P) f" where " $f:\alpha \rightarrow \beta$ ". Using the ML function IsMonoid the appropriate monoid assumptions for ("Z","P") can be generated. Finally, the extra assumption that the valuation function " $f:\alpha \rightarrow \beta$ " is strict is introduced (i.e " $f(1) = 1$ ").

The first two lemmas, for the "1" and "Nil" cases, go through as usual via simplification. However, the "Cons" case requires extra strictness assumptions to be introduced. These assumptions are the strictness of "f" and the left-strictness of "P". The lemma is stated below:-

'FreeMonCons' ..]- "Va l. !FMtm(Cons a l) = P(f(a),!FMtm(l))"

The proof goes through by a definedness analysis on "a", followed by simplification. This is tactically expressed as:-

GENDEFCASESTAC THEN SIMPTAC

The simpset consists of standard identities concerning lists, the two assumptions mentioned above, and the two equations obtained by instantiating the definition "FreeMonoid" by the terms "1" and "Cons a l". The same tactic as the above gives the 'FreeMonUnit' lemma, which is stated below:-

'FreeMonUnit' ..]- "Va:α l. !FMtm(Unit a) = f(a)"

Here again extra assumptions are required in addition to those needed from the first case study. There, it was assumed that "Z" is a right-identity for "P". This is also needed here, but, as well, the function "f" is assumed to be strict. The simpset used here is the same as for 'FreeMonCons' except that the "right-identity" assumption replaces the "left-strictness" assumption used there; the definition of "Unit" is also necessary.

The need for these extra assumptions is due to the conditional form of the basic identities like 'NullCons' and 'TailCons'. The antecedents of these rules each require the definedness of certain quantities. When these rules are applied during conditional simplification, definedness assumptions concerning the values of free variables are eventually needed. It is these more primitive assumptions that are introduced by the definedness analyses. For example, to reduce the term "Null(Cons a Nil)", it is necessary to know that "a" is defined (i.e. " $\partial(a) \equiv \text{TT}$ ") in order to apply the 'NullCons' rule. The strictness of "f" is used to show the truth of the subgoal " $\text{!FMtm}(\text{Unit } 1) \equiv f(1)$ ".

Note that, in proving the 'FreeMonUnit' lemma, the 'FreeMonCons' rule was not used directly, even though it had been proven earlier. To use this (within simplification), the "left strictness" assumption would have also been necessary. By using the above simpset instead, this extra assumption was avoided.

The next lemma corresponds to 'FreeMonAppend' from the original study. This is:-

```
'FreeMonAppend'
....]- "Vl1 l2. !FMtm(l1 @ l2) ≡ P(!FMtm(l1), !FMtm(l2))"
```

The tactic used to prove this is:-

```
List1TAC' THEN SIMPTAC
```

Note that the embedded definedness analysis introduced by List1TAC' leads to the strictness of "f" assumption being used. The simpset contains this assumption as well as all the corresponding lemmas from the original simpset used in the first study.

When the tactic proof for this lemma was originally developed*, I had not proven the present version of 'FreeMonCons'. The earlier version 'FreeMonCons' still contained the conditional term and its attendant condition "Null(Cons a 1)". So when this lemma was used it introduced further occurrences of the conditional. Now, the original tactic for 'FreeMonCons' involved an explicit conditional case analysis on the value of the term "Null(Cons a 1)" and a definedness analysis on "a"! The rather baroque form of this tactic led me to reconsider the form of the 'FreeMonCons' lemma. By noting that a "definedness analysis" could be included in the proof of 'FreeMonCons', the form of the lemma was made simpler. This, in turn, led to a simplification of the original 'FreeMonAppend' tactic.

This experience shows that the form of lemmas used as auxiliary results can have a radical effect on the structure of tactics which utilise them. In particular, by finding a more appropriate form of a lemma for use in another proof, this could lead to improvements in the behaviour of tactics which depend upon them. In this case, the more efficient lemma could lead to a fairly dramatic improvement (removal of a case analysis, for example) in every tactic which depended on it. The new lemma was, in this case, easily found. However, this might not always be the case!

Before moving to the tactical proofs of the Uniqueness Theorem, a tactic for monoid induction is introduced, as in the first case study. The proof component of the basic monoid induction tactic used previously, MONOIDTAC, depends only on the shape of certain constants and lemmas (e.g. the signature of "Unit" and "@", and the form of the 'ConsUnit' lemma), and not on how they are derived. Because of this, the proof component of MONOIDTAC generalises directly to a corresponding basic monoid induction tactic, called MONOID1TAC, applicable in this case study. As before, the general form of this tactic is not quite optimal for their application

* The second study presented was the first study to be formalised within LCP. Hence, the experience of the simpler study presented first was not to hand.

here. A second tactic, called MON1TAC, is introduced, which is analogous to the tactic MONTAC from the first study; the definition of MON1TAC in ML can be obtained by replacing MONOIDTAC by MONOID1TAC in the definition of MONTAC. Note that, because of this direct correspondence with MONTAC, the tactic MON1TAC does not insert any definedness analyses.

The ML function IsHom, which generates the monoid morphism assertion is also required for the next Theorem; its definition in ML is unchanged.

Having completed these preliminaries, we now consider the corresponding Uniqueness Theorem. As before, an arbitrary function $G: (\alpha) \text{List} \rightarrow \beta$ is assumed to be a monoid morphism from $(\text{"Nil"}, \text{"@"})$ to $(\text{"Z"}, \text{"P"})$. It is also assumed that "G" extends the valuation function "f", expressed in the form $f = \lambda a: \alpha. G(\text{Unit } a)$.

As in the first case study, two tactical proofs are given for this theorem, corresponding to those in the original study. The tactic for the first proof is:-

SIMPTAC THEN List1TAC' THEN SIMPTAC

The simpset used here contains all the corresponding assumptions as well as the corresponding lemmas that were used in this simpset from the first study. The only extra simprule is 'UnitUU' expressing the strictness of "Unit".

The above tactic is similar in form to the corresponding tactic from the first study. Most of the extra antecedents of simprules (like "left-strictness" for 'FreeMonAppend') are directly available as (assumed) hypotheses. However, the strictness of "f" is needed in order to apply the 'FreeMonUnit' and 'FreeMonApp' lemmas. This can be proven from the assumption that "G" extends "f" using simplification; The initial SIMPTAC application substitutes $\lambda a. G(\text{Unit } a)$ for "f" throughout the goal and then strictness can be derived in situ, since both "G" and "Unit" are known to be strict.

Note that potential difficulties concerning matching of function-typed terms do not arise here. This is because in each conditional simprule in which the strictness of "f" is a

antecedent, the symbol "f" is not applied within the lhs of the consequent. That is, each rule is of the form:-

$$\forall Z P f. (f(1) = 1) \ \& \ \dots \supset \ (\dots(f)\dots) = (\dots)"$$

In other words, the symbol "f" only appears in a rand, or argument, position. Furthermore, within sub-terms of the goal being matched, the term " $\lambda a. G(\text{Unit } a)$ " appears at each occurrence of "f" in the matching term and this is because no β -conversion can be applied to it to disrupt it's form.

The second (monoid-oriented) proof of the Uniqueness theorem, is expressed tactically by:-

MONITAC' THEN SIMPTAC

Here the simpset also corresponds to that used originally, with the single addition of the lemma 'UnitUU'.

3.4 The Third Case Study.

In the final case study in this chapter, the problem is further restricted to the study of freeness in bi-strict monoids, with respect to CPO_1 (i.e. with strict valuation functions and "Unit" function). A bi-strict monoid (" Z ", " P ") on a domain D , is a (left-strict) monoid on D that is also right-strict; that is, the following additional equation holds:-

$$\vdash \forall x:D. P(x, 1) = 1$$

As before, the problem is to show that a specific construction yields a free algebra in the category of all bi-strict monoids, with respect to the category of all strict functions.

3.4.1 The Theory L2.

The new list operator, List2, is defined (within the LCF theory L2) as the least solution of the domain equation:-

$$(\alpha)\text{List2} \approx \text{dot}_1 + (\alpha \circ ((\alpha)\text{List2}))$$

This produces a bi-strict "Cons" function, where $(\text{Cons } 1 \ 1) = 1 = (\text{Cons } a \ 1)$. The main consequences of this are that 'NullCons' requires both arguments of "Cons" to be defined; that is:-

```

'NullCons'
]- "Va l. (∂(a) = TT) & (∂(l) = TT) ⊃
      Null(Cons a l) = FF"

```

Similarly, the 'HeadCons' lemma now requires the definedness of the list argument, viz:

```

'HeadCons' ]- "Va l. (∂(l) = TT) ⊃ Head(Cons a l) = a"

```

These additional conditions increase the opportunities for definedness analyses later on as further assumptions need to be available "in scope" for the lemmas to be applied as conditional simprules.

3.4.2 The Theory LFUN2.

Turning now to the theory LFUN2 where "Unit" and "Append" are defined, all the lemmas from the second case study go through at the expense of an extra definedness analysis for list variables occurring within a "Cons" term. For example, consider the 'ConsApp' lemma:-

```

'ConsApp' ]- "Va l1 l2. ∂(a) = TT & ∂(l1) = TT ⊃
      (Cons a l1) @ l2 = Cons a (l1 @ l2)"

```

This now requires definedness analyses on both "a" and "l₁", to provide the definedness conditions to enable the 'HeadCons' and 'TailCons' lemmas to be engaged. In addition, the strictness properties of 'Cons', 'Head', 'Tail' and 'Null' are necessary for those cases in which one of "a" or "l" is undefined. This proof is tactically expressed as:-

```

REPEAT (GENDEFCASETAC THEN SIMPTAC)

```

Definedness analysis on the leading quantified variable is done, followed immediately by simplification (on the two cases). The "undefined" case is usually easily eliminated here. This is repeated until no subgoals remain or all of the quantifiers have been exhausted.

The list induction tactic, List2TAC', (most of which is automatically derivable) is similar to that given for each of the

previous case studies. The sole difference is that GENDEF CASESTAC replaces each use of GENTAC applied to the induction step subgoal. This means that a definedness case analysis is performed on both quantified variables introduced for the inductive case. So, assuming that List2INDTAC is the basic, automatically generated, structural induction tactic for the domain List2, the tactic List2TAC' is given by:-

```

let List2TAC' =
  List2INDTAC THENL
    [IDTAC; IDTAC; (GENDEF CASESTAC THEN IMPTAC'
                     THEN GENDEF CASESTAC)]
;;

```

The two lemmas 'AppNil', 'Assoc' go through straight-forwardly using the tactic:-

```
List2TAC THEN SIMPTAC
```

There are also two extra lemmas to prove here that are needed later. Neither of these are valid for the previous case studies. They are the right-strictness and the definedness of "Append":-

```

'AppUU' ]- "Vl. l @ l = l"
'DEFApp' ]- "Vl1 l2.  $\partial(l_1 @ l_2) \equiv \partial(l_1) \Rightarrow \partial(l_2) \mid l$ "

```

Both of these lemmas can also be proven by the previous tactic, although an easy "forward deduction" proof of AppUU is obtained by instantiating the definition of Append appropriately and then simplifying it. For that proof, the basic PPLAMBDA lemma:-

```
'condUU' ]- "Vt:tr. (t  $\Rightarrow$  l | l) = l"
```

is used as a simprule; the proof of this lemma uses truthvalue case analysis on "t", and is located in the LCF theory KERNEL. This completes the discussion of the theory LFUN2.

3.4.3 The Theory LFREE2.

The theory LFREE2 contains the definition of the freeness functional "FreeMonoid", defined as in each of the previous studies. The computation rules are then proven; the "I" and "Nil" cases by simplification and the "Cons" case by definedness analysis

on both quantified variables, interleaved with simplification. The corresponding tactic for this is the same as for the 'ConsApp' lemma in theory LFUN2. However, it is also necessary to assume that "f" is strict and that the binary operation "P" is bi-strict; Accordingly, the definition of the ML function IsMonoid is changed to include the extra "right-strictness" assumption for the binary operation.

The 'FreeMonUnit' lemma, is proven by the same tactic as for 'ConsApp' above. As in the second study, it is necessary to assume that "f" is strict and that "Z" is a right-identity for "P".

The next lemma to be proven is a slightly stronger version of the 'FreeMonAppend' lemma that uses one less hypothesis - namely, that "f" is strict. However, a slightly more detailed and involved tactic is required also. The stronger proof described below makes direct use of basic list identities and appropriate defining equations for "Append" and "FreeMonoid". The 'FreeMonCons' lemma is not used in simplification directly; instead, the definition of "FreeMonoid" is instantiated with a suitable "Cons" term. Other necessary simprules includes the 'DEFApp' lemma and the assumption that ("Z", "P") forms a (bi-strict) monoid. As in both previous case studies the assumption that "Z" is a right identity for "P" is not used. The goal to be proven is stated below:-

$$\forall l_1 l_2. \text{!FMtm}(l_1 @ l_2) = P(\text{!FMtm}(l_1), \text{!FMtm}(l_2))$$

The basic plan is to first do a structural induction on "l₁", followed by a definedness analysis on "l₂" with a final round of simplification to finish off. This is easily expressed tactically as:-

```
List2TAC' THEN SIMPTAC
      THEN GENDEFCASESTAC
      THEN SIMPTAC
```

As usual, the "Cons" case in the proof is the least trivial part. The new variable introduced by induction is "l₁'". Each application of "FreeMonoid" to a "Cons" term is expanded. This contains various terms such as "Null(Cons a (l₁' @ l₂))" and so on. In order to show that the above term is "FF", for example, it is

necessary to know that " $\partial(l_1' @ l_2) = TT$ ". This, in turn, uses 'DEFApp' to express the lhs in terms of " $\partial(l_1')$ " and " $\partial(l_2)$ ". The term " $\partial(l_1')$ " is determined by the definedness case analysis embedded in List2TAC', and the term " $\partial(l_2)$ " is dealt with by the explicit use of GENDEPCASESTAC. The strictness of "f" is not required, because, in the case where the term " $f(l)$ " appears, "a" is assumed to be "1" and so terms like " $Null(Cons\ a\ (l_1' @ l_2))$ " also reduce to "1". Since this term occurs within the condition part of a conditional, this reduces to "1" directly. By appeals to the strictness of "FreeMonoid" and the assumed left-strictness of "P", the equality of the given terms is established.

A variant of this proof goes through for the second case study also, showing that the 'FreeMonAppend' lemma is generally independent of whether f is strict or not. The proof of this stronger form essentially avoids using the 'FreeMonCons' lemma as a simprule. Instead, the more primitive expansion is used from which strictness can be derived using the properties of conditional without analysing the value of " $f(l)$ ".

As previously, a monoid induction tactic, MON2TAC' is available acting in exactly the same way as the monoid tactics used earlier. The final theorem, 'FreeMonUnique', is given two proofs as before. These use the corresponding simpsets from the second case study (including the right strictness assumption on "P") as well as the corresponding tactics, obtained by replacing List1TAC' by List2TAC' to get the tactic for the first proof. The second proof is given by a similar replacement of monoid induction tactics. The only major difference from the second case study is that the simpset for the first proof also needs the following easily proven lemmas*, in addition to the lemma's 'UUApp' and 'AppUU':-

```
] - "$@(1 : (α)List2 # (α)List2) = 1 : (α)List2"
..]- "P(1 : β # β) = 1 : β"
```

The need for these arises because of multiple case analyses

* PPLAMBDA terms involving infix operators, like "@", may also be written in prefix notation by prefixing the operator name by a single \$ symbol. For example, the term " $l_1 @ l_2$ " is identical to the term " $\$(l_1, l_2)$ ". See [LCF], p83.

introducing pairs of "1" elements, which when simplified are coalesced into a single "1" element, of a Cartesian product type. Unfortunately, the simplifier cannot match these reduced terms against equations expressing the strictness of operators for specific argument positions.

3.5 Summary.

In the above, we considered how to use PPLAMBDA to formally express the freeness criterion for monoids in several different settings. The formulation of this criterion made essential use of PPLAMBDA's capability for expressing propositions about higher type, polymorphic quantities.

Most of the formal proofs were performed in a tactical fashion, and closely followed the informal proofs sketched earlier on. The tactics were mainly of the form "Do induction, then simplify". The form of these proofs were generally preserved between case studies, except when extra definedness analyses were needed, due to the introduction of strict Cons operations in the latter case studies. Also the preservation of structure extended also to the various assumptions required during the proofs.

The general similarity of form can be explained by noting that the principal functions of interest had identical definitions in each case study (i.e. "@", "Unit", "FreeMonoid"). Note that even when extra definedness analyses were needed, they could all be systematically performed by judicious applications of GENDEF CASESTAC to eliminate a universal quantifier and introduce the required definedness case analysis.

Chapter 4

The Multiset case study.

In this chapter, an LCF case study is presented in which the correctness of a simulation of multisets (or "bags") in terms of lists is formalised using PPLAMBDA and then proven correct. In Section 4.1 below, the freeness criterion for multisets is formalised within PPLAMBDA, using Burstall and Landin's technique from Chapter 3. In the following section, a simulation of multisets by an effective quotient of a domain of finite sequences is informally introduced and motivated.

The next section starts the more formal development of the case study. The simple theory structure used is presented, along with the various function constants introduced in each theory. Section 4.4 then contains a statement of the correctness criteria formally proven. This is followed by a section stating the additional notation, conventions, assumptions and tactics that will be used in presenting the case study.

The style of presenting proofs in this case study is different from that used in Chapter 3. The format will be to first present the "informal style" proof, followed by a tactic for generating the formal proof. Such an arrangement facilitates the comparison of the forms of argument used in each proof.

The case study itself is given in Section 4.6, with a number of basic results concerning the operations and equivalence predicate. The next three sub-sections then present major phases of the development of the overall proof. The first stage shows the symmetry of the multiset equivalence predicate, the second then develops various congruence properties, finishing off with a proof of transitivity. The third and final stage then shows that the Append operation is commutative with respect to the multiset equivalence predicate.

In Section 4.7, Avra Cohn's work in applying Resolution oriented methods within LCF's tactical proof methodology is extended and illustrated by reexamining the tactical proofs of two

previous theorems from this study using these techniques.

Finally, in Section 4.8, an informal set-theoretic proof is given of the freeness property, building upon the formally verified results given earlier.

4.1 Commutative Monoids and Multisets.

The multiset algebra over a given domain of generators, A , is defined to be the freely-generated commutative (continuous and bi-strict) monoid with respect to some inclusion of the domain of generators, A .

So, recalling Section 2.5, a commutative monoid consists of a Scott domain M with a constant $\square: M$ and a continuous binary function $\oplus: M^2 \rightarrow M$ such that the following equational properties hold:

'Left-strict' $\vdash \forall m:M. \square \oplus m \equiv \square$
 'Right-strict' $\vdash \forall m:M. m \oplus \square \equiv \square$
 'Left-identity' $\vdash \forall m:M. \square \oplus m \equiv m$
 'Right-identity' $\vdash \forall m:M. m \oplus \square \equiv m$
 'Associativity' $\vdash \forall m_1 m_2 m_3:M. (m_1 \oplus m_2) \oplus m_3 \equiv m_1 \oplus (m_2 \oplus m_3)$
 'Commutativity' $\vdash \forall m_1 m_2:M. m_1 \oplus m_2 \equiv m_2 \oplus m_1$

The data constraint describing the freeness criterion can be given using Burstall and Landin's technique, as used in the case studies in Chapter 3. Specifically, a second-order functional, `FreeBag`, is introduced with the type:-

`FreeBag: (B # (B2 → B)) → (A → B) → (M → B)`

As before, the first curried parameter represents a given commutative monoid (with carrier domain given by B). The second curried parameter gives a (strict) valuation of the generators contained in A , within the carrier B . The result of applying these two parameters is then a strict commutative monoid morphism from the carrier of the multiset algebra, M , to the carrier of the given commutative monoid algebra, B . The freeness criterion is simply that the resulting commutative monoid morphism is the unique morphism extending the given valuation on generators, with respect to a (specified) injection of generators $U: A \rightarrow M$.

All this may be stated axiomatically as follows. Let

$(Z:B, P:B^2 \rightarrow B)$ be any commutative monoid algebra on B and let $f: A \rightarrow B$ be any (strict) valuation of the generator domain, A , within the carrier domain B . The freeness criterion can now be expressed as follows (with the aid of some meta-abbreviations for formulae):-

```

'FB1' ⊢ (FreeBag (Z,P) f) is_strict

'FB2' ⊢ ( (Z,P) is_commutative_monoid
        & f is_strict
        )
      ⊃ ∀a:A. (FreeBag (Z,P) f) (U(a)) = f(a)

'FB3' ⊢ ( (Z,P) is_commutative_monoid
        & f is_strict
        )
      ⊃ (FreeBag (Z,P) f) is_CM_morphism_from
                                                (□, ⊗) to (Z,P)

'FB4' ⊢ ( (Z,P) is_a_commutative_monoid
        & f is_strict
        & G is_CM_morphism_from (□, ⊗) to (Z,P)
        & ∀a:A. G(U(a)) = f(a)
        )
      ⊃ ∀m:M. (FreeBag (Z,P) f) (m) = G(m)

```

The first property FB1 states that FreeBag is suitably strict. The second states that FreeBag $(Z,P)(f)$ extends the valuation function f (assuming that (Z,P) is a commutative monoid algebra and that f is strict). The third property states that $(\text{FreeBag } (Z,P) f)$ is a commutative monoid morphism from the commutative monoid $(\square, \otimes)$ to (Z,P) , given the same assumptions as for the second axiom. Finally, if G is any strict commutative monoid morphism from $(\square, \otimes)$ to (Z,P) which extends the valuation function $f:A \rightarrow B$ then $(\text{FreeBag } (Z,P) f) = G$. In other words, $(\text{FreeBag } (Z,P) f)$ is the unique strict commutative monoid morphism extending the valuation function f .

The properties FB2 - 4 are non-trivial implicative formulae whose antecedents state "pre-conditions" upon the arguments to the second-order polymorphic functional, FreeBag. Note that these properties do not specify a unique function when the antecedents fail to hold (i.e. the value of the function is not constrained). However, its behaviour is entirely determined whenever the

antecedents hold (by property FB4).

The formula abbreviations used above are defined by:-

" $(Z:\beta, P:\beta^2 \rightarrow \beta)$ is_commutative_monoid" means:-

" $(\forall x:\beta. P(1,x) \equiv 1) \quad \& \quad (\forall x:\beta. P(x,1) \equiv 1) \quad \&$
 $(\forall x:\beta. P(Z,x) \equiv x) \quad \& \quad (\forall x:\beta. P(x,Z) \equiv x) \quad \&$
 $(\forall x \ y \ z:\beta. P(P(x,y),z) \equiv P(x,P(y,z))) \quad \&$
 $(\forall x \ y:\beta. P(x,y) \equiv P(y,x))"$

" $(f:\alpha \rightarrow \beta)$ is_strict" means:- " $f(1_\alpha) \equiv 1_\beta$ "

" F is_CM_morphism_from $(Z_1:\alpha, P_1:\alpha^2 \rightarrow \alpha)$ to $(Z_2:\beta, P_2:\beta^2 \rightarrow \beta)$ "
means:-

" $F(1_\beta) \equiv 1_\beta \quad \&$
 $F(Z_1) \equiv Z_2 \quad \&$
 $(\forall x \ y:\alpha. F(P_1(x,y)) \equiv P_2(F(x),F(y)))"$

4.2 Simulating Multisets.

This case study was originally motivated by [Loeckx80a] and [Loeckx80b] in which an "algorithmic specification" of sets was formalised and proven correct within the AFFIRM theorem proving system (see [Musser80], [Gerhart et al 80]). The formalisation was first given in a notation borrowed from Stanford LCF, and then translated into corresponding AFFIRM statements. The equivalence function was defined by using the anti-symmetric property of subset. It was then formally shown that the "set equivalence" function defined did represent an equivalence relation and that suitable congruence properties held for the operators. Finally, the implemented operations were informally shown to simulate the desired operations, by means of an "abstraction", or "retrieve", function (see [Milner71], [Hoare72] or [Jones80]).

However, the specification of sets used was not entirely satisfactory in itself, since the definitions made use of the ability to select elements from an apparently unordered set. This can only be achieved if an underlying representation is taken into account.

The previous discussion has described what a commutative monoid algebra consists of and gives the additional freeness requirements. We now describe a simple "implementation" of a particular commutative monoid algebra freely generated over some flat domain

of atoms, A.

The idea is to simulate the algebra by constructing the flat domain consisting of all finite sequences of atomic elements from A. This domain is then given an effective congruence predicate which equates together all list values representing the same "abstract" multiset value.

This predicate is defined by recursion over lists, and makes use of auxiliary functions for removing the first occurrence of a given element from a list and for testing membership of a list. Both of these functions are also defined recursively over lists, and need to test for equality of atomic values from A. This leads directly to the pre-requisite that the domain A possesses a (continuous) equivalence predicate.

This particular simulation of multisets by lists permits the multiset union operation to be represented by a simple concatenation operation on lists. The empty multiset is represented by the empty list.

The correctness of this simulation is to show that the recursively defined multiset equivalence predicate does indeed represent an equivalence relation and that the operations satisfy the properties of being a commutative monoid algebra with respect to this equivalence predicate. It is also necessary to show that the equivalence predicate is a congruence for the operations. These propositions have been formalised in PPLAMBDA and then formally proven within LCF.

Satisfying these requirements certainly ensures that the algebra represents some commutative monoid. It does not, however, show that this is freely generated over the chosen generators. It is conceivable that the given implementation satisfies more relations or equations than those required. The above has not ruled out the possibility that there could be lists which are equated together but which ought to represent distinct "abstract" multisets.. Hence, it must also be shown that the constructed congruence predicate exactly characterises the equality relation for the commutative monoid algebra freely generated by A. Also it must be shown that each "abstract" multiset value possesses a

representation in terms of some list value (in general, there will be many such representatives). The proofs of such propositions were not carried out formally using LCP, but are included as a self contained part of the informal proof of correctness (see Section 4.8 below).

4.3 Theory structure and basic definitions.

The theory structure for the case study is presented here, as well as definitions of the function constants introduced. The theory inheritance diagram is given in Figure 4.1 below.

The theory BASIC, introduced in Section 2.1.2, forms the base for this short tower of theories, and includes the basic theories PL, KERNEL and SMASH amongst its ancestors.

The theory EQFUN was developed in Section 2.4.2 and contains general properties of equivalence predicates, which can be used by giving suitable specialisations. The next theory, ATOM, introduces a single polymorphic, binary predicate:-

- EqAt - : $\alpha \rightarrow \alpha \rightarrow tv$

No axioms are given for this function. However, it is permitted for theorems to explicitly mention, as a collection of hypotheses that EqAt represents a continuous equivalence predicate. The polymorphism indicates that no commitment, or bias, towards a



FIGURE 4.1

particular domain of atoms has been made.

The theory LA (for "lists of atoms") axiomatises the polymorphic flat domain of finite lists, $(\alpha)L$, using the domain equation:-

$$(\alpha)L \cong \underline{\text{dot}}_1 + (\alpha \otimes (\alpha)L)$$

The usual list manipulating primitives are defined here (e.g. Head, Tail, Null, etc). The theory is identical to that given in Section 3.4, the third case study of Chapter 3.

The theory LAFUN contains all the definitions of the main functions of interest for the case study. These are:-

- EqBA - $:(\alpha)L \# (\alpha)L \rightarrow \text{tr}$
 - Minus - $:(\alpha)L \# \alpha \rightarrow (\alpha)L$
 - IsIn - $:\alpha \# (\alpha)L \rightarrow \text{tr}$

In addition, the following standard functions are used:-

- @ - $:(\alpha)L \# (\alpha)L \rightarrow (\alpha)L$
 Unit $:\alpha \rightarrow (\alpha)L$

The definitions of the concatenation and unit functions are as for the case studies given in Chapter 3.

The predicate EqBA is intended to represent the equality predicate on multisets over some domain of generators, A. It is defined by the axiom:-

'EqBA']- $\forall l_1 l_2 : (A)L.$
 $(l_1 \text{ EqBA } l_2) = (\text{Null } l_1) \Rightarrow (\text{Null } l_2) \mid$
 $((\text{Head } l_1) \text{ IsIn } l_2) \text{ and}$
 $((\text{Tail } l_1) \text{ EqBA } (l_1 \text{ Minus } (\text{Head } l_1)))$

The underlying algorithm for multiset equivalence is explained in the following way:- if the first list, l_1 , is empty, then so also must the second list l_2 be. Otherwise, check if the first element of l_1 belongs to the list l_2 . If so, then check that the remainder of the first list, l_1 , is equivalent to l_2 with one occurrence of the first element of l_1 removed. On the other hand, if the first element of l_1 does not belong to l_2 then the two lists represent distinct "abstract" multiset values.

The membership function, IsIn, is defined as follows:-

'IsIn']- $\forall a: \alpha. \forall l: (\alpha)L.$
 $(a \text{ IsIn } l) = (\exists(a) \text{ and } (\text{Null } l)) \Rightarrow \text{FF} \mid$
 $(a \text{ EqAt } (\text{Head } l)) \text{ or } (a \text{ IsIn } (\text{Tail } l))$

This definition represents the obvious algorithm in which the given element, "a" is compared with each element of the given list "l", in turn. If any are found to be equal to "a" then the result is "TT"; otherwise the result is "FF".

The function "Minus" is defined as follows:-

'Minus']- $\forall a: \alpha \ l: (\alpha)L.$
 $(l \text{ Minus } a) = \exists(a) \text{ and } \text{Null } (l) \Rightarrow \text{Nil} \mid$
 $(a \text{ EqAt } \text{Head}(l)) \Rightarrow \text{Tail}(l) \mid$
 $\text{Cons } (\text{Head } l) ((\text{Tail } l) \text{ Minus } a)$

"Minus" produces the input list, "l", with the first occurrence of the given element, "a", removed. This definition describes the obvious algorithm in which the list is traversed, skipping the first occurrence of the given element, "a".

4.4 Correctness criteria.

As discussed in section 4.1, the correctness of this multiset simulation may be stated as properties of the equivalence predicate EqBA. The properties to be formally proven can be grouped under the three headings equivalence, congruence and validity. These are as follows:-

4.4.1 Equivalence correctness.

Show that EqBA represent an effective equivalence predicate. The effectiveness follows from the fact that it is lambda definable. In terms of PPLAMBDA this involves showing that the following properties hold:-

- Definedness of the predicate EqBA

$\forall l_1 \ l_2: (\alpha)L. \exists(l_1 =_{\text{BA}} l_2) = \exists(l_1) \text{ and } \exists(l_2)$

This is shown by the lemma 'DEFEqBA' (whose proof is similar to Lemma 4.5).

- Reflexivity of EqBA (see Theorem 4.6)

$\forall l: (\alpha)L. l =_{\text{BA}} l = \exists(l)$

- Symmetry of EqBA (see Theorem 4.9)

$$"\forall l_1 l_2 : (\alpha)L. (l_1 =_{BA} l_2) = (l_2 =_{BA} l_1)"$$

- Transitivity of EqBA (see Theorem 4.16)

$$"\forall l_1 l_2 l_3 : (\alpha)L. (l_1 =_{BA} l_2) = TT \ \& \ (l_2 =_{BA} l_3) = TT \ \supset \ (l_1 =_{BA} l_3) = TT"$$

4.4.2 Congruence requirements.

The required congruence properties show that certain restricted forms of substitution are permissible. The first property given below states that EqBA is a congruence for the Unit function:-

$$"\forall a a' : (\alpha)L. (a =_{At} a') = TT \ \supset \ Unit(a) =_{BA} Unit(a') = TT"$$

This is an easy corollary of Lemma 4.23, stating the congruence property for the Cons function, and the definition of Unit. The second requirement is the congruence property for the concatenation operation, and is shown by Lemma 4.20.

$$\begin{aligned} & "\forall l_1 l_1' l_2 l_2' : (\alpha)L. \\ & \quad (l_1 =_{BA} l_1' = TT) \ \& \ (l_2 =_{BA} l_2' = TT) \ \supset \\ & \quad ((l_1 @ l_2) =_{BA} (l_1' @ l_2')) = TT" \end{aligned}$$

It turns out that as a part of showing that all these requirements are met, other congruence properties are proven as intermediate results.

4.4.3 Validity requirements.

The validity properties state the extent to which the given simulation satisfies the multiset properties expressed in terms of the equivalence predicate EqBA. In general, all of the desired properties should hold assuming that all quantities involved are defined. Since all of the operations involved are total and strict, the simulation is guaranteed to satisfy the formal requirements even if undefined values arise. Hence, we have the following requirements:-

- "Bi-strictness" properties (see Theorem 4.3)

" $\forall l:(\alpha)L. l @ l \equiv l$ "

" $\forall l:(\alpha)L. l @ l \equiv l$ "

- "Identity" properties (see Theorem 4.7 (a) and (b))

$\forall l:(\alpha)L. \partial(l) \equiv \text{TT} \supset (\text{Nil} @ l) \equiv_{\text{BA}} l \equiv \text{TT}$

$\forall l:(\alpha)L. \partial(l) \equiv \text{TT} \supset (l @ \text{Nil}) \equiv_{\text{BA}} l \equiv \text{TT}$

- The "Associativity" property (see Theorem 4.7 (c))

$\forall l_1 l_2 l_3:(\alpha)L.$

$\partial(l_1) \equiv \text{TT} \ \& \ \partial(l_2) \equiv \text{TT} \ \& \ \partial(l_3) \equiv \text{TT} \supset$

$(l_1 @ l_2) @ l_3 \equiv_{\text{BA}} l_1 @ (l_2 @ l_3) \equiv \text{TT}$

- The "Commutativity" property (see Theorem 4.19)

" $\forall l_1 l_2:(\alpha)L.$

$\partial(l_1) \equiv \text{TT} \ \& \ \partial(l_2) \equiv \text{TT} \supset (l_1 @ l_2) \equiv_{\text{BA}} (l_2 @ l_1) \equiv \text{TT}$ "

Generally speaking, slightly stronger theorems are actually proven from which the above requirements can be easily derived as easy corollaries.

However, these properties by themselves do not fully show that the given "implementation" correctly simulates the desired multiset algebra. This part of the proof gives a set-theoretical construction of a particular (denotational) model of the required multisets. This construction is made using functions and predicates which are formally discussed within PPLAMBDA, and shows the sense in which the multiset algebra is "simulated" by these functions. Because of the set-theoretical nature of part of the construction, not all of this proof can be naturally formalised within PPLAMBDA. A detailed, but informal, account of this part of the proof is given in Section 4.8.

4.5 Further preliminary details and remarks.

In order to ease the description of the case study itself, extra notation, tactics and conventions are introduced and established.

4.5.1 The form of initial goals.

In general, the initial goal for a tactically proven lemma consists of the goal formula as quoted. The assumption list at least contains the formulae 'DEFEqAt', 'RefLEqAt', 'SymEqAt' and 'TransEqAt' asserting that EqAt is an equivalence predicate. The standard simpset will consist of the following simprules:-

- The basic PPLAMBDA simpset BASICSS (including properties of smash pairing and projections).
- Various standard identities from the theory KERNEL, such as 'DEFCond', 'DEFxTT' and 'condUU'.
- Basic list identities, expressing interactions between the constructors "Nil" and "Cons", the selectors "Head" and "Tail" and the discriminator "Null". Also included are the various definedness properties for these constants.
- Properties of the (conditional) propositional functions and, or and not, including the associativity of and and or.
- Basic computational properties for "@", "IsIn", "Minus" and "EqBA" including their definedness properties.
- The assumption 'DEFEqAt' and 'RefLEqAt' are also available as simprules. In addition the devised strictness properties 'UUEqAt' and 'EqAtUU' are also included.

In proving particular lemmas, alterations to this simpset are indicated, by either adding or subtracting simprules as required.

4.5.2 Informal notation.

From now on, the following abbreviations and notational variants of functions and predicates shall be used to shorten formulae and improve their readability. These notations are shown below in Figure 4.2:-

$(a_1 =_{At} a_2)$	for	$(a_1 \text{ EqAt } a_2)$
$(l_1 =_{BA} l_2)$	for	$(l_1 \text{ EqBA } l_2)$
$(l - a)$	for	$(l \text{ Minus } a)$
$(a \leftarrow l)$	for	$(a \text{ IsIn } l)$
$(a :: l)$	for	$(\text{Cons } a \ l)$
$H(l)$	for	$\text{Head}(l)$
$T(l)$	for	$\text{Tail}(l)$

FIGURE 4.2

4.5.3 Auxiliary lemmas and global assumptions.

During the course of this case study various minor lemmas and standard identities were needed. Their proofs are mostly straight forward often involving little more than simplification and some case analysis or a simple induction. Some auxiliary lemmas related to the theories LA and LAFUN are presented in Figures 4.3 and 4.4 below.

Besides the auxiliary lemmas mentioned above, the following (labelled) global assumptions are made, concerning the nature of the function EqAt:-

```
'DEFEqAt'   $\forall a_1 a_2: \alpha. \partial(a_1 =_{At} a_2) = \partial(a_1) \text{ and } \partial(a_2)$ 
'ReflEqAt'  $\forall a: \alpha. (a =_{At} a) = \partial(a)$ 
'SymEqAt'   $\forall a_1 a_2: \alpha. (a_1 =_{At} a_2) = (a_2 =_{At} a_1)$ 
'TransEqAt'  $\forall a_1 a_2 a_3: \alpha.$ 
               $(a_1 =_{At} a_2) = TT \ \& \ (a_2 =_{At} a_3) = TT \supset$ 
               $(a_1 =_{At} a_3) = TT$ 
```

Now, the lemmas described and proven in Section 2.4.2 were formally proven within the theory EQFUN. Hence, by specialising them suitably, the following (named) lemmas are obtained:-

```
'UUEqAt'  .]-  $\forall a: \alpha. (l =_{At} a) = l$ 
'EqAtUU'  .]-  $\forall a: \alpha. (a =_{At} l) = l$ 
'EqAtGenTrans'
...]-  $\forall a_1 a_2 a_3: \alpha \ t: tr.$ 
       $(a_1 =_{At} a_2) = TT \ \& \ (a_2 =_{At} a_3) = t \supset (a_1 =_{At} a_3) = t$ 
'EqAtSelfCongruence'
...]-  $\forall a_1 a_2 a_3 a_4: \alpha.$ 
       $(a_1 =_{At} a_2) = TT \ \& \ (a_3 =_{At} a_4) = TT \supset$ 
       $(a_1 =_{At} a_3) = (a_2 =_{At} a_4)$ 
```

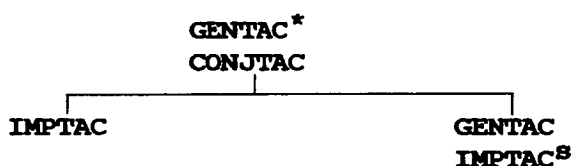
4.5.4 Tactic composition notation.

In [Cohn79], Avra Cohn introduced a precise, although informal, diagrammatic notation for compactly displaying the detailed structure of composite tactics. Such tactics are arranged in

columns, eliding the occurrences of the THEN tactical. If necessary, the THENL tactical permits specific tactics to be applied to specific subgoals, introducing branching into the form of the tactic. Each use of THENL generally introduces several new columns into the tactic diagram corresponding to each subgoal produced by the first argument to THENL. Tactics of the form (REPEAT T) are noted by writing T^* . Also, tactics of the form (T THEN SIMPTAC) are noted by writing T^S . For example, the composite tactic:-

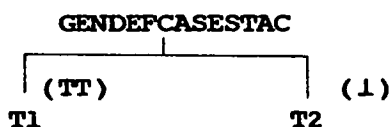
```
(REPEAT GENTAC) THEN CONJTAC THENL
  [ IMPTAC; (GENTAC THEN IMPTAC THEN SIMPTAC) ]
```

could be represented by the tactic diagram:-



The ORELSE tactical also introduces branching into the form of tactics; the tactic (T1 ORELSE T2) can be written as (T1 ? T2) where the tactics T1 and T2 may be composite tactics, involving several columns of component tactics.

Finally, names may be associated with branches as an aid to explaining the behaviour of the tactic; as a simple example:-



This indicates that tactic T1 is applied in the defined case, and that tactic T2 is applied in the undefined case.

4.5.5 Additional Tactics.

The structural induction tactics for lists used in this case study are called LINDTAC' and LINDTAC. These are similar to List2TAC' and List2TAC used in the final case study in Chapter 3 (see Section 3.5). In that case study, various definedness case analyses were performed as a routine matter, producing a number of

cases that are equivalent to the "1" case. The proofs conducted there were not particularly lengthy. However, this would be much more wasteful of effort here since proofs are longer and need to "branch" more.

Hence, the number of subgoals returned from the induction tactic are reduced to the minimum necessary. The behaviour of the tactic LINDTAC' is given below:-

	<table><tr><td>$\forall l.F[l]$</td></tr><tr><td>ss</td></tr><tr><td>fml</td></tr></table>	$\forall l.F[l]$	ss	fml
$\forall l.F[l]$				
ss				
fml				
LINDTAC'	=====			
	subgoal1 subgoal2 subgoal3			

where

subgoal1 =	<table><tr><td>F[1]</td></tr><tr><td>ss</td></tr><tr><td>fml</td></tr></table>	F[1]	ss	fml
F[1]				
ss				
fml				
subgoal2 =	<table><tr><td>F[Nil]</td></tr><tr><td>ss</td></tr><tr><td>fml</td></tr></table>	F[Nil]	ss	fml
F[Nil]				
ss				
fml				
subgoal3 =	<table><tr><td>F[Cons a 1']</td></tr><tr><td>{ .⊢F[1'], .⊢θ(1') = TT, .⊢θ(a) = TT } ∪ ss</td></tr><tr><td>["F[1']" ; "θ(1') = TT" ; "θ(a) = TT"] @ fml</td></tr></table>	F[Cons a 1']	{ .⊢F[1'], .⊢θ(1') = TT, .⊢θ(a) = TT } ∪ ss	["F[1']" ; "θ(1') = TT" ; "θ(a) = TT"] @ fml
F[Cons a 1']				
{ .⊢F[1'], .⊢θ(1') = TT, .⊢θ(a) = TT } ∪ ss				
["F[1']" ; "θ(1') = TT" ; "θ(a) = TT"] @ fml				

where the goal formula " $F[l]$ " admits induction in the variable l . Also, the variables " a " and " l' " are fresh variables of appropriate types, which do not occur freely in the assumptions or the goal formula. The definition of this tactic in ML is considered briefly in Section 5.6.

Another useful (parameterised) tactic is ABSURDTAC :thm $\rightarrow$ tactic, which caters for a simple class of "inconsistent" goals. The tactic simply includes the given theorem as an antecedent to the goal and then simplified, with failure if this does not reduce to a trivial goal. The main idea is that the included theorem should reduce to a standard contradiction when simplified using the local simpset. This gives a standard tautology recognised by the simplifier which then eliminates goal.

4.5.6 Presentation of proofs.

The lemmas presented here were first sketched informally by hand. Tactics for performing the proofs were then derived. In this process, various parts of the proof were corrected and certain useful lemmas discovered (e.g Lemma 4.14 and Lemma 4.21). Each lemma is presented by a detailed informal proof and then an LCF tactic-oriented description is given which broadly encapsulates this proof. In this way, the formal and informal proofs can be compared.

Each proof is, in general, presented in a "top-down", or goal-directed way. However, the order in which lemmas are presented is "bottom up" - that is, lemmas are proven before they are used. This has the unfortunate effect of shielding the motivating instances of a lemma until after it has been made available. A "top down" order of presentation is probably easier to understand and, at first glance, lends itself more to the tactical approach. However, such a pure "top down" strategy (at least in terms of proof discovery) also has the danger of assuming lemmas which, in fact, are either false or, at any rate, unprovable.

So, in practice, the natural development of a proof usually follows a mixed strategy with "top-down" and "bottom-up" phases. The "top down" activity consists of decomposing desired goals into subgoals which then have to be achieved in order to achieve the original goal. The "bottom up" activity consists of working from known information towards the achievement of particular goals. The roles of these approaches are dual to one another; the former determines what the intermediate subgoals should be and hence the overall shape of the development, while the latter provides some of the link between presently known or available information and any outstanding subgoals.

So, to reiterate, lemmas are, in general, proven before they are used. However, the proofs of lemmas tend to be "top down" interspaced with short stretches of forwards (usually equational) proof.

4.6.1 Multisets: Basic Results.

We begin by establishing a number of standard properties about each of the functions introduced in Section 4.3. For completeness, a selection of useful properties from the theory LA is given in Figure 4.3, and the basic results from the theory LAFUN are listed in Figure 4.4.

The majority of these results are easily proven by directly specialising the appropriate definition and simplifying the resulting theorem using a standard collection of (conditional) list identities. However, this simple approach fails if the theorem statement involves quantities with indeterminate definedness (e.g. variables). In such situations, definedness case analysis can be applied, as used in the case studies in Chapter 3.

The first two lemmas state familiar properties of the concatenation and unit functions:-

Lemma 4.1

```

'UUApp'      ]-  $\forall l. l @ l = l$ 
'AppUU'      ]-  $\forall l. l @ l = l$ 
'NilApp'     ]-  $\forall l. Nil @ l = l$ 
'ConsApp'    ]-  $\forall l_1 l_2 a. (a :: l_1) @ l_2 = a :: (l_1 @ l_2)$ 
'AppNil'     ]-  $\forall l. l @ Nil = l$ 
'AssocApp'   ]-  $\forall l_1 l_2 l_3. (l_1 @ l_2) @ l_3 = l_1 @ (l_2 @ l_3)$ 
'DEFApp'     ]-  $\forall l_1 l_2. \partial(l_1 @ l_2) = \partial(l_1) \text{ and } \partial(l_2)$ 

```

Proof

By simplification, definedness case analysis and structural induction over lists; similar proofs to these have been considered in Chapter 3.

QED

A sample of useful facts from Theory LA

```

'HUU'        ]-  $H(l) = l$ 
'TNil'       ]-  $T(Nil) = l$ 
'TCons'      ]-  $\forall a:\alpha l:(\alpha)L. \partial(a) = TT \supset T(a :: l) = l$ 
'NullCons'   ]-  $\forall l a. Null(a :: l) = \text{not}(\partial(a) \text{ and } \partial(l))$ 
'DEFNil'     ]-  $\partial(Nil) = TT$ 
'DEFCons'    ]-  $\forall a:\alpha l:(\alpha)L. \partial(a :: l) = \partial(a) \text{ and } \partial(l)$ 
'NullIMPNil' ]-  $\forall l:(\alpha)L. Null(l) = TT \supset l = Nil$ 

```

FIGURE 4.3

Some basic properties of constants in theory LAFUN.

Append

```

'UUApp'      ]-  $\forall l. l @ 1 = 1$ 
'AppUU'      ]-  $\forall l. l @ 1 = 1$ 
'NilApp'     ]-  $\forall l. Nil @ 1 = 1$ 
'ConsApp'    ]-  $\forall l_1 l_2 a. (a :: l_1) @ l_2 = a :: (l_1 @ l_2)$ 
'AppNil'     ]-  $\forall l. l @ Nil = 1$ 
'AssocApp'   ]-  $\forall l_1 l_2 l_3. (l_1 @ l_2) @ l_3 = l_1 @ (l_2 @ l_3)$ 
'DEFApp'     ]-  $\forall l_1 l_2. \partial(l_1 @ l_2) = \partial(l_1) \text{ and } \partial(l_2)$ 

```

Unit

```

'UnitUU'     ]-  $Unit(1) = 1$ 
'DEFUnit'    ]-  $\forall a. \partial(Unit(a)) = \partial(a)$ 
'ConsUnit'   ]-  $\forall a l. (a :: 1) = (Unit a) @ 1$ 

```

EqBA

```

'UUEqBa'     ]-  $\forall l. l =_{BA} 1 \equiv 1$ 
'EqBAUU'     ]-  $\forall l. l =_{BA} 1 \equiv 1$ 
'NilEqBA'    ]-  $\forall l. Nil =_{BA} 1 \equiv Null(1)$ 
'ConsEqBA'   ]-  $\forall l_1 l_2 a.$ 
                $(a :: l_1) =_{BA} l_2 \equiv$ 
                $\partial(l_1) \text{ and } (a \leftarrow l_2) \text{ and } (l_1 =_{BA} (l_2 - a))$ 
'DEFEqBA'    ]-  $\forall l_1 l_2. \partial(l_1 =_{BA} l_2) \equiv \partial(l_1) \text{ and } \partial(l_2)$ 

```

Minus

```

'UUMinus'    ]-  $\forall a. 1 - a \equiv 1$ 
'MinusUU'    ]-  $\forall l. l - 1 \equiv 1$ 
'NilMinus'   ]-  $\forall a. Nil - a \equiv \partial(a) \Rightarrow Nil \mid 1$ 
'ConsMinus'  .]-  $\forall l a_1 a_2.$ 
                $(a_1 :: l) - a_2 \equiv (a_1 =_{At} a_2) \Rightarrow 1 \mid a_1 :: (l - a_2)$ 
'DEFMinus'   .]-  $\forall l a. \partial(l - a) \equiv \partial(l) \text{ and } \partial(a)$ 

```

IsIn

```

'UUIsIn'     ]-  $\forall l. l \leftarrow 1 \equiv 1$ 
'IsInUU'     ]-  $\forall a. a \leftarrow 1 \equiv 1$ 
'IsInNil'    ]-  $\forall a. a \leftarrow Nil \equiv \partial(a) \Rightarrow FF \mid 1$ 
'IsInCons'   .]-  $\forall l a_1 a_2.$ 
                $a_1 \leftarrow (a_2 :: l) \equiv \partial(l) \text{ and } ((a_1 =_{At} a_2) \text{ or } a_1 \leftarrow 1)$ 
'DEFIsIn'    .]-  $\forall l a. \partial(a \leftarrow 1) \equiv \partial(a) \text{ and } \partial(l)$ 

```

FIGURE 4.4

Lemma 4.2

```

'UnitUU'      ]- Unit(1) = 1
'DEFUnit'     ]-  $\forall a. \partial(\text{Unit}(a)) = \partial(a)$ 
'ConsUnit'    ]-  $\forall a \ 1. (a :: 1) = (\text{Unit } a) @ 1$ 

```

Proof

By simplification in the first two cases and definedness case analyses followed by simplification for the third. The proofs rely on strictness and definedness results for Cons and Nil, as well as the definedness properties of conditionals.

QED

The first requirement to be established is given by the next theorem which demonstrates the bi-strictness of the predicate function, EqBA.

Lemma 4.3

```

'UUEqBA'      ]-  $\forall l. l =_{BA} 1 = 1$ 
'EqBAUU'      ]-  $\forall l. l =_{BA} 1 = 1$ 

```

Proof

Both of these go through by simplification of an instantiated definition. The first lemma just relies upon the strictness of the Null predicate, 'NullUU'. In showing the second lemma, both arms of the conditional defining $=_{BA}$ are shown to be undefined. As for the first lemma, the strictness of the Null predicate again shows that the first arm is undefined. Showing that the second arm is undefined requires the left strictness of and, 'UUand' and of Minus, 'UUMinus'. Finally, simplifying with the following lemma:-

```

'condUU' ]-  $\forall t:tr. (t = 1 \mid 1) = 1$ 

```

from the theory KERNEL, completes the proof.

QED

The next lemma gives a detailed account of a simple proof requiring properties of $=_{At}$ directly, and is the first proof to be described formally in this chapter.

Lemma 4.4 'IsInCons'

.]- $\forall l \ a_1 \ a_2.$

$$a_1 \leftarrow (a_2 :: l) = \partial(l) \text{ and } ((a_1 =_{At} a_2) \text{ or } a_1 \leftarrow l)$$

Proof

We shall assume that EqAt satisfies the definedness property for an equivalence predicate which states that:-

$$\forall a_1 \ a_2 : \alpha. \partial(a_1 =_{At} a_2) = \partial(a_1) \text{ and } \partial(a_2)$$

This entitles us to use the bi-strictness lemmas, 'UUEqBA' and 'EqBAUU', from the above. The above assumption is used similarly in the proof of 'ConsMinus' (but it is unnecessary for 'ConsEqBA'). Next, the definition of "IsIn" is appropriately specialised to give:-

$$\begin{aligned}]- \ a_1 \leftarrow (a_2 :: l) = \\ \partial(a_1) \text{ and } \text{Null}(a_2 :: l) \Rightarrow \text{FF} \mid \\ (a_1 =_{At} (H(a_2 :: l)) \text{ or } (a_1 \leftarrow (T(a_2 :: l)))) \end{aligned}$$

This equation cannot be further simplified directly because each of the basic list manipulating operators are strict. So this equation is used, for the purposes of this proof, as the defining property of "IsIn" for this pattern of arguments. Of course, the theorem "IsInCons" will be an improvement on this once it is established, but in order to do so, some rule such as this is first needed.

The proof of 'IsInCons' now proceeds by definedness case analysis on the quantified variables "l", "a₁", and "a₂". The goal is to show the truth of the equation $a_1 \leftarrow (a_2 :: l) = \partial(l) \text{ and } ((a_1 =_{At} a_2) \text{ or } (a_1 \leftarrow l))$ under various definedness assumptions.

So, assuming that $\partial(l) = 1$, this gives $l = 1$ and then $a_1 \leftarrow (a_2 :: l) = a_1 \leftarrow (a_2 :: 1) = a_1 \leftarrow 1 = 1$ (using the theorem 'IsInUU'). Also, we have $\partial(l) \text{ and } (...) = 1 \text{ and } (...) = 1$, so giving the equation in this case.

Now, assume that $\partial(l) = \text{TT}$ and that $\partial(a_1) = 1$, giving $a_1 = 1$. So, $a_1 \leftarrow (a_2 :: l) = 1 \leftarrow (a_2 :: l) = 1$ (using the basic theorem 'UUIsIn'). On the other hand, we also have $\partial(l) \text{ and } (a_1 =_{At} a_1) \text{ or } (a_1 \leftarrow l) = \text{TT and } ((1 =_{At} a_2) \text{ or } (1 \leftarrow l)) = (1 =_{At} a_2) \text{ or } (1 \leftarrow l) = (1 \text{ or } 1) = 1$, using the properties of "and" and "or" with the

left-strictness of EqAt.

Next, assume that $\partial(1) = \partial(a_1) = \text{TT}$ and that $\partial(a_2) = 1$. This proceeds much as in the previous case except that the right-strictness of EqAt and the theorem 'IsInUU' is used.

We now arrive at the final case, where $\partial(1) = \partial(a_1) = \partial(a_2) = \text{TT}$. In this case, we use the definition of "IsIn" directly to give:-

$$\begin{aligned} a_1 \leftarrow (a_2 :: 1) &= (\partial(a_1) \text{ and } \text{Null}(a_2 :: 1)) \Rightarrow \text{FF} \mid \\ & \quad (a_1 =_{\text{At}} (\text{H}(a_2 :: 1)) \text{ or } a_1 \leftarrow (\text{T}(a_2 :: 1))) \\ &= (\text{TT and FF}) \Rightarrow \text{FF} \mid (a_1 =_{\text{At}} a_2 \text{ or } a_1 \leftarrow 1) \\ &= (a_1 =_{\text{At}} a_2 \text{ or } a_1 \leftarrow 1) \end{aligned}$$

The above argument follows from definedness and various simple (conditional) list identities. Also, $\partial(1) \text{ and } (a_1 =_{\text{At}} a_2 \text{ or } a_1 \leftarrow 1) = \text{TT and } (a_1 =_{\text{At}} a_2 \text{ or } a_1 \leftarrow 1) = (a_1 =_{\text{At}} a_2 \text{ or } a_1 \leftarrow 1)$.

Hence, in every case, the required equation has been established.

QED

A tactic for this lemma is:-

GENDEF CASESTAC^{S*}

where the initial simpset additionally contains the bi-strictness theorems, 'UUEqBA' and 'EqBAUU', as well as the Cons case specialisation of the definition of IsIn.

The definition of the domain $(\alpha)L$ gives rise to a bi-strict list "Cons" function. Because of this, certain list identities have to be predicated upon definedness of certain terms (e.g. list selection via Head, Tail). In order to apply these conditional theorems as simprules, it will be necessary to show that certain terms are defined. In order to do this, it is helpful to characterise the definedness properties of the available functions in a form suitable for use during simplification. This usually involves expressing the definedness of the result in terms of the definedness, or other properties, of the arguments.

In this case study, the functions introduced are strict and total, meaning that the result is defined precisely when all arguments are themselves defined. The theorems 'DEF EqBA', 'DEF Minus' and 'DEF IsIn' each depend upon the definedness assumption on the predicate EqAt. The proofs of all these theorems

are all fairly similar and only the proof of 'DEFMinus' is given here.

Lemma 4.5 'DEFMinus'

.]- $\forall l \ a. \ \partial(l - a) = \partial(l) \ \underline{\text{and}} \ \partial(a)$

Proof

By list induction on "l" and then definedness analysis on "a" as necessary.

Suppose that $l = l$. Then $\partial(l - a) = \partial(l - a) = \partial(l) = l$ and also $\partial(l) \ \underline{\text{and}} \ \partial(a) = \partial(l) \ \underline{\text{and}} \ \partial(a) = (l \ \underline{\text{and}} \ \partial(a)) = l$, showing the equality.

Suppose that $l = \text{Nil}$. Then $\partial(l - a) = \partial(\text{Nil} - a) = \partial(\partial(a) \Rightarrow \text{Nil} \mid l) = \partial(a) \Rightarrow \partial(\text{Nil}) \mid \partial(l) = \partial(a) \Rightarrow \text{TT} \mid l = \partial(a)$, using the elementary lemmas from the 'KERNEL' theory:-

'DEFCond']- $\forall t:tr \ x \ y:\alpha. \ \partial(t \Rightarrow x \mid y) = t \Rightarrow \partial(x) \mid \partial(y)$

and

'DEFxTT']- $\forall t:tr \ x:\alpha. \ \partial(x) \Rightarrow \text{TT} \mid t = \partial(x)$

Both of these lemmas may be proven using a simple Boolean case analysis. We will also have occasion to use the following similar lemmas, also from the theory 'KERNEL':-

'BooleanCond']- $\forall t:tr. \ (t \Rightarrow \text{TT} \mid \text{FF}) = t$

and

'TTDEft']- $\forall t:tr. \ (t \Rightarrow \text{TT} \mid \text{TT}) = \partial(t)$

Returning now to the lemma in hand, suppose that $l = (a_1 :: l_1)$ where $\partial(a_1) = \partial(l_1) = \text{TT}$ with the assumption that:-

$\forall a_2. \ \partial(l_1 - a_2) = \partial(l_1) \ \underline{\text{and}} \ \partial(a_2)$

So, consider $\partial(l - a) = \partial((a_1 :: l_1) - a) = \partial((a_1 \text{ =At } a) \Rightarrow l_1 \mid a_1 :: (l_1 - a)) = (a_1 \text{ =At } a) \Rightarrow \partial(l_1) \mid \partial(a_1 :: (l_1 - a)) = (a_1 \text{ =At } a) \Rightarrow \text{TT} \mid (\partial(a_1) \ \underline{\text{and}} \ \partial(l_1 - a)) = (a_1 \text{ =At } a) \Rightarrow \text{TT} \mid (\partial(a_1) \ \underline{\text{and}} \ (\partial(l_1) \ \underline{\text{and}} \ \partial(a))) = (a_1 \text{ =At } a) \Rightarrow \text{TT} \mid (\text{TT} \ \underline{\text{and}} \ (\text{TT} \ \underline{\text{and}} \ \partial(a))) = (a_1 \text{ =At } a) \Rightarrow \text{TT} \mid \partial(a)$, using the definedness property of "Cons" and the induction hypothesis.

Now, also consider $\partial(l) \ \underline{\text{and}} \ \partial(a) = \partial(a_1 :: l_1) \ \underline{\text{and}} \ \partial(a) = (\partial(a_1) \ \underline{\text{and}} \ \partial(l_1)) \ \underline{\text{and}} \ \partial(a) = \partial(a)$. Hence, we now have to prove that:-

$\forall a. \ (a_1 \text{ =At } a) \Rightarrow \text{TT} \mid \partial(a) = \partial(a)$

Proceeding now by definedness case analysis on "a". Suppose that $\partial(a) = \perp$. Then $a = \perp$ and we have that $(a_1 \text{ =At } a) \Rightarrow \text{TT} \mid \partial(a) = (a_1 \text{ =At } \perp) \Rightarrow \text{TT} \mid \partial(\perp) = \perp \Rightarrow \text{TT} \mid \perp = \perp = \partial(a)$, using the right-strictness of EqAt (derived from its corresponding definedness property). Otherwise suppose that $\partial(a) = \text{TT}$. Now, $(a_1 \text{ =At } a) \Rightarrow \text{TT} \mid \partial(a) = (a_1 \text{ =At } a) \Rightarrow \text{TT} \mid \text{TT} = \partial(a_1 \text{ =At } a) = \partial(a_1)$ and $\partial(a) = \text{TT} = \partial(a)$, which also uses the definedness property of EqAt. In each case, the appropriate equation holds.

QED

A tactic for this lemma is:-

LINDTAC' THEN GENDEFCASESTAC⁸

Proofs such as these are often surprisingly long. In this case, the result might also be reached by noting that the definition is primitive recursive in form, and that it only depends upon other functions that are also total and strict.

Note that the determination of definedness for an arbitrary, finite collection of function definitions is related to the classical "halting" problem for Turing machines and hence is generally undecidable.

Theorem 4.6 (Reflexivity of "EqBA")

..]- $\forall l:(\alpha) \text{ L. } (l \text{ =BA } l) = \partial(l)$

(on the assumption that both 'DEFEqAt' and 'RefLEqAt' hold.)

Proof By list induction on l.

Suppose that $l = \perp$. Then $(l \text{ =BA } l) = (\perp \text{ =BA } \perp) = \perp \partial(\perp)$, using lemma 'UUEqBA' (which depends on 'DEFEqAt').

Suppose that $l = \text{Nil}$. Then $(l \text{ =BA } l) = (\text{Nil} \text{ =BA } \text{Nil}) = \text{Null}(\text{Nil}) = \text{TT} = \partial(\text{Nil})$.

Suppose that $l = (a :: l_1)$ where $\partial(a) = \partial(l_1) = \text{TT}$ and with the induction hypothesis that $(l_1 \text{ =BA } l_1) = \partial(l_1)$. Now, $a \leftarrow (a :: l_1) = \partial(l_1)$ and $((a \text{ =At } a) \text{ or } a \leftarrow l_1) = \text{TT}$ and $(\partial(a) \text{ or } (a \leftarrow l_1)) = \text{TT}$ or $(a \leftarrow l_1) = \text{TT}$, and also $((a :: l_1) - a) = (a \text{ =At } a) \Rightarrow l_1 \mid a :: (l_1 - a) = \partial(a) \Rightarrow l_1 \mid a :: (l_1 - a) = l_1$. Both of these derivations assume that 'RefLEqAt' holds. So, we have that $(l \text{ =BA } l) = (a :: l_1) \text{ =BA } (a :: l_1) = (\partial(l_1) \text{ and } (a \leftarrow (a :: l_1)))$ and $(l_1$

$\neg_{BA}((a :: l_1) - a) = (TT \text{ and } TT) \text{ and } (l_1 \neg_{BA} l_1) = Tt \text{ and } \partial(l_1) = TT$. Also, $\partial(1) = \partial(a :: l_1) = \partial(a) \text{ and } \partial(l_1) = TT \text{ and } TT = TT$.

QED

A tactic for proving this lemma is

LINDTAC' THEN SIMPTAC

(using the standard simpset)

Theorem 4.7

(a) $..]- \forall l:(\alpha)L. (Nil @ l) \neg_{BA} l = \partial(l)$

(b) $..]- \forall l:(\alpha)L. (l @ Nil) \neg_{BA} l = \partial(l)$

(c) $..]- \forall l_1 l_2 l_3:(\alpha)L.$

$$((l_1 @ l_2) @ l_3) \neg_{BA} (l_1 @ (l_2 @ l_3)) = \partial(l_1) \text{ and } (\partial(l_2) \text{ and } \partial(l_3))$$

(on the assumption that both 'DEFEqAt' and 'RefLEqAt' hold).

Proof

Each lemma is shown by simplification using standard list identities and the above assumptions. Only the proof for (c) is shown as the others are similar. So, $((l_1 @ l_2) @ l_3) \neg_{BA} (l_1 @ (l_2 @ l_3)) = (l_1 @ (l_2 @ l_3)) \neg_{BA} (l_1 @ (l_2 @ l_3)) = \partial(l_1 @ (l_2 @ l_3)) = \partial(l_1) \text{ and } \partial(l_2 @ l_3) = \partial(l_1) \text{ and } (\partial(l_2) \text{ and } \partial(l_3))$, using Theorem 4.6.

QED

A tactic for all of these theorems is:-

SIMPTAC

where the initial simpset is enriched by Theorem 4.6, 'Appnil' and 'AssocApp'.

4.6.2 Multisets: Symmetry of EqBA.

Lemma 4.8

$..]- \forall a:\alpha l_1 l_2:(\alpha)L.$

$$l_1 \neg_{BA}(a :: l_2) =$$

$$(\partial(l_2) \text{ and } (a \leftarrow l_1)) \text{ and } ((l_1 - a) \neg_{BA} l_2)$$

(on the assumption that both 'DEFEqAt' and 'SymEqAt' holds).

Proof

The first stage is to do a definedness case analysis on the value of "a".

Suppose that $\partial(a) \equiv 1$, and so $a \equiv 1$. Then $l_1 \text{ =BA } (a :: l_2) \equiv l_1 \text{ =BA } (1 :: l_2) \equiv l_1 \text{ =BA } 1 \equiv 1$. Now, by a definedness case analysis on the value of "l₂", suppose that $\partial(l_2) \equiv 1$. Then $(\partial(l_2) \text{ and } (a \leftarrow l_1)) \text{ and } ((l_1 - a) \text{ =BA } l_2) \equiv (1 \text{ and } (...)) \text{ and } (...) \equiv 1$. On the other hand, suppose that $\partial(l_2) \equiv \text{TT}$. Then $(\partial(l_2) \text{ and } (a \leftarrow l_1)) \text{ and } ((l_1 - a) \text{ =BA } l_2) \equiv (\text{TT and } (1 \leftarrow l_1)) \text{ and } (...) \equiv 1 \text{ and } (...) \equiv 1$. In either case, the desired equation holds.

Now, suppose that $\partial(a) \equiv \text{TT}$ and proceed with a list induction on "l₁".

Suppose that $l_1 \equiv 1$. So, then $l_1 \text{ =BA } (a :: l_2) \equiv 1 \text{ =BA } (a :: l_2) \equiv 1$. By doing a definedness case analysis on the value of "l₂", similar to the one given above, we have that $(\partial(l_2) \text{ and } (a \leftarrow l_1)) \text{ and } ((l_1 - a) \text{ =BA } l_2) \equiv 1$, which gives the desired equation.

Suppose that $l_1 \equiv \text{Nil}$. Then $l_1 \text{ =BA } (a :: l_2) \equiv \text{Nil =BA } (a :: l_2) \equiv \text{Null}(a :: l_2) \equiv \text{not } (\partial(a) \text{ and } \partial(l_2)) \equiv \text{not}(\text{TT and } \partial(l_2)) \equiv \text{not}(\partial(l_2))$. Now, do a definedness case analysis on the value of "l₂", and suppose that $\partial(l_2) \equiv 1$ and thence $l_2 \equiv 1$. So, $\text{not } (\partial(l_2)) \equiv \text{not } (1) \equiv 1$ and also, $(\partial(l_2) \text{ and } (a \leftarrow l_1)) \text{ and } ((l_1 - a) \text{ =BA } l_2) \equiv (1 \text{ and } (...)) \text{ and } (...) \equiv 1$. Alternatively, suppose that $\partial(l_2) \equiv \text{TT}$, and then $\text{not } (\partial(l_2)) \equiv \text{not}(\text{TT}) \equiv \text{FF}$. Also $(\partial(l_2) \text{ and } (a \leftarrow l_1)) \text{ and } ((l_1 - a) \text{ =BA } l_2) \equiv (\text{TT and } (a \leftarrow \text{Nil})) \text{ and } (...) \equiv (\partial(a) \neq \text{FF} \mid 1) \text{ and } (...) \equiv \text{FF}$. In either case, the desired equation holds.

Suppose that $l_1 \equiv (a_1 :: l_3)$ where $\partial(a_1) = \partial(l_3) = \text{TT}$ with the induction hypothesis:-

$$\forall l_4. l_3 \text{ =BA } (a :: l_4) \equiv$$

$$(\partial(l_4) \text{ and } (a \leftarrow l_3)) \text{ and } ((l_3 - a) \text{ =BA } l_4)$$

(It is important for this proof that the quantifier is included in the induction hypothesis).

The proof continues with a truth value cases analysis on the value of "a =_{At} a₁".

Suppose that $a \text{ =At } a_1 \equiv 1$. Then, using the 'DEFEqAt' property,

we have that $1 = \partial(a \text{ =At } a_1) = \partial(a)$ and $\partial(a_1) = \text{TT}$ and $\text{TT} = \text{TT}$ which is a contradiction.

Suppose that $(a \text{ =At } a_1) = \text{TT}$. So, from the 'SymEqAt' property, we get that $(a_1 \text{ =At } a) = \text{TT}$. Now, $a_1 \leftarrow (a :: l_2) = \partial(l_2)$ and $((a_1 \text{ =At } a) \text{ or } (a_1 \leftarrow l_2)) = \partial(l_2)$ and $(\text{TT or } (a_1 \leftarrow l_2)) = \partial(l_2)$ and TT . Also, $((a :: l_2) - a_1) = (a \text{ =At } a_1 \Rightarrow l_2 \mid a :: (l_2 - a_1) = l_2)$. So, $l_1 \text{ =BA}(a :: l_2) = (a_1 :: l_3) \text{ =BA}(a :: l_2) = (\partial(l_3) \text{ and } (a_1 \leftarrow (a :: l_2)))$ and $(l_3 \text{ =BA}((a :: l_2) - a_1)) = (\text{TT and } (\partial(l_2) \text{ and } \text{TT}))$ and $(l_3 \text{ =BA } l_2) = (\partial(l_2) \text{ and } \text{TT})$ and $(l_3 \text{ =BA } l_2) = \partial(l_2)$ and $(l_3 \text{ =BA } l_2)$.

On the other hand, consider the value of $(a \leftarrow l_1) = a \leftarrow (a_1 :: l_3) = \partial(l_3)$ and $((a \text{ =At } a_1) \text{ or } (a \leftarrow l_3)) = \text{TT}$ and $(\text{TT or } (a \leftarrow l_3)) = \text{TT}$. Also, $(l_1 - a) = ((a_1 :: l_3) - a) = (a_1 \text{ =At } a) \Rightarrow l_3 \mid a_1 :: (l_3 - a) = l_3$. So, $(\partial(l_2) \text{ and } (a \leftarrow l_1))$ and $((l_1 - a) \text{ =BA } l_2) = (\partial(l_2) \text{ and } \text{TT})$ and $(l_3 \text{ =BA } l_2) = \partial(l_2)$ and $(l_3 \text{ =BA } l_2)$. This completes this case.

Suppose that $(a \text{ =At } a_1) = \text{FF}$. As in the previous case, the 'SymEqAt' property shows that $(a_1 \text{ =At } a) = \text{FF}$. Now, doing a definedness case analysis on the value of " l_2 ", suppose that $\partial(l_2) = 1$, and so $l_2 = 1$. Then, as usual, $l_1 \text{ =BA}(a :: l_2) = l_1 \text{ =BA}(a :: 1) = (l_1 \text{ =BA } 1) = 1$. Also, $(\partial(l_2) \text{ and } (a \leftarrow l_1))$ and $((l_1 - a) \text{ =BA } l_2) = (1 \text{ and } (...))$ and $(...) = 1$, which shows identity.

Suppose now that $\partial(l_2) = \text{TT}$, and consider the following calculations. Now, $a_1 \leftarrow (a :: l_2) = \partial(l_2)$ and $((a_1 \text{ =At } a) \text{ or } (a_1 \leftarrow l_2)) = \text{TT}$ and $(\text{FF or } (a_1 \leftarrow l_2)) = a_1 \leftarrow l_2$. Also, $((a :: l_2) - a_1) = (a \text{ =At } a_1) \Rightarrow l_2 \mid a :: (l_2 - a_1) = a :: (l_2 - a_1)$. So, $l_1 \text{ =BA}(a :: l_2) = (a_1 :: l_3) \text{ =BA}(a :: l_2) = (\partial(l_3) \text{ and } (a_1 \leftarrow (a :: l_2)))$ and $(l_3 \text{ =BA}((a :: l_2) - a_1)) = (\text{TT and } (a_1 \leftarrow l_2))$ and $(l_3 \text{ =BA}(a :: (l_2 - a_1))) = (a_1 \leftarrow l_2)$ and $(l_3 \text{ =BA}(a :: (l_2 - a_1)))$. Now, apply the induction hypothesis by putting " $(l_2 - a_1)$ " for " l_4 " to get the equation $l_3 \text{ =BA}(a :: (l_2 - a_1)) = (\partial(l_2 - a_1) \text{ and } (a \leftarrow l_3))$ and $((l_3 - a) \text{ =BA } (l_2 - a_1)) = ((\partial(a_1) \text{ and } \partial(l_2)) \text{ and } (a \leftarrow l_3))$ and $((l_3 - a) \text{ =BA } (l_2 - a_1)) = (a \leftarrow l_3)$ and $((l_3 - a) \text{ =BA } (l_2 - a_1))$. So substituting we get that $l_1 \text{ =BA}(a :: l_2) = (a_1 \leftarrow l_2)$ and $(a \leftarrow l_3)$ and $((l_3 - a) \text{ =BA } (l_2 - a_1))$.

On the other hand, consider $(a \leftarrow l_1) = a \leftarrow (a_1 :: l_3) = \partial(l_3)$

and $((a \text{ =At } a_1) \text{ or } (a \leftarrow l_3)) = \text{TT}$ and $(\text{FF or } (a \leftarrow l_3)) = a \leftarrow l_3$.
 Also, $(l_1 - a) = ((a_1 :: l_3) - a) = (a_1 \text{ =At } a) \rightarrow l_3 \mid a_1 :: (l_3 - a) = a_1 :: (l_3 - a)$. Now, $(\partial(l_2) \text{ and } (a \leftarrow l_1)) \text{ and } ((l_1 - a) \text{ =BA } l_2) = (\text{TT and } (a \leftarrow l_3)) \text{ and } ((a_1 :: (l_3 - a)) \text{ =BA } l_2) = (a \leftarrow l_3) \text{ and } (\partial(l_3 - a) \text{ and } (a_1 \leftarrow l_2) \text{ and } ((l_3 - a) \text{ =BA } (l_2 - a_1))) = (a \leftarrow l_3) \text{ and } ((\partial(a) \text{ and } \partial(l_3)) \text{ and } (a_1 \leftarrow l_2)) \text{ and } ((l_3 - a) \text{ =BA } (l_2 - a_1)) = (a \leftarrow l_3) \text{ and } ((\text{TT and TT}) \text{ and } (a_1 \leftarrow l_2)) \text{ and } ((l_3 - a) \text{ =BA } (l_2 - a_1)) = (a \leftarrow l_3) \text{ and } (a_1 \leftarrow l_2) \text{ and } ((l_3 - a) \text{ =BA } (l_2 - a_1))$.

The equality of the expression $(a \leftarrow l_3) \text{ and } (a_1 \leftarrow l_2) \text{ and } ((l_3 - a) \text{ =BA } (l_2 - a_1))$ with $(a \leftarrow l_2) \text{ and } (a \leftarrow l_3) \text{ and } ((l_3 - a) \text{ =BA } (l_2 - a_1))$ is easily shown by appealing to the simple (propositional) lemma, AndComm, from the theory PL:-

$\vdash \forall t_1 t_2 t_3: \text{tr.}$

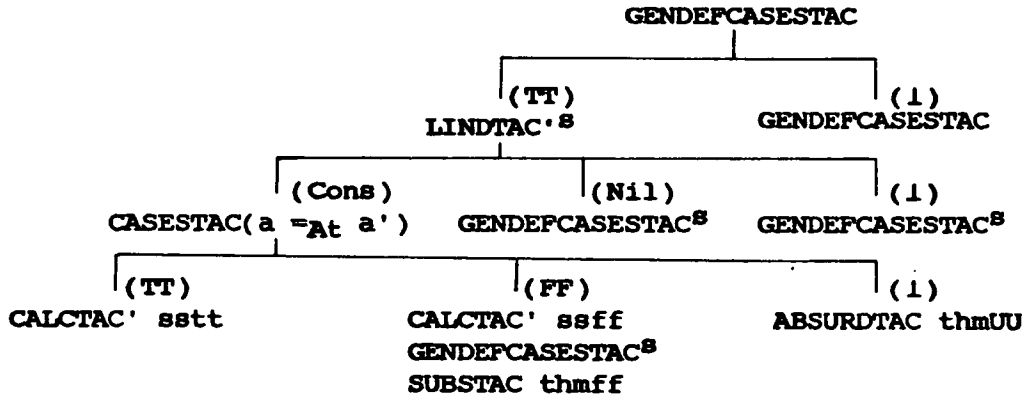
$\partial(t_1) = \text{TT} \ \& \ \partial(t_2) = \text{TT} \ \supset$

$t_1 \text{ and } (t_2 \text{ and } t_3) = t_2 \text{ and } (t_1 \text{ and } t_3)$

which is easily proven by truth value case analysis on the values of "t1" and "t2". Since $\partial(a \leftarrow l_3) = \partial(a_1 \leftarrow l_2) = \text{TT}$, this completes the proof of Lemma 4.8.

QED

A tactic which gives the above lemma is:-



where:-

$\text{sstt} = \{ \{ \text{SymEqAt}, (a \text{ =At } a') = \text{TT} \} \vdash (a' \text{ =At } a) = \text{TT} \}$

$\text{ssff} = \{ \{ \text{SymEqAt}, (a \text{ =At } a') = \text{FF} \} \vdash (a' \text{ =At } a) = \text{FF} \}$

$\text{thmff} =$

.... $\vdash (a \leftarrow l_1') \text{ and } ((a' \leftarrow l_2) \text{ and } ((l_1' - a) \text{ =BA } (l_2 - a')))) = (a' \leftarrow l_2) \text{ and } ((a \leftarrow l_1') \text{ and } ((l_1' - a) \text{ =BA } (l_2 - a'))))$
 (on the assumptions that $\partial(a) = \partial(a') = \partial(l_1') = \partial(l_2) = \text{TT}$.)

$\text{thmUU} = . \vdash \partial(a \text{ =At } a') = \partial(a) \text{ and } \partial(a')$

The theorem `thmff` above is a simplified instance of the lemma 'AndComm', from the theory PL. The parameterised tactic `CALCTAC` : `simpset` $\rightarrow$ tactic is one of a small set of simplification tactics introduced in Section 5.6, for including and applying extra `simprules` to a goal; this tactic simplifies with the union of the given and local `simpset`, and the goal's `simpset` remains unchanged.

In retrospect, lemma 4.8 should perhaps have been proven in two stages. The difficulty is that the induction hypothesis must act on a quantified variable in which a definedness cases analysis needs to be done BEFORE doing the induction. I would now proceed by first proving the following weaker lemma:-

$$\vdash \forall a:\alpha. \partial(a) = \text{TT} \supset$$

$$\forall l_1 l_2:(\alpha)L. \partial(l_2) = \text{TT} \supset$$

$$l_1 =_{\text{BA}}(a :: l_2) = (a \leftarrow l_1) \text{ and } ((l_1 - a) =_{\text{BA}} l_2)$$

The difference between this lemma and that above is that the critical definedness assumption is included within the scope of the quantifiers. This lemma's proof is similar to the last part of the one above; the leading quantifier on "a" is stripped off with its definedness assumption; a list induction is then done on " l_1 ", and as before, a case analysis on " $a =_{\text{At}} a$ ", is used to finish the proof. To prove the original lemma using this lemma, do definedness case analysis on "a" and " l_2 " and then apply this lemma when both variables are defined.

Theorem 4.9 (The symmetry of `EqBA`)

$$..]- \forall l_1 l_2. (l_1 =_{\text{BA}} l_2) = (l_2 =_{\text{BA}} l_1)$$

(on the assumption that both '`DEFEqAt`' and '`SymEqAt`' hold).

Proof

The first step is to do list induction on l_1 .

Suppose that $l_1 = \text{I}$. Then we have that $(\text{I} =_{\text{BA}} l_2) = \text{I} = (l_2 =_{\text{BA}} \text{I})$ since "`EqBA`" is bi-strict (by '`UUEqAt`' and '`EqAtUU`').

Suppose that $l_1 = \text{Nil}$. So, the desired result is

$\forall l_2. (\text{Nil} =_{\text{BA}} l_2) = (l_2 =_{\text{BA}} \text{Nil})$ which after simplification becomes $\forall l_2. \text{Null}(l_2) = (l_2 =_{\text{BA}} \text{Nil})$. We continue by list case analysis on l_2 .

Suppose that $l_2 = \text{I}$, then straightforwardly, we have that $\text{Null}(l_2) = \text{Null}(\text{I}) = \text{I} = (\text{I} =_{\text{BA}} \text{Nil}) = (l_2 =_{\text{BA}} \text{Nil})$.

Suppose that $l_2 = \text{Nil}$, then, by computation, $\text{Null}(l_2) = \text{Null}(\text{Nil}) = \text{TT} \equiv (\text{Nil} =_{\text{BA}} \text{Nil}) \equiv (l_2 =_{\text{BA}} \text{Nil})$.

Suppose that $l_2 = (a_1 :: l_3)$ where $\partial(a_1) = \partial(l_3) = \text{TT}$. (Note that no induction hypothesis is assumed here). Then $\text{Null}(l_2) = \text{Null}(a_1 :: l_3) = \text{not}(\partial(a_1) \text{ and } \partial(l_3)) \equiv \text{not}(\text{TT}) = \text{FF}$. Also, we have that $l_2 =_{\text{BA}} \text{Nil} \equiv (a_1 :: l_3) =_{\text{BA}} \text{Nil} \equiv (\partial(l_3) \text{ and } (a_1 \leftarrow \text{Nil})) \text{ and } (l_3 =_{\text{BA}} (\text{Nil} - a_1)) \equiv (\text{TT} \text{ and } (\partial(a_1) \Rightarrow \text{FF} \mid \perp)) \text{ and } (l_3 =_{\text{BA}} (\text{Nil} - a_1)) \equiv \text{FF} \text{ and } (l_3 =_{\text{BA}} (\text{Nil} - a_1)) \equiv \text{FF}$. So, in this case also, $\text{Null}(l_2) = \text{FF} \equiv l_2 =_{\text{BA}} \text{Nil}$.

This completes the case where $l_1 = \text{Nil}$. Suppose now that $l_1 = (a_1 :: l_3)$ where $\partial(a_1) = \partial(l_3) = \text{TT}$ with the induction hypothesis that:-

$$\forall l_4 . (l_3 =_{\text{BA}} l_4) \equiv (l_4 =_{\text{BA}} l_3)$$

Now, $l_1 =_{\text{BA}} l_2 \equiv (a_1 :: l_3) =_{\text{BA}} l_2 \equiv (\partial(l_3) \text{ and } (a_1 \leftarrow l_2)) \text{ and } (l_3 =_{\text{BA}} (l_2 - a_1)) \equiv (a_1 \leftarrow l_2) \text{ and } (l_3 =_{\text{BA}} (l_2 - a_1))$. Also, $l_2 =_{\text{BA}} l_1 \equiv l_2 =_{\text{BA}} (a_1 :: l_3) \equiv (\partial(l_3) \text{ and } (a_1 \leftarrow l_2)) \text{ and } ((l_2 - a_1) =_{\text{BA}} l_3) \equiv (a_1 \leftarrow l_2) \text{ and } ((l_2 - a_1) =_{\text{BA}} l_3)$ using lemma 4.8 (which depends upon the assumption 'SymEqAt'). Now, by using the induction hypothesis and putting " $(l_2 - a_1)$ " for " l_4 " we have that $l_3 =_{\text{BA}} (l_2 - a_1) \equiv (l_2 - a_1) =_{\text{BA}} l_3$. Now, $(l_1 =_{\text{BA}} l_2) \equiv (a_1 \leftarrow l_2) \text{ and } (l_3 =_{\text{BA}} (l_2 - a_1)) \equiv (a_1 \rightarrow l_2) \text{ and } ((l_2 - a_1) =_{\text{BA}} l_3) \equiv (l_2 =_{\text{BA}} l_1)$. This completes the proof of Theorem 4.9.

QED

Using list case analysis instead of induction in the above indicates the redundancy of the induction hypothesis in the proof to establish the "Cons" case. A tactic which proves the above lemma is:-

```

      LINDTAC'S
      |
      (Nil)
      |
    LCASESTAC'S
      |
      (Cons)
      |
    GENDEPCASESTAC'S*
```

where the initial simpset includes Lemma 4.8 as a simprule. The definedness case analyses are required, since the case analysis tactic LCASESTAC described below does not introduce these

assumptions directly.

LCASESTAC is a schematically generated tactic for applying case analyses over lists. The derivation of such tactics is briefly discussed in Section 5.2.2. The behaviour of the tactic is:-

$\forall l:(\alpha)L. F[l]$
ss
fml

LCASESTAC

$F[l]$	$F[Nil]$	$\forall l' a'. F[Cons a' l']$
ss	ss	ss
fml	fml	fml

where the (fresh) variables a' and l' have appropriate types and neither occur freely in the assumptions or the (original) goal formula. Note that this goal formula does not have to admit induction.

4.6.2 Multisets: Transitivity property of EqBA.

The next few lemmas develop certain intermediate congruence and other properties for $=_{BA}$, in preparation for the proof of transitivity, Theorem 4.16.

Lemma 4.10

....]- $\forall a_1 a_2:\alpha l:(\alpha)L.$

$$(a_1 =_{At} a_2) \equiv TT \supset (a_1 \leftarrow l) \equiv (a_2 \leftarrow l)$$

(on the assumption that "EqAt" is an equivalence predicate).

Proof

Taking " a_1 " and " a_2 " to be arbitrary values from an arbitrary domain, α , proceed by list induction on " l ".

Suppose that $l \equiv l$. Then $a_1 \leftarrow l \equiv a_1 \leftarrow l \equiv l \equiv a_2 \leftarrow l \equiv a_2 \leftarrow l$.

Suppose that $l \equiv Nil$, and assume that $(a_1 =_{At} a_2) \equiv TT$. Hence, $TT \equiv \partial(a_1 =_{At} a_2) \equiv \partial(a_1)$ and $\partial(a_2)$. So, using the 'AndAnalysis' lemma, we get that $\partial(a_1) \equiv TT \equiv \partial(a_2)$. Now, $a_1 \leftarrow Nil \equiv \partial(a_1) \rightarrow FF \mid l \equiv FF$, and also, $a_2 \leftarrow Nil \equiv \partial(a_2) \rightarrow FF \mid l \equiv FF$.

Suppose that $l \equiv (a_3 :: l_1)$ where $\partial(a_3) \equiv \partial(l_1) \equiv \text{TT}$ and with the induction hypothesis:-

$$(a_1 \text{ =At } a_2) \equiv \text{TT} \supset (a_1 \leftarrow l_1) \equiv (a_2 \leftarrow l_1).$$

Now, assume that $(a_1 \text{ =At } a_2) \equiv \text{TT}$. So, we have that $a_1 \leftarrow l \equiv a_2 \leftarrow (a_3 :: l_1) \equiv \partial(l_1)$ and $((a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv \text{TT}$ and $((a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv (a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1)$. Similarly, we have that $a_2 \leftarrow l \equiv a_2 \leftarrow (a_3 :: l_1) \equiv (a_2 \text{ =At } a_3) \text{ or } (a_2 \leftarrow l_1)$. Now, by using the induction hypothesis, we have that $a_1 \leftarrow l_1 \equiv a_2 \leftarrow l_1$, since $(a_1 \text{ =At } a_2) \equiv \text{TT}$, and so $(a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1) \equiv (a_1 \text{ =At } a_3) \text{ or } (a_2 \leftarrow l_1)$. Also, using the 'EqAtSelfCongruence' lemma (which depends upon 'DEFEqAt', 'SymEqAt' and 'TransEqAt') we have that (putting a_3 for a_4)"-

$$\begin{aligned} \dots]- (a_1 \text{ =At } a_2) \equiv \text{TT} \ \& \ (a_3 \text{ =At } a_3) \equiv \text{TT} \quad \supset \\ (a_1 \text{ =At } a_3) \equiv (a_2 \text{ =At } a_3) \end{aligned}$$

The first antecedent is immediate and the second antecedent follows from 'RefleEqAt' where $(a_3 \text{ =At } a_3) \equiv \partial(a_3) \equiv \text{TT}$. Hence, we can infer that $(a_1 \text{ =At } a_3) \equiv (a_2 \text{ =At } a_3)$, and so $(a_1 \leftarrow l) \equiv (a_1 \text{ =At } a_3) \text{ or } (a_2 \leftarrow l_1) \equiv (a_2 \text{ =At } a_3) \text{ or } (a_2 \leftarrow l_1) \equiv (a_2 \leftarrow l)$, completing the proof of Lemma 4.10.

QED

A tactic which proves this lemma is:-

```

GENDEFCASESTACS
GENDEFCASESTAC+
LINDTACS
|
(Cons)
|
IMPTAC'
CALCTAC' ssl

```

where

$$\text{ssl} = \{ \dots]- a' \text{ =At } a' \equiv \text{TT} \supset a_1 \text{ =At } a' \equiv a_2 \text{ =At } a' \}$$

The simprule contained in `ssl` above is obtained by specialising the 'EqAtSelfCongruence' lemma and discharging the first hypothesis (i.e. $a_1 \text{ =At } a_2 \equiv \text{TT}$). This will prevent the simprule from causing the simplifier to "loop", since the variables " a_1 " and " a_2 " are not instantiable as they will occur free among the hypotheses of the simprule.

Note that we have deviated from the informal proof by generating two definedness case analyses for the variables "a₁" and "a₂"; this is tactically smoother since otherwise these assumptions would have been obtained by ad-hoc forwards derivations, and then engaged as extra simplrules.

Lemma 4.11

...]- $\forall a_1 a_2: \alpha \ 1: (\alpha)L$.

$$(a_1 =_{At} a_2) = FF \supset a_1 \leftarrow (1 - a_2) = a_1 \leftarrow 1$$

(on the assumption that 'DEFEqAt', 'SymEqAt' and 'TransEqAt' hold).

Proof By list induction on 1

Suppose that $1 = 1$. Then $a_1 \leftarrow (1 - a_2) = a_1 \leftarrow (1 - a_2) = a \leftarrow 1 = 1$.

Suppose that $1 = Nil$, and also that $a_1 =_{At} a_2 = FF$. So, $\partial(a_1)$ and $\partial(a_2) = \partial(FF) = TT$. Now, by the propositional 'AndAnalysis' lemma, we obtain that $\partial(a_2) = TT$, and also $\partial(a_1) = TT$. So, we have that $a_1 \leftarrow (1 - a_2) = a_1 \leftarrow (Nil - a_2) = a_1 \leftarrow (\partial(a_2) \Rightarrow Nil \mid 1) = a_1 \leftarrow Nil = a_1 \leftarrow 1 = \partial(a_1) \Rightarrow FF \mid 1 = FF$.

Suppose that $1 = (a_3 :: l_1)$ where $\partial(a_3) = \partial(l_1) = TT$ and with the induction hypothesis that:

$$(a_1 =_{At} a_2) = FF \supset a_1 \leftarrow (l_1 - a_2) = a_1 \leftarrow l_1$$

We also assume $(a_1 =_{At} a_2) = FF$ and consequently obtain $\partial(a_1) = TT$, and also $\partial(a_2) = TT$ in the same way given above.

Proceeding with a truth value case analysis on the value of $(a_2 =_{At} a_3)$, suppose that $(a_2 =_{At} a_3) = 1$. But this contradicts the assumptions that both $\partial(a_2) = TT = \partial(a_3)$, since $1 = \partial(a_2 =_{At} a_3) = \partial(a_2)$ and $\partial(a_3) = TT$.

Suppose that $(a_2 =_{At} a_3) = TT$. Now, by applying the 'EqAtGenTrans' lemma (which depends upon the assumptions 'DEFEqAt', 'SymEqAt' and 'TransEqAt' to obtain:-

...]- $(a_3 =_{At} a_2) = TT \ \& \ (a_2 =_{At} a_1) = FF \supset (a_3 =_{At} a_1) = FF$ (by putting "FF" for "t", "a₃" for "a₁", and "a₁" for "a₃"). Both antecedents are obtained by use of 'SymEqAt' by $(a_3 =_{At} a_2) = (a_2 =_{At} a_3) = TT$, as well as $(a_2 =_{At} a_1) = (a_1 =_{At} a_2) = FF$. Hence, we have that $(a_3 =_{At} a_1) = FF$. So, by a further application of 'SymEqAt', we get $(a_1 =_{At} a_3) = (a_3 =_{At} a_1) = FF$. (This

deduction could also have been made using the lemma 'EqAt Self Congruence', but at the (minor) expense of using the lemma 'RefLEqAt' to obtain one of the antecedents.)

So, $(1 - a_2) \equiv ((a_3 :: l_1) - a_2) \equiv ((a_3 =_{At} a_2) \Rightarrow l_1 \mid a_3 :: (l_1 - a_2)) \equiv l_1$. Hence, $a_1 \leftarrow (1 - a_2) \equiv a_1 \leftarrow l_1$. On the other hand, $a_1 \leftarrow 1 \equiv a_1 \leftarrow (a_3 :: l_1) \equiv \partial(l_1) \text{ and } ((a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv \text{TT and } (\text{FF or } (a_1 \leftarrow l_1)) \equiv a_1 \leftarrow l_1$

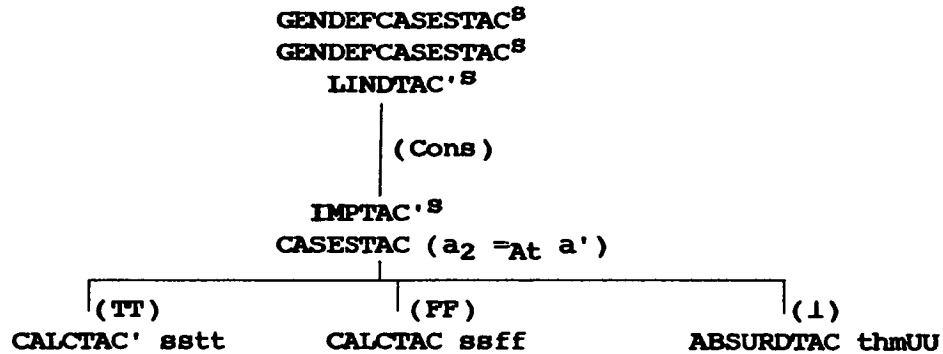
Suppose that $(a_2 =_{At} a_3) \equiv \text{FF}$. So, using 'SymEqAt' we have that $(a_3 =_{At} a_2) \equiv (a_2 =_{At} a_3) \equiv \text{FF}$. Now, $(1 - a_2) \equiv ((a_3 :: l_1) - a_2) \equiv (a_3 =_{At} a_2) \Rightarrow l_1 \mid a_3 :: (l_1 - a_2) \equiv a_3 :: (l_1 - a_2)$. Hence, $a_1 \leftarrow (1 - a_2) \equiv a_1 \leftarrow (a_3 :: (l_1 - a_2)) \equiv \partial(l_1 - a_2) \text{ and } ((a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow (l_1 - a_2)))$. Now, by the induction hypothesis, we have that $a_1 \leftarrow (l_1 - a_1) \equiv a_1 \leftarrow l_1$, since we may anyway assume that $(a_1 =_{At} a_2) \equiv \text{FF}$. Therefore, we have that $a_1 \leftarrow (1 - a_2) \equiv (\partial(l_1) \text{ and } \partial(a_2)) \text{ and } ((a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv \text{TT and } ((a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv (a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow l_1)$. However, we also have that $(a_1 \leftarrow 1) \equiv (a_1 \leftarrow (a_3 :: l_1)) \equiv \partial(l_1) \text{ and } ((a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv \text{TT and } ((a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv (a_1 =_{At} a_3) \text{ or } (a_1 \leftarrow l_1)$. This completes the proof of Lemma 4.11.

QED

As indicated within the proof above, dependence upon the assumption 'RefLEqAt' was avoided by use of the lemma 'EqAtGenTrans' instead of lemma 'EqAtSelfCongruence'. It can be seen that the actual proof used there resembles a matching case of the proof of the original 'SelfCongruence' lemma. This illustrates a fairly common phenomenon occurring in proofs; using a "general purpose" theorem can lead to weaker theorems that require more hypotheses than otherwise. These extra assumptions can arise in order to directly satisfy certain hypotheses of the theorem being appealed to.

A standard ploy which may remedy the situation is just to use (a portion of) its proof instead to possibly give a stronger result. A successful application of this was made in the above lemma. This could also have been applied elsewhere to remove some dependance upon the assumption 'RefEqAt' (where it arises due to

the way that the 'EqAtSelfCongruence' lemma is applied). However, this is not necessary in this case study since other lemmas will have to depend upon 'RefLEqAt' (e.g. the reflexivity theorem for "EqBA", Theorem 4.6). A tactic for proving the above lemma is:-



where:-

$\text{sstt} = \{ \dots \} - a_1 =_{\text{At}} a' = \text{FF} \}$

$\text{ssff} = \{ \dots \} - a' =_{\text{At}} a_2 = \text{FF} \}$

$\text{thmUU} = \dots - \partial(a_2 =_{\text{At}} a') = \partial(a_2) \text{ and } \partial(a')$

The theorem thmUU is obtained by specialising the (global) assumption 'DEFEqAt'. The simp rule in ssff is obtained by specialising the assumption 'SymEqAt' and simplifying with the assumption that " $a_2 =_{\text{At}} a' = \text{FF}$ ". Finally, the theorem used in the simpset sstt is obtained by specialising 'EqAtGenTrans' once and 'SymEqAt' three times and then simplifying with the assumptions that $a_2 =_{\text{At}} a' = \text{TT}$, and $a_1 =_{\text{At}} a_2 = \text{FF}$ as appropriate.

Lemma 4.12

$\dots \} - \forall a: \alpha \ l_1 \ l_2: (\alpha)L .$

$(l_1 =_{\text{BA}} l_2) \equiv \text{TT} \supset a \leftarrow l_1 = a \leftarrow l_2$

(on the assumption that "EqAt" is an equivalence predicate).

Proof

The first step is to do a definedness case analysis on the value of "a".

Suppose that $\partial(a) \equiv \perp$, and hence $a \equiv \perp$. Then $a \leftarrow l_1 \equiv \perp \leftarrow l_1 \equiv \perp \equiv \perp \leftarrow l_2 \equiv a \leftarrow l_2$

Suppose that $\partial(a) \equiv \text{TT}$, and continuing with list induction on l_1 , suppose that $l_1 \equiv \perp$. This falsifies the antecedent of the

desired result, since we have that $1 \equiv (1 \text{ =BA } l_2) \equiv (l_1 \text{ =BA } l_2) \equiv \text{TT}$.

Suppose that $l_1 \equiv \text{Nil}$ and assume that $(l_1 \text{ =BA } l_2) \equiv \text{TT}$. So, substituting we have that $\text{TT} \equiv (l_1 \text{ =BA } l_2) \equiv \text{Nil} \text{ =BA } l_2 \equiv \text{Null}(l_2)$. By the 'NullIMPNil' lemma, this shows that $l_2 \equiv \text{Nil}$, and so $l_1 \equiv l_2$. Therefore, $a \leftarrow l_1 \equiv a \leftarrow l_2$.

Suppose that $l_1 \equiv a_1 :: l_3$ where $\partial(a_1) \equiv \partial(l_3) \equiv \text{TT}$ and with the induction hypothesis that:-

$$\forall l_4:(\alpha)L. (l_3 \text{ =BA } l_4) \equiv \text{TT} \supset a \leftarrow l_3 \equiv a \leftarrow l_4$$

We also assume that $l_1 \text{ =BA } l_2 \equiv \text{TT}$. So, by simplification, this gives $\text{TT} \equiv l_1 \text{ =BA } l_2 \equiv (a_1 :: l_3) \text{ =BA } l_2 \equiv (\partial(l_3) \text{ and } (a_1 \leftarrow l_2)) \text{ and } (l_3 \text{ =BA } (l_2 - a_1)) \equiv (a_1 \leftarrow l_2) \text{ and } (l_3 \text{ =BA } (l_2 - a_1))$. So, using the 'AndAnalysis' lemma, we have that $a_1 \leftarrow l_2 \equiv \text{TT}$, and also $l_3 \text{ =BA } (l_2 - a_1) \equiv \text{TT}$. Now, $a \leftarrow l_1 \equiv a \leftarrow (a_1 :: l_3) \equiv \partial(l_3) \text{ and } ((a \text{ =At } a_1) \text{ or } (a \leftarrow l_3)) \equiv \text{TT and } ((a \text{ =At } a_1) \text{ or } (a \leftarrow l_3)) \equiv (a \text{ =At } a_1) \text{ or } (a \leftarrow l_3)$.

We continue by doing a truth value case analysis on the value of $(a \text{ =At } a_1)$. Suppose that $(a \text{ =At } a_1) \equiv 1$. Then, using the assumption 'DEFEqAt', we have that $1 \equiv \partial(a \text{ =At } a_1) \equiv \partial(a) \text{ and } \partial(a_1) \equiv \text{TT}$, which is a contradiction.

Suppose that $(a \text{ =At } a_1) \equiv \text{TT}$. Then, by using Lemma 4.10 (with "a" for "a₁", "a₁" for "a₂" and "l₂" for "l"), we obtain:-

$$\dots]- (a \text{ =At } a_1) \equiv \text{TT} \supset a \leftarrow l_2 \equiv a_1 \leftarrow l_2$$

Hence, we have that $a \leftarrow l_2 \equiv a_1 \leftarrow l_2 \equiv \text{TT}$. Moreover, $a \leftarrow l_1 \equiv (a \text{ =At } a_1) \text{ or } (a \leftarrow l_3) \equiv \text{TT or } (a \leftarrow l_3) \equiv \text{TT} \equiv a \leftarrow l_2$.

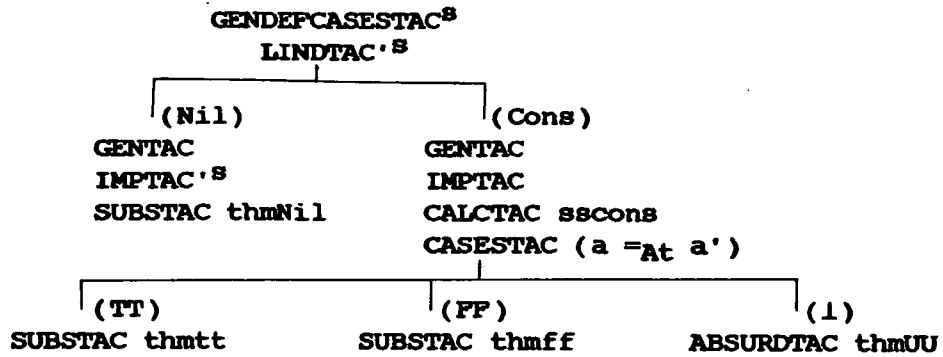
Suppose that $(a \text{ =At } a_1) \equiv \text{FF}$, and so $a \leftarrow l_1 \equiv (a \text{ =At } a_1) \text{ or } (a \leftarrow l_3) \equiv \text{FF or } (a \leftarrow l_3) \equiv a \leftarrow l_3$. Now, $l_3 \text{ =BA } (l_2 - a_1) \equiv \text{TT}$ and so by the induction hypothesis (putting " $(l_2 - a_1)$ " for " l_4 ") we have that $l_3 \text{ =BA } (l_2 - a_1) \equiv \text{TT} \supset a \leftarrow l_3 \equiv a \leftarrow (l_2 - a_1)$. Hence, we find that $a \leftarrow l_1 \equiv a \leftarrow l_3 \equiv a \leftarrow (l_2 - a_1)$. Now, by applying Lemma 4.11 (with "a" for "a₁", "a₁" for "a₂" and "l₂" for "l") to show that

$$\dots]- (a \text{ =At } a_1) \equiv \text{FF} \supset a \leftarrow (l_2 - a_1) \equiv a \leftarrow l_2$$

Hence, this gives $a \leftarrow l_1 \equiv a_1 \leftarrow (l_2 - a_1) \equiv a \leftarrow l_2$. This completes the proof of Lemma 4.12.

QED

A tactic which proves this lemma is:-



where

$\text{thmUU} = \dots]- \partial(a =_{\text{At}} a') \equiv \partial(a) \text{ and } \partial(a')$
 $\text{thmNil} = \dots]- l_2 \equiv \text{Nil}$
 $\text{thmtt} = \dots]- a \leftarrow l_2 \equiv a' \leftarrow l_2$
 $\text{thmff} = \dots]- a \leftarrow (l_2 - a') \equiv a \leftarrow l_2$
 $\text{sscons} = \{ \dots]- a' \leftarrow l_2 \equiv \text{TT}, \dots]- l' =_{\text{BA}} (l_2 - a') \equiv \text{TT} \}$

As usual, thmUU is obtained by specialising 'DEFEqAt'; thmNil is a specialisation of the lemma 'NullIMPNil'; thmff is a specialisation of Lemma 4.11, and thmtt is a specialisation of Lemma 4.10. All antecedents of these theorems were assumed as hypotheses, permitting their use with SUBSTAC, or as (non-looping) simprules. The two theorems comprising sscons are obtained by specialising the 'AndAnalysis' lemma, assuming the antecedent, and then applying each of the conjunction elimination rules to the consequent.

Lemma 4.13

$\dots]- \forall a_1 a_2 : \alpha \ l : (\alpha) L. (a_1 =_{\text{At}} a_2) \equiv \text{TT} \supset l - a_1 \equiv l - a_2$
 (on the assumption that "EqAt" is an equivalence predicate).

Proof By list induction on l

Suppose that $l \equiv l$. Then we have that $l - a_1 \equiv l - a_1 \equiv l \equiv l - a_1 \equiv l \equiv l - a_2 \equiv l - a_2$.

Suppose that $l \equiv \text{Nil}$, and also that $(a_1 =_{\text{At}} a_2) \equiv \text{TT}$. So, using the 'DEFEqAt' property, we have that $\text{TT} \equiv \partial(a_1 =_{\text{At}} a_2) \equiv \partial(a_1) \text{ and } \partial(a_2)$, and so using the 'AndAnalysis' lemma we have that $\partial(a_1) \equiv \text{TT}$, and also $\partial(a_2) \equiv \text{TT}$. Now, $l - a_1 \equiv \text{Nil} - a_1 \equiv \partial(a_1) \Rightarrow \text{Nil} \mid l \equiv \text{Nil} \equiv \partial(a_2) \Rightarrow \text{Nil} \mid l \equiv \text{Nil} - a_2 \equiv l - a_2$.

Suppose that $l \equiv a_3 :: l_1$ where $\partial(a_3) \equiv TT \equiv \partial(l_1)$ and with the induction hypothesis:-

$$(a_1 =_{At} a_2) \equiv TT \supset l_1 - a_1 \equiv l_1 - a_2$$

We may also assume that $(a_1 =_{At} a_2) \equiv TT$. Now, consider $l - a_1 \equiv (a_3 :: l_1) - a_1 \equiv (a_3 =_{At} a_1) \neq l_1 \mid a_3 :: (l_1 - a_1)$. Using the induction hypothesis we have that $(l_1 - a_1) \equiv (l_2 - a_2)$, and so, substituting, this gives:- $l - a_1 \equiv ((a_3 =_{At} a_1) \neq l_1 \mid a_3 :: (l_1 - a_2))$. Next, apply the 'EqAtSelfCongruence' lemma (which depends upon 'DEFEqAt', 'SymEqAt' and 'TransEqAt') to get

$$\dots]- (a_3 =_{At} a_3) \equiv TT \ \& \ (a_1 =_{At} a_2) \equiv TT \supset$$

$$(a_3 =_{At} a_1) \equiv (a_3 =_{At} a_2)$$

(i.e. putting "a₃" for "a₁" and "a₂", "a₁" for "a₃" and "a₂" for "a₄"). The first antecedent is easily obtained by using 'RefLEqAt' to give $(a_3 =_{At} a_3) \equiv \partial(a_3) \equiv TT$; the second antecedent is directly available. Hence, we have that $(a_3 =_{At} a_1) \equiv (a_3 =_{At} a_2)$. So, substituting in the above, we get that $((a_3 =_{At} a_1) \neq l_1 \mid a_3 :: (l_1 - a_2)) \equiv (a_3 =_{At} a_2) \neq l_1 \mid a_3 :: (l_1 - a_2) \equiv (a_3 :: l_1) - a_2 \equiv l - a_2$, which completes the proof of lemma 4.13.

QED

A tactic which proves this lemma is:-

```

GENDEFCASESTACS
GENDEFCASESTACS
LINDTACS
|
(Cons)
|
IMPTACS
CALCTACS sscons

```

where:-

$$sscons = \{ \dots]- a' =_{At} a' \equiv TT \supset a' =_{At} a_1 \equiv a' =_{At} a_2 \}$$

This theorem is obtained by specialising the 'EqAtSelfCongruence' lemma and assuming the the second antecedent; this prevents the conditional simp rule from causing the simplifier to loop, since the variables "a₁" and "a₂" are then not instantiable.

Lemma 4.14

....]- $\forall a b: \alpha \text{ l}:(\alpha)L. (1 - a) - b \equiv (1 - b) - a$

(on the assumption that "EqAt" is an equivalence predicate).

Proof

First of all, do a definedness case analysis on the variable "a". Suppose that $\partial(a) \equiv 1$ and hence $a \equiv 1$. Then $(1 - a) - b \equiv (1 - 1) - b \equiv 1 - b \equiv 1 \equiv (1 - b) - 1 \equiv (1 - b) - a$.

Suppose that $\partial(a) \equiv \text{TT}$, and then do a definedness case analysis on the variable "b". Suppose that $\partial(b) \equiv 1$ and hence $b \equiv 1$. Then, as above, $(1 - a) - b \equiv (1 - a) - 1 \equiv 1 \equiv 1 - a \equiv (1 - 1) - a \equiv (1 - b) - a$.

So now suppose that both $\partial(a) \equiv \text{TT}$, and that $\partial(b) \equiv \text{TT}$, we shall proceed by truth value case analysis on the value of $(a =_{\text{At}} b)$.

Suppose that $(a =_{\text{At}} b) \equiv 1$. This directly contradicts the definedness of "a" and "b", since $1 \equiv \partial(a =_{\text{At}} b) \equiv \partial(a)$ and $\partial(b) \equiv \text{TT}$.

Suppose that $(a =_{\text{At}} b) \equiv \text{TT}$. By using Lemma 4.13, we have that $\forall \text{ l}:(\alpha)L. 1 - a = 1 - b$, since $(a =_{\text{At}} b) \equiv \text{TT}$. So, clearly, we have that $(1 - a) - b \equiv (1 - b) - b \equiv (1 - b) - a$.

Suppose that $(a =_{\text{At}} b) \equiv \text{FF}$, and proceed with list induction on 1 .

Suppose that $1 \equiv 1$. Then $(1 - a) - b \equiv (1 - a) - b \equiv 1 - b \equiv 1 \equiv 1 - a \equiv (1 - b) - a \equiv (1 - b) - a$.

Suppose that $1 \equiv \text{Nil}$. Then $(1 - a) - b \equiv (\text{Nil} - a) - b \equiv (\partial(a) \Rightarrow \text{Nil} \mid 1) - b \equiv \text{Nil} - b \equiv \partial(b) \Rightarrow \text{Nil} \mid 1 \equiv \text{Nil}$. Similarly, $(1 - b) - a \equiv \text{Nil}$.

Suppose that $1 \equiv (c :: l_1)$ where $\partial(c) \equiv \partial(l_1) \equiv \text{TT}$ with the induction hypothesis that $(l_1 - a) - b \equiv (l_1 - b) - a$. Then we have that $1 - a \equiv (c :: l_1) - a \equiv (c =_{\text{At}} a) \Rightarrow l_1 \mid c :: (l_1 - a)$, and also $1 - b \equiv (c :: l_1) - b \equiv (c =_{\text{At}} b) \Rightarrow l_1 \mid c :: (l_1 - b)$.

Proceeding now by truth value case analysis on the value of $(c =_{\text{At}} a)$, suppose that $(c =_{\text{At}} a) \equiv 1$. This directly contradicts the definedness of "c" and "a", since $1 \equiv \partial(c =_{\text{At}} a) \equiv \partial(c)$ and $\partial(a) \equiv \text{TT}$.

Suppose that $(c =_{\text{At}} a) \equiv \text{TT}$. Now, by the lemma

'EqAtSelfCongruence' (which depends upon 'DEFEqAt', 'SymEqAt' and 'TransEqAt') to get

$$\dots] - (c =_{At} a) \equiv TT \ \& \ (b =_{At} b) \equiv TT \ \supset \ (c =_{At} b) \equiv (a =_{At} b)$$

The first antecedent is already available, and the second is gotten by applying 'RefLEqAt' with $(b =_{At} b) \equiv \partial(b) \equiv TT$. Hence, we have that $(c =_{At} b) \equiv (a =_{At} b) \equiv FF$. Now, evaluating, we have that $1 - a \equiv l_1$, and also $1 - b \equiv c :: (l_1 - b)$. So, $(1 - b) - a \equiv (c :: (l_1 - b)) - a \equiv (c =_{At} a) \Rightarrow (l_1 - b) \mid c :: ((l_1 - b) - a) \equiv l_1 - b \equiv (1 - a) - b$.

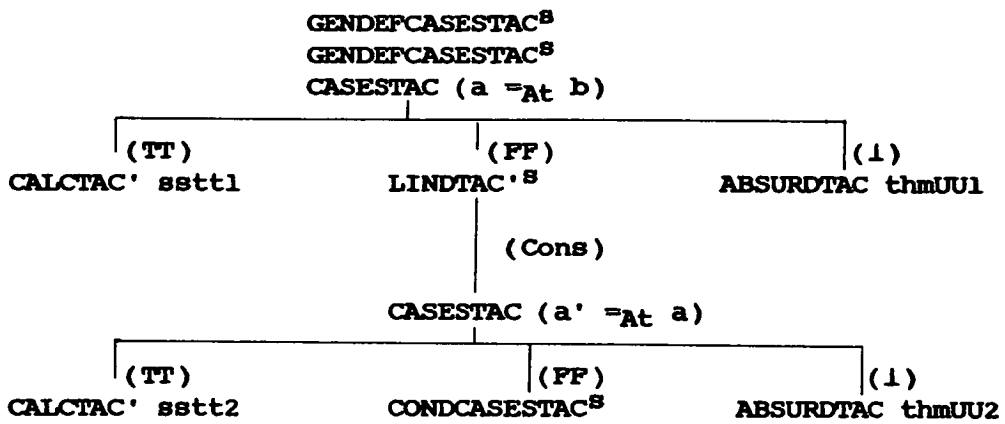
Suppose that $(c =_{At} a) \equiv FF$. Now this gives $(1 - a) - b \equiv (c :: (l_1 - a)) - b \equiv (c =_{At} b) \Rightarrow (l_1 - a) \mid c :: ((l_1 - a) - b) \equiv (c =_{At} b) \Rightarrow (l_1 - a) \mid c :: ((l_1 - b) - a)$, using the induction hypothesis. Continuing with truth value case analysis on the value of $(c =_{At} b)$, suppose that $(c =_{At} b) \equiv 1$. This contradicts the definedness of "c" and "b", since $1 \equiv \partial(c =_{At} b) \equiv \partial(c)$ and $\partial(b) \equiv TT$.

Suppose that $(c =_{At} b) \equiv TT$. Then we have that $(1 - a) - b \equiv (c =_{At} b) \Rightarrow (l_1 - a) \mid c :: (l_1 - b) - a \equiv l_1 - a$, and also $1 - b \equiv (c =_{At} b) \Rightarrow l_1 \mid c :: (l_1 - b) \equiv l_1$. Hence, $(1 - a) - b \equiv l_1 - a \equiv (1 - b) - a$.

Suppose that $(c =_{At} b) \equiv FF$. Then we have that $(1 - a) - b \equiv c :: (l_1 - b) - a$ and also $1 - b \equiv c :: (l_1 - b)$. Hence, $(1 - b) - a \equiv (c :: (l_1 - b)) - a \equiv ((c =_{At} a) \Rightarrow (l_1 - b) \mid c :: ((l_1 - b) - a)) \equiv c :: ((l_1 - b) - a) \equiv (1 - a) - b$, since we originally assumed that $(c =_{At} a) \equiv FF$. This completes the proof of Lemma 4.14.

QED

A tactic which proves this lemma is:-



where

$$\text{thmUU1} = \dots] - \partial(a \text{ =At } b) \equiv \partial(a) \text{ and } \partial(b)$$

$$\text{thmUU2} = \dots] - \partial(a' \text{ =At } a) \equiv \partial(a') \text{ and } \partial(a)$$

$$\text{sstt1} = \{ \dots] - \forall l. l - a \equiv l - b \}$$

$$\text{sstt2} = \{ \dots] - b \text{ =At } b \equiv \text{TT} \supset a' \text{ =At } b \equiv a \text{ =At } b \}$$

The theorems thmUU1 and thmUU2 are each obtained by specialising 'DEFEqAt'; the theorem in simpset sstt1 is obtained by specialising Lemma 4.13 and the theorem in the simpset sstt2 is obtained by specialising the 'EqAtSelfCongruence' lemma. As usual, various antecedents have been assumed as hypotheses to make certain variables non-instantiable during simplification.

Lemma 4.15

$$\dots] - \forall a_1 a_2 : \alpha \quad l_1 l_2 : (\alpha)L .$$

$$(a_1 \text{ =At } a_2) \equiv \text{TT} \quad \& \quad (l_1 \text{ =BA } l_2) \equiv \text{TT} \supset$$

$$(l_1 - a_1) \text{ =BA } (l_2 - a_2) \equiv \text{TT}$$

(on the assumption the "EqAt" is an equivalence predicate).

Proof By list induction on l_1

Suppose that $l_1 \equiv l$. This contradicts the second antecedent, since $l \equiv (l \text{ =BA } l_2) \equiv (l_1 \text{ =BA } l_2) \equiv \text{TT}$.

Suppose that $l_1 \equiv \text{Nil}$ and also assume that both $(a_1 \text{ =At } a_2) \equiv \text{TT}$, as well as $(l_1 \text{ =BA } l_2) \equiv \text{TT}$. Now, we have that $\text{TT} \equiv \partial(a_1 \text{ =At } a_2) \equiv \partial(a_1) \text{ and } \partial(a_2)$, using the 'DEFEqAt' assumption. So, by applying 'And Analysis' we obtain $\partial(a_1) \equiv \text{TT}$, and also $\partial(a_2) \equiv \text{TT}$. Consider that $\text{TT} \equiv l_1 \text{ =BA } l_2 \equiv \text{Nil} \text{ =BA } l_2 \equiv \text{Null}(l_2)$. So, applying the 'NullIMPNil' lemma, we have that $l_2 \equiv \text{Nil}$ and so

$l_1 \equiv l_2 \equiv \text{Nil}$. Then $l_1 - a_1 \equiv \text{Nil} - a_1 \equiv \partial(a_1) \Rightarrow \text{Nil} \mid 1 \equiv \text{Nil} \equiv \partial(a_2) \Rightarrow \text{Nil} \mid 1 \equiv \text{Nil} - a_2 \equiv l_2 - a_2$.

Hence, $(l_1 - a_1) \equiv_{\text{BA}} (l_2 - a_2) \equiv \text{Nil} \equiv_{\text{BA}} \text{Nil} \equiv \text{Null}(\text{Nil}) \equiv \text{TT}$.

Suppose that $l_1 \equiv (a_3 :: l_3)$ where $\partial(a_3) \equiv \partial(l_3) \equiv \text{TT}$ with the induction hypothesis that:-

$\forall l_4. (a_1 \equiv_{\text{At}} a_2) \equiv \text{TT} \ \& \ (l_3 \equiv_{\text{BA}} l_4) \equiv \text{TT} \supset$

$(l_3 - a_1) \equiv_{\text{BA}} (l_4 - a_2) \equiv \text{TT}$

We may also assume that $(a_1 \equiv_{\text{At}} a_2) \equiv \text{TT}$ and that $l_1 \equiv_{\text{BA}} l_2 \equiv \text{TT}$. As in the previous case, we show that $\partial(a_1) \equiv \text{TT}$, and also that $\partial(a_2) \equiv \text{TT}$, using 'DEFEqAt' and 'And Analysis'. Now, $\text{TT} \equiv l_1 \equiv_{\text{BA}} l_2 \equiv (a_3 :: l_3) \equiv_{\text{BA}} l_2 \equiv (\partial(l_3) \text{ and } (a_3 \leftarrow l_2)) \text{ and } (l_3 \equiv_{\text{BA}} (l_2 - a_3)) \equiv (\text{TT} \text{ and } (a_3 \leftarrow l_2)) \text{ and } (l_3 \equiv_{\text{BA}} (l_2 - a_3)) \equiv (a_3 \leftarrow l_2) \text{ and } (l_3 \equiv_{\text{BA}} (l_2 - a_3))$. So, applying the 'And Analysis' lemma we get that $(a_3 \rightarrow l_2) \equiv \text{TT}$, and also $(l_3 \equiv_{\text{BA}} (l_2 - a_3)) \equiv \text{TT}$. Now, $(l_1 - a_1) \equiv ((a_3 :: l_3) - a_1) \equiv ((a_3 \equiv_{\text{At}} a_1) \Rightarrow l_3 \mid a_3 :: (l_3 - a_1))$.

Continuing with a truth valued case analysis on the value of $(a_3 \equiv_{\text{At}} a_1)$, suppose that $(a_3 \equiv_{\text{At}} a_1) \equiv 1$. However, this gives a contradiction, since $1 \equiv \partial(a_3 \equiv_{\text{At}} a_1) \equiv \partial(a_3) \text{ and } \partial(a_1) \equiv \text{TT}$, assuming the 'DEFEqAt' property.

Suppose that $(a_3 \equiv_{\text{At}} a_1) \equiv \text{TT}$. Then, using the 'TransEqAt' property, with "a₃" for "a₁", "a₁" for "a₂" and "a₂" for "a₃" we have that:-

.]- $(a_3 \equiv_{\text{At}} a_1) \equiv \text{TT} \ \& \ (a_1 \equiv_{\text{At}} a_2) \equiv \text{TT} \supset (a_3 \equiv_{\text{At}} a_2) \equiv \text{TT}$

Both of these antecedents are already available and so we have that $(a_3 \equiv_{\text{At}} a_2) \equiv \text{TT}$. Now, by using Lemma 4.13, with "a₃" for "a₁" and "l₂" for "l", we get:-

....]- $(a_3 \equiv_{\text{At}} a_2) \equiv \text{TT} \supset (l_2 - a_3) \equiv (l_2 - a_2)$

Hence, we can show that $(l_2 - a_3) \equiv (l_2 - a_2)$. So, $(l_1 - a_1) \equiv (a_3 \equiv_{\text{At}} a_1) \Rightarrow l_3 \mid a_3 :: (l_3 - a_1) \equiv l_1$. Therefore,

$(l_1 - a_1) \equiv_{\text{BA}} (l_2 - a_2) \equiv l_3 \equiv_{\text{BA}} (l_2 - a_2) \equiv l_3 \equiv_{\text{BA}} (l_2 - a_3) \equiv \text{TT}$.

Suppose that $(a_3 \equiv_{\text{At}} a_1) \equiv \text{FF}$. So, by applying the lemma 'EqAtSelfCongruence', with "a₃" for "a₁" and "a₂", "a₁" for "a₃" and "a₂" for "a₄", we get that:-

...]- $(a_3 \equiv_{\text{At}} a_3) \equiv \text{TT} \ \& \ (a_1 \equiv_{\text{At}} a_2) \equiv \text{TT} \supset$

$(a_3 \equiv_{\text{At}} a_1) \equiv (a_3 \equiv_{\text{At}} a_2)$

The first antecedent follows by using 'ReflEqAt' since $(a_3 =_{At} a_3) = \partial(a_3) = TT$; the second antecedent is already available, and so they together show that $(a_3 =_{At} a_1) = (a_3 =_{At} a_2)$. Hence,
 $(a_3 =_{At} a_2) = (a_3 =_{At} a_1) = FF$.

Now, $(l_1 - a_1) = (a_3 =_{At} a_1) \Rightarrow l_3 \mid a_3 :: (l_3 - a_1) = a_3 :: (l_3 - a_1)$. Hence, $(l_1 - a_1) =_{BA} (l_2 - a_2) = a_3 :: (l_3 - a_1) =_{BA} (l_2 - a_2) = (\partial(l_3 - a_1) \text{ and } (a_3 \leftarrow (l_2 - a_2)))$
and $((l_3 - a_1) =_{BA} ((l_2 - a_2) - a_3)) = ((\partial(l_3) \text{ and } \partial(a_1)) \text{ and } (a_3 \leftarrow (l_2 - a_2)))$
and $((l_3 - a_1) =_{BA} ((l_2 - a_2) - a_3)) = (a_3 \leftarrow (l_2 - a_2)) \text{ and } ((l_3 - a_1) =_{BA} ((l_2 - a_2) - a_3)) =$
 $(a_3 \leftarrow (l_2 - a_2)) \text{ and } ((l_3 - a_1) =_{BA} ((l_2 - a_2) - a_3)).$

Now, by Lemma 4.11, with "a₃" for "a₁", "a₁" for "a₂" and "l₂" for "l", we have that:-

$$....] - (a_3 =_{At} a_2) = FF \supset a_3 \leftarrow (l_2 - a_2) = a_3 \leftarrow l_2$$

Since the antecedent is already available, we have that

$a_3 \leftarrow (l_2 - a_2) = a_3 \leftarrow l_2$. Also, by Lemma 4.14, with "a₂" for "a", "a₃" for "b", and "l₂" for "l", we have that:-

$$....] - (l_3 - a_2) - a_3 = (l_3 - a_3) - a_2$$

These results then show that $(l_1 - a_1) =_{BA} (l_2 - a_2) = (a_3 \leftarrow (l_2 - a_2)) \text{ and } ((l_3 - a_1) =_{BA} ((l_2 - a_2) - a_3)) =$
 $(a_3 \leftarrow l_2) \text{ and } ((l_3 - a_1) =_{BA} ((l_2 - a_3) - a_2)) =$
 $TT \text{ and } ((l_3 - a_1) =_{BA} ((l_2 - a_3) - a_2)).$

Now, by applying the induction hypothesis, with $(l_2 - a_3)$ for "l₄", we have:-

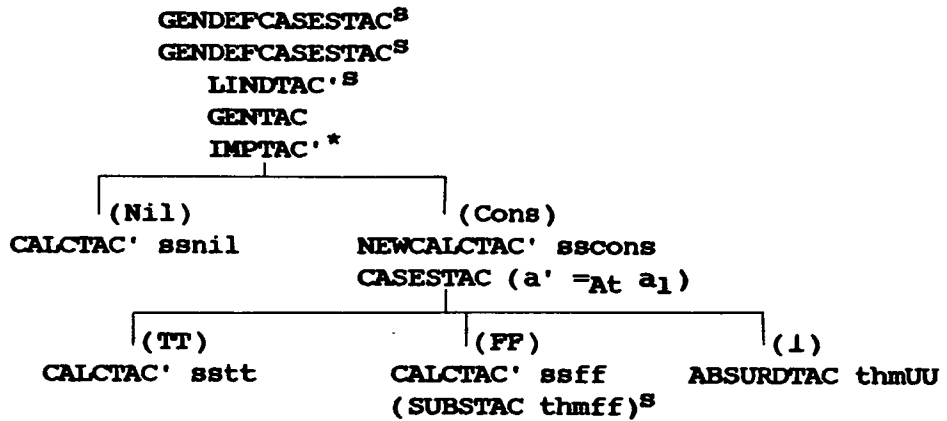
$$.] - (a_1 =_{At} a_2) = TT \ \& \ (l_3 =_{At} (l_2 - a_3)) = TT \supset \\ (l_3 - a_1) =_{BA} ((l_2 - a_3) - a_2) = TT$$

Both of these antecedents are already available and so we have that $(l_3 - a_1) =_{BA} ((l_2 - a_3) - a_2) = TT$.

This completes the proof of Lemma 4.15.

QED

A tactic which proves this lemma is:-



where

$$\begin{aligned}
 \text{thmUU} &= .\text{.}] - \partial(a' =_{\text{At}} a_1) \equiv \partial(a') \text{ and } \partial(a_1) \\
 \text{thmff} &= \dots.] - (l_1' - a_2) - a' \equiv (l_1' - a') - a_2 \\
 \text{ssnil} &= \{ .\text{.}] - l_2 = \text{Nil} \} \\
 \text{sscons} &= \{ .\text{.}] - a_3 \leftarrow l_2 \equiv \text{TT}, .\text{.}] - l_1' =_{\text{BA}} (l_2 - a') \equiv \text{TT} \} \\
 \text{sstt} &= \{ \dots.] - a' =_{\text{At}} a_2 \equiv \text{TT}, \\
 &\quad \dots\dots.] - (l_2 - a') \equiv (l_2 - a_2) \} \\
 \text{ssff} &= \{ \dots.] - a' =_{\text{At}} a' \equiv \text{TT} \supset a' =_{\text{At}} a_2 \equiv a' =_{\text{At}} a_1, \\
 &\quad \dots\dots.] - a' =_{\text{At}} a_2 \equiv \text{FF} \supset a' \leftarrow (l_2 - a_2) \equiv a' \leftarrow l_2 \}
 \end{aligned}$$

The theorem thmUU is standard; theorem thmff is obtained by specialising Lemma 4.14 and since the rhs matches the lhs, it cannot be used as a simprule and has to be applied via SUBSTAC. The theorem contained in ssnil is obtained by specialising the Lemma 'NullIMPNil' and then assuming the antecedent, making it suitable for use as a simprule. The two theorems contained in the simpset sscons are obtained by specialising the 'AndAnalysis' lemma, assuming the antecedent and applying conjunction elimination.

The first theorem in the simpset sstt is obtained by specialising 'TransEqAt' and assuming the antecedents; the second theorem is obtained by specialising Lemma 4.13 and then eliminating the antecedent by using the first theorem with Modus Ponens.

Finally, the simpset ssff contains two theorems, the first of which is obtained by specialising 'EqAtSelfCongruence' and eliminating the second antecedent. This is done by using the (minor) lemma that $\dots.] - a_2 =_{\text{At}} a_1 \equiv \text{TT}$, which is proven using 'SymEqAt' and the assumption that $a_1 =_{\text{At}} a_2 \equiv \text{TT}$. The second theorem in ssff is proven by specialising Lemma 4.14.

The parameterised tactic `NEWCALCTAC' : thm → tactic`, acts like `CALCTAC'`, except that the resulting goal's simpset becomes the union of the given and local simpset, after simplification (see Section 5.6)

Theorem 4.16 (The transitivity of `EqBA`)

....]- $\forall l_1 l_2 l_3 : (\alpha)L.$

$(l_1 =_{BA} l_2) \equiv TT \ \& \ (l_2 =_{BA} l_3) \equiv TT \supset$

$l_1 =_{BA} l_3 \equiv TT$

(on the assumption that "`EqAt`" is an equivalence predicate.)

Proof By list induction on l_1 .

Suppose that $l_1 \equiv I$. This contradicts the first antecedent, since $I \equiv I =_{BA} l_2 \equiv l_1 =_{BA} l_2 \equiv TT$.

Suppose that $l_1 \equiv Nil$, and also assume that $l_1 =_{BA} l_2 \equiv TT$, and also that $l_2 =_{BA} l_3 \equiv TT$. So, simplyfying, we have $TT \equiv l_1 =_{BA} l_2 \equiv Nil =_{BA} l_2 \equiv Null(l_2)$. Now, using the '`NullIMPNil`' lemma, we have that $l_2 \equiv Nil$ also. Hence, $l_1 \equiv Nil \equiv l_2$, and so $l_1 =_{BA} l_3 \equiv l_2 =_{BA} l_3 \equiv TT$.

Suppose that $l_1 \equiv (a_1 :: l_4)$ where $\partial(a_1) \equiv \partial(l_4) \equiv TT$ and with the induction hypothesis:-

$\forall l_5 l_6. (l_4 =_{BA} l_5) \equiv TT \ \& \ (l_5 =_{BA} l_6) \equiv TT \supset$

$(l_4 =_{BA} l_6) \equiv TT$

We may also assume that $(l_1 =_{BA} l_2) \equiv TT$, and also $(l_2 =_{BA} l_3) \equiv TT$. Hence $TT \equiv (l_1 =_{BA} l_2) \equiv ((a_1 :: l_4) =_{BA} l_2) \equiv (\partial(l_4) \text{ and } (a_1 \leftarrow l_2)) \text{ and } (l_4 =_{BA} (l_2 - a_1))$. So, by using the '`AndAnalysis`' lemma, we find that $(a_1 \leftarrow l_2) \equiv TT$, and also $(l_4 =_{BA} (l_2 - a_1)) \equiv TT$. Now, $l_1 =_{BA} l_3 \equiv (a_1 :: l_4) =_{BA} l_3 \equiv (\partial(l_4) \text{ and } (a_1 \leftarrow l_3)) \text{ and } (l_4 =_{BA} (l_3 - a_1)) \equiv (a_1 \leftarrow l_3) \text{ and } (l_4 =_{BA} (l_3 - a_1))$. So, by Lemma 4.12, putting " a_1 " for " a ", " l_2 " for " l_1 " and " l_3 " for " l_2 ", we obtain:-

....]- $(l_2 =_{BA} l_3) \equiv TT \supset a_1 \leftarrow l_2 \equiv a_1 \leftarrow l_3$

Hence, $a_1 \leftarrow l_3 \equiv a_1 \leftarrow l_2 \equiv TT$, and so $l_1 =_{BA} l_3 \equiv (a_1 \leftarrow l_3) \text{ and } (l_4 =_{BA} (l_3 - a_1)) \equiv TT \text{ and } (l_4 =_{BA} (l_3 - a_1)) \equiv l_4 =_{BA} (l_3 - a_1)$. Now, by using Lemma 4.15, putting " a_1 " for both " a_1 " and " a_2 ", " l_2 " for " l_1 " and " l_3 " for " l_2 ", this gives :-

$$\dots] - (a_1 =_{At} a_1) \equiv TT \ \& \ (l_2 =_{BA} l_3) \equiv TT \ \supset$$

$$(l_2 - a_1) =_{BA} (l_3 - a_1) \equiv TT$$

The first antecedent can be obtained using 'RefLEqAt', giving

$a_1 =_{At} a_1 = \partial(a_1) \equiv TT$; the second antecedent is already available.

Hence, this gives $(l_2 - a_1) =_{BA} (l_3 - a_1) \equiv TT$. Now, by applying the induction hypothesis with $(l_2 - a_1)$ for " l_5 ", and $(l_3 - a_1)$ for " l_6 ", to produce:-

$$l_4 =_{BA}(l_2 - a_1) \equiv TT \ \& \ (l_2 - a_1) =_{BA}(l_3 - a_1) \equiv TT \ \supset$$

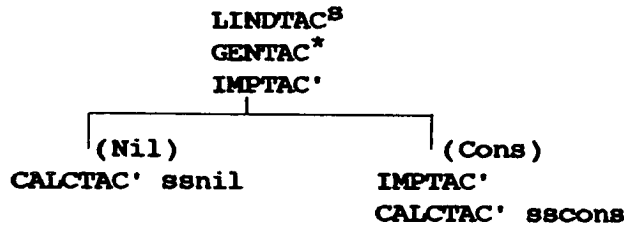
$$l_4 =_{BA}(l_3 - a_1) \equiv TT.$$

Both antecedents are directly available, and so $l_1 =_{BA} l_3 =$

$l_4 =_{BA}(l_3 - a_1) \equiv TT$, which completes the proof of Theorem 4.16.

QED

A tactic for proving this theorem is:-



where

$$\text{ssnil} = \{ \dots] - l_2 \equiv \text{Nil} \}$$

$$\text{sscons} = \{ \dots] - a' \leftarrow l_2 \equiv TT,$$

$$\dots] - l_1' =_{BA} (l_2 - a') \equiv TT,$$

$$\dots] - a' \leftarrow l_2 \equiv a' \leftarrow l_3,$$

$$\dots] - (l_2 - a') =_{BA} (l_3 - a') \equiv TT \ \supset$$

$$l_1' =_{BA} (l_3 - a') \equiv TT$$

}

The initial simpset is enriched with Lemma 4.14. The lemma in simpset ssnil is obtained by specialising the 'NullIMPNil' lemma and then assuming the antecedent. The first two theorems in simpset sscons are obtained by specialising the 'AndAnalysis' lemma and using the conjunction elimination rules. The third theorem is obtained by specialising Lemma 4.12 and assuming the antecedents. Note that this lemma cannot be used directly as a conditional simplrule, since a free variable in it's condition is not matched on the lhs of its consequent.

The final theorem is obtained by assuming the induction hypothesis and then specialising it. The first antecedent is then

eliminated by using Modus Ponens with the second theorem in this simpset. Note that the second antecedent will be eliminated within conditional simplification by applying Lemma 4.14 as a simprule.

Observe that by separating the two applications of IMP_TAC', the Nil case has the second antecedent of the original goal remaining. This contains an occurrence of the specialised variable "l₂" which may then be re-written using the simpset ssnil. The result of that is a classical tautology with identical antecedent and conclusion, so solving that case.

4.6.4 Multisets: Congruence and Commutativity of Append using EqBA.

Lemma 4.17

$\vdash \forall a:\alpha \ l_1 \ l_2:(\alpha)L.$

$$\partial(l_2) \equiv \text{TT} \supset a \leftarrow (l_1 @ l_2) \equiv (a \leftarrow l_1) \text{ or } (a \leftarrow l_2)$$

Proof By definedness case analysis on the value of "a" and then list induction on "l₁".

Suppose that $\partial(a) \equiv \text{I}$. Then $a \equiv \text{I}$ and so $a \leftarrow (l_1 @ l_2) \equiv \text{I} \leftarrow (l_1 @ l_2) \equiv \text{I}$. Also, we have that $(a \leftarrow l_1) \text{ or } (a \leftarrow l_2) \equiv (\text{I} \leftarrow l_1) \text{ or } (\text{I} \leftarrow l_2) \equiv \text{I or I} \equiv \text{I}$, giving the result.

Now, assume that $\partial(a) \equiv \text{TT}$, and continue with list induction on "l₁".

Suppose that $l_1 \equiv \text{I}$. So we have that $a \leftarrow (l_1 @ l_2) \equiv a \leftarrow (\text{I} @ l_2) \equiv a \leftarrow \text{I} \equiv \text{I}$ and also that $(a \leftarrow l_1) \text{ or } (a \leftarrow l_2) \equiv (a \leftarrow \text{I}) \text{ or } (a \leftarrow l_2) \equiv \text{I or } (a \leftarrow l_2) \equiv \text{I}$.

Suppose that $l_1 \equiv \text{Nil}$. Then, $a \leftarrow (l_1 @ l_2) \equiv a \leftarrow (\text{Nil} @ l_2) \equiv a \leftarrow l_2$ and also we have that $(a \leftarrow l_1) \text{ or } (a \leftarrow l_2) \equiv (a \leftarrow \text{Nil}) \text{ or } (a \leftarrow l_2) \equiv (\partial(a) \Rightarrow \text{FF} \mid \text{I}) \text{ or } (a \leftarrow l_2) \equiv \text{FF or } (a \leftarrow l_2) \equiv a \leftarrow l_2$.

Suppose that $l_1 \equiv (a_1 :: l_3)$ where $\partial(a_1) \equiv \partial(l_3) \equiv \text{TT}$ with the induction hypothesis:-

$$\forall l_4 . \partial(l_4) \equiv \text{TT} \supset a \leftarrow (l_3 @ l_4) \equiv (a \leftarrow l_3) \text{ or } (a \leftarrow l_4)$$

Now, $a \leftarrow ((a_1 :: l_3) @ l_2) \equiv a \leftarrow (a_1 :: (l_3 @ l_2)) \equiv (a \text{ =At } a_1) \text{ or } (a \leftarrow (l_3 @ l_2))$. So, applying the induction hypothesis (with "l₂" for "l₄") we note that $a \leftarrow (l_3 @ l_2) \equiv (a \leftarrow l_3) \text{ or } (a \leftarrow l_2)$, since that $\partial(l_2) \equiv \text{TT}$. So, $a \leftarrow ((a_1 :: l_3) @ l_2) \equiv (a \text{ =At } a_1) \text{ or } ((a \leftarrow l_3) \text{ or } (a \leftarrow l_2))$. On the other hand, we have that

$(a \leftarrow (a_1 :: l_3)) \text{ or } (a \leftarrow l_2) \equiv (\partial(l_3) \text{ and } ((a =_{At} a_1) \text{ or } (a \leftarrow l_3)))$
 $\text{or } (a \leftarrow l_2) \equiv ((a =_{At} a_1) \text{ or } (a \leftarrow l_3)) \text{ or } (a \leftarrow l_2) \equiv (a =_{At} a_1) \text{ or}$
 $((a \leftarrow l_3) \text{ or } (a \leftarrow l_2))$, by 'OrAssoc', the associativity of the "or"
operator, and the definedness assumption that $\partial(l_3) \equiv \text{TT}$. So,
finally, we have that $a \leftarrow (l_1 @ l_2) \equiv ((a =_{At} a_1) \text{ or } (a \leftarrow l_3)) \text{ or}$
 $(a \leftarrow l_2) \equiv (a \leftarrow l_1) \text{ or } (a \leftarrow l_2)$. This completes the proof of
Lemma 4.17.

QED

The original proof of this lemma was not developed at the time to sufficient depth by hand to notice the use of "or" associativity. Instead, a proof using case analysis on $(a =_{At} a')$ was assumed to be necessary and accordingly performed, and that proof needed the extra assumption that 'DEFEqAt' held.

A tactic for proving this lemma is:-

```
GENDEFCASESTAC'S
  LINDTAC'S
  GENTAC
  IMPTAC'S
```

Lemma 4.18

..]- $\forall a:\alpha \ l_1 \ l_2:(\alpha)L$.

$(a \leftarrow l_1) \equiv \text{TT} \supset (l_1 @ l_2) - a \equiv (l_1 - a) @ l_2$

(on the assumption that both 'DEFEqAt' and 'SymEqAt' hold).

Proof By definedness case analysis on the value of "a" and then list induction on " l_1 ".

Suppose that $\partial(a) \equiv \text{I}$ and so $a \equiv \text{I}$. But this falsifies the assumption that $a \rightarrow l_1 \equiv \text{TT}$ (since "IsIn" is left strict).

Suppose that $\partial(a) \equiv \text{TT}$ and continue with list induction on l_1 .

Suppose that $l_1 \equiv \text{I}$; as before, this falsifies the assumption that $a \leftarrow l_1 \equiv \text{TT}$, due to the right-strictness of "IsIn".

Suppose that $l_1 \equiv \text{Nil}$. Then, $a \leftarrow l_1 \equiv a \leftarrow \text{Nil} \equiv$
 $\partial(a) \Rightarrow \text{FF} \mid \text{I} \equiv \text{FF}$, falsifying the assumption that $a \leftarrow l_1 \equiv \text{TT}$.

Suppose that $l_1 \equiv (a_1 :: l_3)$ where $\partial(a_1) \equiv \partial(l_3) \equiv \text{TT}$ and the induction hypothesis is:-

$\forall l_2. (a \leftarrow l_3) \equiv \text{TT} \supset (l_1 @ l_2) - a \equiv (l_1 - a) @ l_2$

So, continuing we may now assume that $\text{TT} \equiv a \leftarrow l_1 \equiv a \leftarrow (a_1 :: l_3)$
 $\equiv \partial(l_3) \text{ and } ((a =_{At} a_1) \text{ or } (a \leftarrow l_3)) \equiv (a =_{At} a_1) \text{ or } (a \leftarrow l_3)$.

The result to be shown is that $(l_1 @ l_2) - a \equiv (l_1 - a) @ l_2$

using the above assumptions. So simplifying the lhs we get:-

$((a_1 :: l_3) @ l_2) - a = (a_1 :: (l_3 @ l_2)) - a = ((a_1 =_{At} a) \Rightarrow (l_3 @ l_2) \mid (a_1 :: (l_3 @ l_2)) - a)$, and similarly for the rhs we have that $(l_1 - a) @ l_2 = ((a_1 :: l_3) - a) @ l_2 = ((a_1 =_{At} a) \Rightarrow l_3 \mid a_1 :: (l_3 - a)) @ l_2$. Hence the result which we need to establish becomes:- $((a_1 =_{At} a) \Rightarrow (l_3 @ l_2) \mid a_1 :: (l_3 @ l_2)) - a = ((a_1 =_{At} a) \Rightarrow l_3 \mid a_1 :: (l_3 - a)) @ l_2$.

The next step is to consider a truth value case analysis on the value of $(a_1 =_{At} a)$.

Suppose that $(a_1 =_{At} a) = I$. But then we have that $I = \partial(a_1 =_{At} a) = \partial(a_1)$ and $\partial(a) = TT$ which is a basic contradicition (using the property 'DEFEqAt').

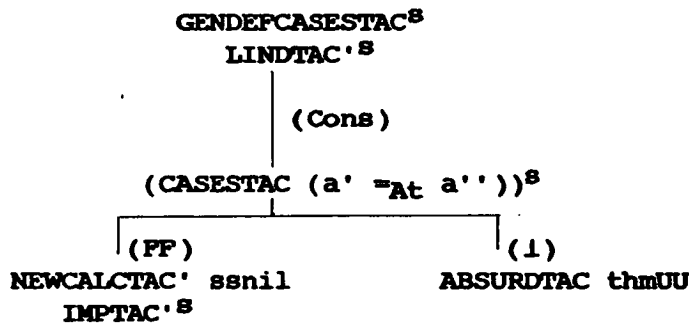
Suppose that $(a_1 =_{At} a) = TT$. Then we have that $(a_1 =_{At} a) \Rightarrow (l_3 @ l_2) \mid a_1 :: ((l_3 @ l_2) - a) = (l_3 @ l_2) = ((a_1 =_{At} a) \Rightarrow l_3 \mid a_1 :: (l_3 - a)) @ l_2$ which is what was wanted.

Suppose now that $(a_1 =_{At} a) = FF$. Then $(a_1 =_{At} a) \Rightarrow (l_3 @ l_2) \mid a_1 :: ((l_3 @ l_2) - a) = a_1 :: ((l_3 @ l_2) - a)$ and also $((a_1 =_{At} a) \Rightarrow l_3 \mid a_1 :: (l_3 - a)) @ l_2 = (a_1 :: (l_3 - a)) @ l_2 = a_1 :: ((l_3 - a) @ l_2)$. Now, by applying the property 'SymEqAt', we have that $(a =_{At} a_1) = (a_1 =_{At} a) = FF$ and also we get $TT = (a =_{At} a_1)$ or $(a \leftarrow l_3) = FF$ or $(a \leftarrow l_3) = (a \leftarrow l_3)$. This result is precisely the enabling condition for the induction hypothesis to be applied. Hence, putting " l_3 " for " l_4 ", we have that $(l_3 @ l_2) - a = a_1 :: (l_3 - a) @ l_2$, which completes this case.

In each of the above case, the required equation was established and so this proof is completed.

QED

A tactic for this lemma is:-



where

thmUU = .]- $\partial(a' \text{ =At } a) \equiv \partial(a')$ and $\partial(a)$

ssnil = { ..]- $a \text{ =At } a' \equiv \text{FF}$ }

The theorem thmUU is standard, and the theorem for ssnil is obtained by specialising 'DEFEqAt' and simplifying with the assumption that $a' \text{ =At } a \equiv \text{FF}$.

Note that, in the FF case, the simpset ssnil is used before 'IMPTAC' is applied. This is to ensure that the antecedent has the full benefit of simplification applied to it before it is incorporated as an assumption and (local) simprule.

Lemma 4.19

..]- $\forall a:\alpha \ l_1 \ l_2:(\alpha)L.$

$(a \leftarrow l_1) \equiv \text{FF} \supset ((l_1 @ l_2) - a \equiv l_1 @ (l_2 - a))$

(on the assumption that both 'DEFEqAt' and 'SymEqAt' hold.)

Proof By list induction on l_1 .

Suppose that $l_1 = 1$. This falsifies the antecedent since $\text{FF} \equiv (a \leftarrow l_1) \equiv a \leftarrow 1 \equiv 1$.

Suppose that $l_1 = \text{Nil}$. Now, this easily gives $((l_1 @ l_2) - a) \equiv ((\text{Nil} @ l_2) - a) \equiv l_2 - a \equiv \text{Nil} @ (l_2 - a) \equiv l_1 @ (l_2 - a)$.

Suppose that $l_1 = (a_1 :: l_3)$ where $\partial(a_1) \equiv \partial(l_1) \equiv \text{TT}$, and with the induction hypothesis:-

$\forall l_4. (a \leftarrow l_3) \equiv \text{FF} \supset (l_3 @ l_4) - a \equiv l_3 @ (l_4 - a)$

Now, we may assume that $\text{FF} \equiv (a \leftarrow l_3) \equiv \text{TT}$ and $((a \text{ =At } a_1) \text{ or } (a \leftarrow l_3)) \equiv (a \text{ =At } a_1) \text{ or } (a \leftarrow l_3)$. Using the 'OrAnalysis' lemma,

we have that $(a \text{ =At } a_1) \equiv \text{FF}$, as well as $(a \leftarrow l_3) \equiv \text{FF}$. Also, from the assumption 'SymEqAt', $(a_1 \text{ =At } a) \equiv (a \text{ =At } a_1) \equiv \text{FF}$. Now, consider $((l_1 @ l_2) - a) \equiv (((a_1 :: l_3) @ l_2) - a) \equiv ((a_1 :: (l_3 @ l_2)) - a) \equiv ((a_1 \text{ =At } a) \rightarrow (l_3 @ l_2)) \mid$

$a_1 :: (l_3 @ l_2)) - a \equiv a_1 :: ((l_3 @ l_2) - a)$. Now, apply the induction hypothesis with " l_2 " for " l_4 ", and using the sub-result that $a \leftarrow l_2 \equiv \text{FF}$, we have that $(l_3 @ l_2) - a \equiv l_3 @ (l_2 - a)$. So, substituting, we get that $a_1 :: ((l_3 @ l_2) - a) \equiv$

$a_1 :: (l_3 @ (l_2 - a)) \equiv (a_1 :: l_3) @ (l_2 - a) \equiv l_1 @ (l_2 - a)$, and this concludes the proof of lemma 4.19.

QED

A tactic which proves this lemma is:-

```

      GENTAC
      LINDTAC'S
      |
      (Cons)
      |
      GENTAC
      IMPTAC
      CALCTAC' sscons

```

where

```

sscons = { .]- a =At a' = FF,
           .]- a < l1' = FF,
           ..]- a' =At a = FF
         }

```

The first two of these rules are obtained by specialising the 'OrAnalysis' lemma, then assuming the antecedents, and using the conjunction elimination rules. The third simprule is obtained by specialising 'SymEqAt' and then simplifying with the first simprule described above.

Lemma 4.20 (Congruence theorem)

```

...]- ∀ l1 l1' l2 l2' : (α)L.
      (l1 =BA l1') = TT & (l2 =BA l2') = TT ⊃
      (l1 @ l2) =BA (l1' @ l2') = TT

```

(on the assumption that 'DEFEqAt' and 'SymEqAt' hold).

Proof By list induction on l₁.

Suppose that l₁ = I. This falsifies the assumption that (l₁ =BA l₁') = TT holds, because of the lemma 'DEFEqBA' (which depends upon the assumption 'DEFEqAt').

Suppose that l₁ = Nil; then the first antecedent becomes (Nil =BA l₁') = TT. So, by computation, we get that Null l₁' = TT, which by applying the list property 'NullIMPNil', we have that l₁' = Nil also. So, (l₁ @ l₂) =BA (l₁' @ l₂') = (Nil @ l₂) =BA (Nil @ l₂') = (l₂ @ l₂') = TT, by the second assumption.

Suppose that l₁ = (a₁ :: l₃) where ∂(a₁) = ∂(l₃) = TT and with the induction hypothesis:-

$$\forall l_3' l_2 l_2'.$$

$$(l_3 =_{BA} l_3') \equiv TT \quad \& \quad (l_2 =_{BA} l_2') \equiv TT \quad \supset$$

$$(l_3 \in l_2) =_{BA} (l_3' \in l_2') \equiv TT$$

Continuing, we are now free to assume $(a_1 :: l_3) =_{BA} l_1' \equiv TT$ as well as $(l_2 =_{BA} l_2') \equiv TT$. Now, by evaluation, we have that $TT \equiv (a_1 :: l_3) =_{BA} l_1' \equiv (a_1 \leftarrow l_1')$ and $(l_3 =_{BA} (l_1' - a_1))$, since l_3 is already taken as defined. So, by the propositional 'AndAnalysis' lemma, we have that $(a_1 \leftarrow l_1') \equiv TT$ and that $(l_3 =_{BA} (l_1' - a_1)) \equiv TT$. Now, consider:-

$$\begin{aligned} (l_1 \in l_2) &=_{BA} (l_1' \in l_2') \\ &\equiv ((a_1 :: l_3) \in l_2) =_{BA} (l_1' \in l_2') \\ &\equiv (a_1 :: (l_3 \in l_2)) =_{BA} (l_1' \in l_2') \\ &\equiv (\partial(l_3 \in l_2) \text{ and } (a_1 \leftarrow (l_1' \in l_2'))) \text{ and } \\ &\quad ((l_3 \in l_2) =_{BA} ((l_1' \in l_2') - a_1)). \end{aligned}$$

Now, $\partial(l_3 \in l_2) \equiv \partial(l_3)$ and $\partial(l_2) \equiv TT$, since l_3 is defined by assumption and l_2 is defined because $(l_2 =_{BA} l_2')$ is defined also (using the 'DEFEqBA' lemma). Also, from lemma 4.17, that, since $\partial(l_2') \equiv TT$ (from definedness of $(l_2 =_{BA} l_2')$) we have $a_1 \leftarrow (l_1' \in l_2') \equiv (a_1 \leftarrow l_1')$ or $(a_1 \leftarrow l_2') \equiv TT$ or $(a_1 \leftarrow l_2') \equiv TT$. Finally, using lemma 4.18 (which depends upon both 'DEFEqAt' and 'SymEqAt'), we have that $((l_1' \in l_2') - a_1) \equiv (l_1' - a_1) \in l_2'$ since $a_1 \leftarrow l_1' \equiv TT$.

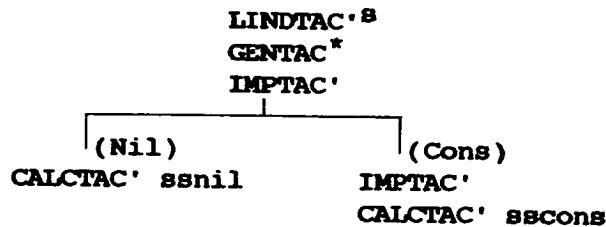
Returning to the above equation, we have that

$$\begin{aligned} (l_1 \in l_2) &=_{BA} (l_1' \in l_2') \\ &\equiv TT \text{ and } TT \text{ and } ((l_3 \in l_2) =_{BA} ((l_1' - a) \in l_2')) \\ &\equiv (l_3 \in l_2) =_{BA} ((l_1' - a) \in l_2') \end{aligned}$$

Now, since we have that $l_3 =_{BA} (l_1' - a) \equiv TT$ as well as $l_2 =_{BA} l_2' \equiv TT$ we may apply the induction hypothesis (with $(l_1' - a)$ for l_3') to finally show that $(l_1 \in l_2) =_{BA} (l_1' \in l_2') \equiv TT$. This completes the proof of Lemma 4.20.

QED

A tactic which proves this theorem is:-



where:-

$$\text{ssnil} = \{ . \} - l_1' = \text{Nil} \}$$

$$\text{sscons} = \{ . \} - a' \leftarrow l_1' = \text{TT}, . \} - l_1'' =_{\text{BA}} (l_1 - a') = \text{TT} \}$$

The initial simpset also includes Lemmas 4.17 and 4.18. The simprule contained in ssnil is obtained by specialising lemma 'NullIMPNIL' and assuming the antecedent. The two simprules forming the simpset sscons are obtained by specialising the 'AndAnalysis' lemma assuming antecedents and using conjunction elimination rules. Note that, as in the tactical proof of Lemma 4.16, by only discharging one of the antecedents, the goal is reducible to a trivial goal by simplifying the antecedent so that it is identical to the consequent.

Lemma 4.21

$$\dots \} - \forall l: (\alpha) L \quad a_1 \ a_2: \alpha.$$

$$(a_1 =_{\text{At}} a_2) = \text{TT} \ \& \ (a_1 \leftarrow l) = \text{TT} \ \supset$$

$$l =_{\text{BA}} ((l - a_1) @ \text{Unit}(a_2)) = \text{TT}$$

(on the assumption that "EqAt" is an equivalence predicate).

Proof

The first step is to do a list induction on "l", and suppose that $l = l$. This contradicts the second antecedent, since $\text{TT} = a_1 \rightarrow l = a_1 \leftarrow l = l$.

Suppose that $l = \text{Nil}$, and assume that $a_1 =_{\text{At}} a_2 = \text{TT}$, and that $a_1 \leftarrow l = \text{TT}$. Now, using 'DEFEqAt' we have that $\text{TT} = \partial(a_1 =_{\text{At}} a_2) = \partial(a_1)$ and $\partial(a_2)$. So, using the 'And Analysis' lemma we get that both $\partial(a_1) = \text{TT}$, and also that $\partial(a_2) = \text{TT}$. However, a contradiction arises, since $\text{TT} = a_1 \leftarrow l = a_1 \leftarrow \text{Nil} = \partial(a_1) \Rightarrow \text{FF} \mid l = \text{FF}$.

Suppose that $l = a_3 :: l_1$ where $\partial(a_3) = \partial(l_1) = \text{TT}$ with the induction hypothesis that

$$\forall a_4 \ a_5: \alpha. (a_4 =_{\text{At}} a_5) = \text{TT} \ \& \ (a_4 \leftarrow l_1) = \text{TT} \ \supset$$

$$l_1 =_{\text{BA}} ((l_1 - a_4) @ \text{Unit}(a_5)) = \text{TT}$$

We also assume that $a_1 \text{ =At } a_2 \equiv \text{TT}$ and that $a_1 \leftarrow 1 \equiv \text{TT}$. As in the previous case, using both 'DEFEqAt' and 'And Analysis' to give $\partial(a_1) \equiv \text{TT}$, and also $\partial(a_2) \equiv \text{TT}$. Now $\text{TT} \equiv a_1 \leftarrow 1 \equiv a_1 \leftarrow (a_3 :: l_1) \equiv \partial(l_1)$ and $((a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv \text{TT}$ and $((a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1)) \equiv (a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1)$.

Proceeding with a truth value case analysis on the value of $(a_3 \text{ =At } a_1)$, suppose that $(a_3 \text{ =At } a_1) \equiv 1$. This is a contradiction, because $1 \equiv \partial(a_3 \text{ =At } a_1) \equiv \partial(a_3)$ and $\partial(a_1) \equiv \text{TT}$.

Put $q \equiv (1 - a_1) \in \text{Unit}(a_2)$, and also $q_1 \equiv (l_1 - a_1) \in \text{Unit}(a_2)$.

Suppose that $(a_3 \text{ =At } a_1) \equiv \text{FF}$. Then, using 'SymEqAt' we have that $(a_1 \text{ =At } a_3) \equiv (a_3 \text{ =At } a_1) \equiv \text{FF}$. Also, $\text{TT} \equiv a_1 \leftarrow 1 \equiv (a_1 \text{ =At } a_3) \text{ or } (a_1 \leftarrow l_1) \equiv \text{FF or } (a_1 \leftarrow l_1) \equiv a_1 \leftarrow l_1$. Now, $1 - a_1 \equiv (a_3 :: l_1) - a_1 \equiv (a_3 \text{ =At } a_1) \Rightarrow l_1 \mid a_3 :: (l_1 - a_1) \equiv a_3 :: (l_1 - a_1)$. So, $q \equiv (1 - a_1) \in \text{Unit}(a_2) \equiv (a_3 :: (l_1 - a_1)) \in \text{Unit}(a_2) \equiv a_3 :: ((l_1 - a_1) \in \text{Unit}(a_2)) \equiv a_3 :: q_1$. Also, we have that $a_3 \leftarrow q \equiv a_3 \leftarrow (a_3 :: q_1) \equiv \partial(q_1)$ and $((a_3 \text{ =At } a_3) \text{ or } (a_3 \leftarrow a_1)) \equiv \text{TT}$ and $(\partial(a_3) \text{ or } (a_3 \leftarrow q_1)) \equiv \text{TT or } (a_3 \leftarrow q_1) \equiv \text{TT}$, since $\partial(q_1) \equiv \partial(l_1 - a_1) \in \text{Unit}(a_2) \equiv \partial(l_1 - a_1)$ and $\partial(\text{Unit } a_2) \equiv (\partial(l_1) \text{ and } \partial(a_1)) \text{ and } \partial(a_2) \equiv \text{TT}$. Moreover, $q - a_3 \equiv (a_3 :: q_1) - a_3 \equiv ((a_3 \text{ =At } a_3) \Rightarrow q_1 \mid a_3 :: (q_1 - a_3)) \equiv q_1$. Now, $1 \text{ =BA } ((1 - a_1) \in \text{Unit}(a_2)) \equiv (1 \text{ =BA } q) \equiv (a_3 :: l_1) \text{ =BA } q \equiv (\partial(l_1) \text{ and } (a_3 \leftarrow q)) \text{ and } (l_1 \text{ =BA } (q - a_3)) \equiv (\text{TT and TT}) \text{ and } (l_1 \text{ =BA } q_1) \equiv (l_1 \text{ =BA } q_1)$. So, by instantiating the induction hypothesis, with " a_1 " for " a_4 ", " a_2 " for " a_5 " we get that:-

$$(a_1 \text{ =At } a_2) \equiv \text{TT} \quad \& \quad (a_1 \leftarrow l_1) \equiv \text{TT} \quad >$$

$$l_1 \text{ =BA } ((l_1 - a_1) \in \text{Unit}(a_2)) \equiv \text{TT}$$

Both antecedents are already available, and so we have that

$$l_1 \text{ =BA } ((l_1 - a_1) \in \text{Unit}(a_2)) \equiv \text{TT}.$$

Hence, in this case, $1 \text{ =BA } ((1 - a_1) \in \text{Unit}(a_2)) \equiv l_1 \text{ =BA } q_1 \equiv l_1 \text{ =BA } ((l_1 - a_1) \in \text{Unit}(a_2)) \equiv \text{TT}$.

Suppose that $(a_2 \text{ =At } a_1) \equiv \text{TT}$. So by using 'TransEqAt' with " a_3 " for " a_1 ", " a_1 " for " a_2 " and " a_2 " for " a_3 " we get"-

$$\dots] - (a_3 \text{ =At } a_1) \equiv \text{TT} \quad \& \quad (a_1 \text{ =At } a_2) \equiv \text{TT} \quad >$$

$$(a_2 \text{ =At } a_2) \equiv \text{TT}$$

Both antecedents are already available, and hence, we have that

$(a_3 =_{At} a_2) \equiv TT$. Also, by 'SymEqAt', we have that $(a_2 =_{At} a_3) \equiv (a_3 =_{At} a_2) \equiv TT$.

Now, $1 - a_1 \equiv (a_3 :: l_1) - a_1 \equiv ((a_3 =_{At} a_1) \Rightarrow l_1 \mid a_3 :: (l_1 - a_1)) \equiv l_1$. So, $q \equiv (1 - a_1) @ Unit(a_2) \equiv l_1 @ Unit(a_2)$. Hence, $1 =_{BA} ((1 - a_1) @ Unit(a_2)) \equiv (a_3 :: l_1) =_{BA} q \equiv (\partial(l_1) \text{ and } (a_3 \leftarrow q)) \text{ and } (l_1 =_{BA}(q - a_3)) \equiv (TT \text{ and } (a_3 \leftarrow q)) \text{ and } (l_1 =_{BA}(q - a_3)) \equiv (a_3 \leftarrow q) \text{ and } (l_1 =_{BA}(q - a_3))$. Now, using Lemma 4.17, $(a_3 \leftarrow q) \equiv a_3 \leftarrow (l_1 @ Unit(a_2)) \equiv (a_3 \leftarrow l_1) \text{ or } (a_3 \leftarrow Unit(a_2))$, since $\partial(Unit(a_2)) \equiv \partial(a_2) \equiv TT$. Also, $q - a_3 \equiv (l_1 @ Unit(a_2)) - a_3$.

We proceed by conducting a truth value case analysis on the value of $a_3 \leftarrow l_1$, and so suppose that $a_3 \leftarrow l_1 \equiv 1$. This gives a contradiction since $1 \equiv \partial(a_3 \leftarrow l_1) \equiv \partial(a_3) \text{ and } \partial(l_1) \equiv TT$.

Suppose that $a_3 \leftarrow l_1 \equiv TT$. Then $a_3 \leftarrow a \equiv (a_3 \leftarrow l_1) \text{ or } (a_3 \leftarrow Unit(a_2)) \equiv TT \text{ or } (a_3 \leftarrow Unit(a_2)) \equiv TT$. Also, by Lemma 4.18, with "a₃" for "a" and "Unit(a₂)" for "l₂" we have that:-

...]- $a_3 \leftarrow l_1 \equiv TT \supset$

$(l_1 @ Unit(a_2)) - a_3 \equiv (l_1 - a_3) @ Unit(a_2)$

Hence, we obtain that $q - a_3 \equiv (l_1 @ Unit(a_2)) - a_3 \equiv$

$(l_1 - a_3) @ Unit(a_2)$. So, $l_1 =_{BA} (q - a_3) \equiv$

$l_1 =_{BA} ((l_1 - a_3) @ Unit(a_2))$. Now, using the induction hypothesis with "a₃" for "a₄" and "a₂" for "a₅", to give:-

$(a_3 =_{At} a_2) \equiv TT \ \& \ (a_3 \leftarrow l_1) \equiv TT \supset$

$l_1 =_{BA} ((l_1 - a_3) @ Unit(a_2)) \equiv TT$

Both antecedents are already available and so we get that

$l_1 =_{BA} ((l_1 - a_3) @ Unit(a_2)) \equiv TT$. Hence, $l_1 =_{BA} (q - a_3) \equiv TT \equiv a_3 \leftarrow q$.

Suppose that $a_3 \leftarrow l_1 \equiv FF$. Then $a_3 \leftarrow q \equiv (a_3 \leftarrow l_1) \text{ or } (a_3 \leftarrow Unit(a_2)) \equiv FF \text{ or } (a_3 \leftarrow Unit(a_2)) \equiv (a_3 \leftarrow Unit(a_2)) \equiv (a_3 \leftarrow (a_2 :: Nil)) \equiv \partial(Nil) \text{ and } ((a_3 =_{At} a_2) \text{ or } (a_3 \leftarrow Nil)) \equiv TT \text{ and } (TT \text{ or } (a_3 \leftarrow Nil)) \equiv TT$. Now, by Lemma 4.19 with "a₃" for "a" and "Unit(a₂)" for "l₂" we get:-

...]- $a_3 \leftarrow l_1 \equiv FF \supset$

$(l_1 @ Unit(a_2)) - a_3 \equiv l_1 @ (Unit(a_2) - a_3)$

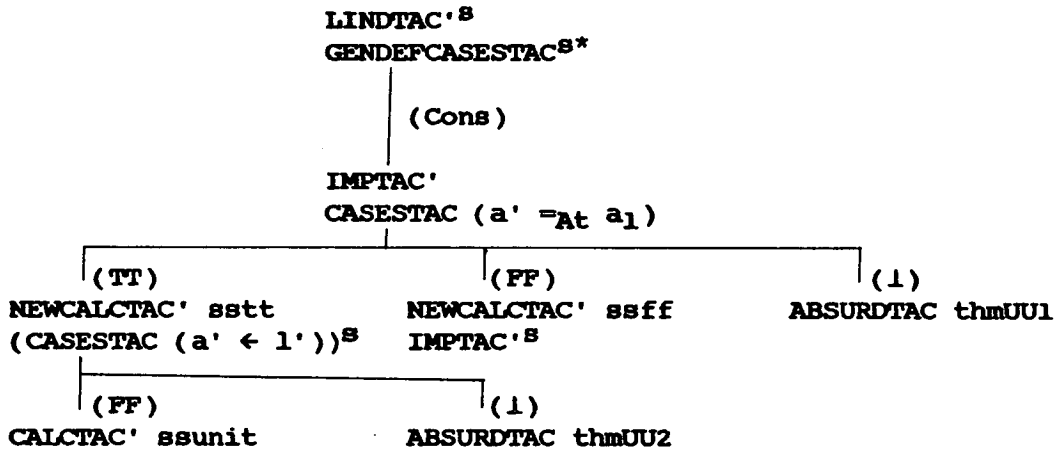
Hence, we have that $q - a_3 \equiv (l_1 @ Unit(a_2)) - a_3 \equiv$

$l_1 @ (\text{Unit}(a_2) - a_3)$. Now, $\text{Unit}(a_2) - a_3 = (a_2 :: \text{Nil}) - a_3 = (a_2 =_{\text{At}} a_3) \neq \text{Nil} \mid a_2 :: (\text{Nil} - a_3) = \text{Nil}$. So $q - a_3 = l_1 @ (\text{Unit}(a_2) - a_3) = l_1 @ \text{Nil} = l_1$. Now, using Theorem 4.6, we have that $l_1 =_{\text{BA}} (q - a_3) = (l_1 =_{\text{BA}} l_1) = \partial(l_1) = \text{TT} = a_3 \leftarrow q$.

So, finally, we have in both of the previous cases that $l =_{\text{BA}} ((l - a_1) @ \text{Unit}(a_2)) = (a_3 \leftarrow q) \text{ and } (l_1 =_{\text{BA}} (q - a_3)) = \text{TT and TT} = \text{TT}$. This completes the proof of Lemma 4.21.

QED

A tactic which proves this lemma is:-



where

$\text{thmUU1} = \dots - \partial(a' =_{\text{At}} a_1) = \partial(a') \text{ and } \partial(a_1)$
 $\text{thmUU2} = \dots - \partial(a' < 1') = \partial(a') \text{ and } \partial(1')$
 $\text{ssff} = \{ \dots - a_1 =_{\text{At}} a' = \text{FF} \}$
 $\text{sstt} = \{ \dots - a_3 =_{\text{At}} a_2 = \text{TT}, \dots - a_2 =_{\text{At}} a_3 = \text{TT} \}$
 $\text{ssunit} = \{] - \forall a. \text{Unit}(a) = a :: \text{Nil} \}$

The initial simpset also includes Theorem 4.6, Lemmas 4.19, 4.18, 4.17 and the 'DEFUnit' lemma as simprules. The theorems thmUU1 and thmUU2 are obtained by specialising 'DEFEqAt' and 'DEFIsIn' appropriately. The simprule in ssff is obtained by specialising 'SymEqAt' and simplifying using the assumption $a' =_{\text{At}} a_1 = \text{FF}$ as a simprule. The first simprule in sstt is obtained by specialising 'TransEqAt' and then assuming both the antecedents; the second simprule is obtained by specialising 'SymEqAt' and then simplifying with the previous simprule. Finally, the simpset ssUnit consists of the definition of Unit.

There are two notable points of interest here. First of all, the definition of "Unit" is not included in the initial simpset, for the same reasons as in Lemma 3.18. Note that ssUnit is only applied in a case where the induction hypothesis is no longer required; in every other case, use of the 'DEFUnit' lemma suffices.

Secondly, note that the second antecedent of the goal is not discharged until it has had an opportunity to be simplified using ssff. This ensures that the antecedent is reduced to $a_1 \leftarrow 1' = \text{TT}$ which may then be used as a simprule.

The following is the main theorem of this study.

Theorem 4.22 (Commutativity of @, with respect to EqBA)

....]- $\forall l_1 l_2 : (\alpha)L. (l_1 @ l_2) =_{\text{BA}} (l_2 @ l_1) = \partial(l_1) \text{ and } \partial(l_2)$
(on the assumption that "EqAt" is an equivalence predicate.)

Proof We begin with a list induction on l_1 .

Suppose that $l_1 = 1$. Then $(l_1 @ l_2) =_{\text{BA}} (l_2 @ l_1) = (1 @ l_2) =_{\text{BA}} (l_2 @ 1) = 1 =_{\text{BA}} 1 = 1$ and also $\partial(l_1) \text{ and } \partial(l_2) = \partial(1) \text{ and } \partial(l_2) = 1 \text{ and } \partial(l_2) = 1$.

Suppose that $l_1 = \text{Nil}$. Then $(l_1 @ l_2) =_{\text{BA}} (l_2 @ l_1) = (\text{Nil} @ l_2) =_{\text{BA}} (l_2 @ \text{Nil}) = l_2 =_{\text{BA}} l_2$. Now, using Theorem 4.6, we have that $l_2 =_{\text{BA}} l_2 = \partial(l_2)$. But also we have $\partial(l_1) \text{ and } \partial(l_2) = \partial(\text{Nil}) \text{ and } \partial(l_2) = (\text{TT} \text{ and } \partial(l_2)) = \partial(l_2)$.

Suppose that $l_1 = a_1 :: l_3$ where $\partial(a_1) = \partial(l_3) = \text{TT}$ and with the induction hypothesis that :-

$\forall l_4. (l_3 @ l_4) =_{\text{BA}} (l_4 @ l_3) = \partial(l_3) \text{ and } \partial(l_4)$

We continue with a definedness case analysis on the value of " l_2 ", and suppose that $\partial(l_2) = 1$, and hence $l_2 = 1$. Then

$(l_1 @ l_2) =_{\text{BA}} (l_2 @ l_1) = (l_1 @ 1) =_{\text{BA}} (1 @ l_1) = (1 =_{\text{BA}} 1) = 1$, and also $\partial(l_1) \text{ and } \partial(l_2) = \text{TT} \text{ and } 1 = 1$.

Suppose that $\partial(l_2) = \text{TT}$, and continue by considering

$$\begin{aligned}
 (l_1 @ l_2) &=_{\text{BA}} (l_2 @ l_1) \\
 &= ((a_1 :: l_3) @ l_2) =_{\text{BA}} (l_2 @ l_1) \\
 &= (a_1 :: (l_3 @ l_2)) =_{\text{BA}} (l_2 @ l_1) \\
 &= (\partial(l_3 @ l_2) \text{ and } (a_1 \leftarrow (l_2 @ l_1))) \text{ and } \\
 &\quad ((l_3 @ l_2) =_{\text{BA}} ((l_2 @ l_1) - a_1)) \\
 &= (\partial(l_3) \text{ and } \partial(l_2)) \text{ and } (a_1 \leftarrow (l_2 @ l_1)) \text{ and } \\
 &\quad ((l_3 @ l_2) =_{\text{BA}} ((l_2 @ l_1) - a_1))
 \end{aligned}$$

$$\begin{aligned}
&= (\text{TT} \text{ and } (a_1 \leftarrow (l_2 @ l_1))) \text{ and} \\
&\quad ((l_3 @ l_2) \text{BA} ((l_2 @ l_1) - a_1)) \\
&= (a_1 \leftarrow (l_2 @ l_1)) \text{ and} \\
&\quad ((l_3 @ l_2) \text{BA} ((l_2 @ l_1) - a_1)).
\end{aligned}$$

By using Lemma 4.17, we have that $a_1 \leftarrow (l_2 @ l_1) = (a_1 \leftarrow l_2) \text{ or } (a_1 \leftarrow l_1)$, since $\partial(l_1) = \partial(a_1 :: l_3) = \partial(a_1) \text{ and } \partial(l_3) = \text{TT}$. So, $(l_1 @ l_2) \text{BA} (l_2 @ l_1) = ((a_1 \rightarrow l_2) \text{ or } (a_1 \leftarrow l_1)) \text{ and } ((l_3 @ l_2) \text{BA} (l_2 @ l_1) - a_1)$. Now, $a_1 \leftarrow l_1 = a_1 \leftarrow (a_1 :: l_3) = \partial(l_3) \text{ and } ((a_1 \text{BA} a_1) \text{ or } (a_1 \leftarrow l_3)) = \text{TT} \text{ and } (\partial(a_1) \text{ or } (a_1 \leftarrow l_3)) = \text{TT} \text{ or } (a_1 \leftarrow l_3) = \text{TT}$. Hence, $(l_1 @ l_2) \text{BA} (l_2 @ l_1) = ((a_1 \leftarrow l_2) \text{ or } \text{TT}) \text{ and } ((l_3 @ l_2) \text{BA} ((l_2 @ l_1) - a_1))$.

Continuing by a truth value case analysis on the value of $(a_1 \leftarrow l_2)$, suppose that $a_1 \leftarrow l_2 = \text{I}$. This gives a contradiction since $\text{I} = \partial(a_1 \leftarrow l_2) = \partial(a_1) \text{ and } \partial(l_2) = \text{TT}$.

Suppose that $a_1 \leftarrow l_2 = \text{FF}$. Then, by Lemma 4.19, with "a₁" for "a", "l₂" for "l₁" and "l₁" for "l₂" we have that:-

$$\dots] - a_1 \leftarrow l_2 = \text{FF} \supset (l_2 @ l_1) - a_1 = l_2 @ (l_1 - a_1)$$

Since the antecedent is already available we have that

$(l_2 @ l_1) - a_1 = l_2 @ (l_1 - a_1)$. However $l_1 - a_1 = (a_1 :: l_3) - a_1 = (a_1 \text{BA} a_1) \rightarrow l_3 \mid a_1 :: (l_2 - a_2) = l_3$. So, $(l_2 @ l_1) - a_1 = l_2 @ l_3$. Therefore, using the induction hypothesis with "l₂" for "l₄", gives that:-

$$(l_3 @ l_2) \text{BA} (l_2 @ l_3) = \partial(l_3) \text{ and } \partial(l_2)$$

Hence, $(l_1 @ l_2) \text{BA} (l_2 @ l_1)$

$$\begin{aligned}
&= ((a \leftarrow l_2) \text{ or } \text{TT}) \text{ and } (l_3 @ l_2) \text{BA} ((l_2 @ l_1) - a_1) \\
&= (\text{FF} \text{ or } \text{TT}) \text{ and } ((l_3 @ l_2) \text{BA} (l_2 @ l_3)) \\
&= \text{TT} \text{ and } (\partial(l_3) \text{ and } \partial(l_2)) = \text{TT}.
\end{aligned}$$

Suppose that $a_1 \leftarrow l_2 = \text{TT}$. Then, by lemma 4.18, with "a₁" for "a", "l₂" for "l₁" and "l₁" for "l₂" to give:-

$$\dots] - a_1 \leftarrow l_2 = \text{TT} \supset (l_2 @ l_1) - a_1 = (l_2 - a_1) @ l_1$$

Since the antecedent is available here, we get that

$(l_2 @ l_1) - a_1 = (l_2 - a_1) @ l_1 = (l_2 - a_1) @ (a_1 :: l_3) = (l_2 - a_1) @ (\text{Unit}(a_1) @ l_3) = ((l_2 - a_1) @ \text{Unit}(a_1)) @ l_3$, using the UnitCons Lemma. Now, by Lemma 4.21 with "l₂" for "l" and "a₁" for "a₂", we get:-

$$\begin{aligned} \dots] - (a_1 \text{ =At } a_1) &\equiv \text{TT} \quad \& \quad (a_1 \leftarrow l_2) \equiv \text{TT} \quad \supset \\ l_2 &\text{ =BA } ((l_2 - a_1) \in \text{Unit}(a_1)) \equiv \text{TT} \end{aligned}$$

The first antecedent can be obtained with 'REFLEqAt' by

$(a_1 \text{ =At } a_1) \equiv \partial(a_1) \equiv \text{TT}$; the second antecedent is already available. Hence we obtain that $l_2 \text{ =BA } ((l_2 - a_1) \in \text{Unit}(a_1)) \equiv \text{TT}$. By Lemma 4.20 with " l_2 " for " l_1 ", $((l_2 - a_1) \in \text{Unit}(a_1))$ for " l_2 " and " l_3 " for " l_4 " to give:-

$$\begin{aligned} \dots] - l_2 &\text{ =BA } ((l_2 - a_1) \in \text{Unit}(a_1)) \equiv \text{TT} \quad \& \quad l_3 \text{ =BA } l_3 \equiv \text{TT} \quad \supset \\ (l_2 \in l_3) &\text{ =BA } (((l_2 - a_1) \in \text{Unit}(a_1)) \in l_3) \equiv \text{TT} \end{aligned}$$

The first antecedent is already proven; the second antecedent can be obtained by using Theorem 4.6 since $l_3 \text{ =BA } l_3 \equiv \partial(l_3) \equiv \text{TT}$. Hence, we obtain that $(l_2 \in l_3) \text{ =BA } (((l_2 - a_1) \in \text{Unit}(a_1)) \in l_3) \equiv \text{TT}$. So using the induction hypothesis with " l_2 " for " l_4 " to give

$(l_3 \in l_2) \text{ =BA } (l_2 \in l_3) \equiv \partial(l_3)$ and $\partial(l_2) \equiv \text{TT}$. So, by Theorem 4.16 with $(l_3 \in l_2)$ for " l_1 ", $(l_2 \in l_3)$ for " l_2 " and $((l_2 - a_1) \in \text{Unit}(a_1)) \in l_3$ for " l_3 " to give:-

$$\begin{aligned} \dots] - (l_3 \in l_2) &\text{ =BA } (l_2 \in l_3) \equiv \text{TT} \quad \& \\ (l_2 \in l_3) &\text{ =BA } (((l_2 - a_1) \in \text{Unit}(a_1)) \in l_3) \equiv \text{TT} \quad \supset \\ (l_3 \in l_2) &\text{ =BA } (((l_2 - a_1) \in \text{Unit}(a_1)) \in l_3) \equiv \text{TT} \end{aligned}$$

Both of the antecedents are already available, and so we have that

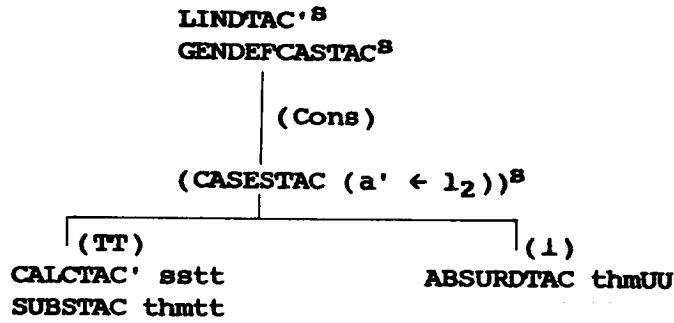
$$\begin{aligned} (l_3 \in l_2) &\text{ =BA } ((l_2 \in l_1) - a_1) \\ &\equiv (l_3 \in l_2) \text{ =BA } (((l_2 - a_1) \in \text{Unit}(a_1)) \in l_3) \\ &\equiv \text{TT}. \end{aligned}$$

Hence, $(l_1 \in l_2) \text{ =BA } (l_2 \in l_1) \equiv (a_1 \leftarrow l_2 \text{ or } \text{TT})$ and $\text{TT} \equiv \text{TT}$.

In each of the previous cases, $(l_1 \in l_2) \text{ =BA } (l_2 \in l_1) \equiv \text{TT} \equiv \partial(l_1)$ and $\partial(l_2)$, since $\partial(l_1)$ and $\partial(l_2) \equiv \partial(a_1 \text{ : } l_3)$ and $\partial(l_2) \equiv (\partial(a_1) \text{ and } \partial(l_3))$ and $\partial(l_2) \equiv \text{TT}$. This completes the proof of Theorem 4.22.

QED

A tactic which proves this theorem is:-



where:-

thmUU = $\lambda a' \lambda l_2. \partial(a' \leftarrow l_2) \equiv \partial(a') \text{ and } \partial(l_2)$

thmtt = $\lambda l_1' \lambda l_2. \lambda a. \lambda l. (a :: l) \equiv \text{Unit}(a) @ l$

$(l_1' @ l_2) \equiv_{\text{BA}} ((l_2 - a') @ (\text{Unit}(a') @ l_1')) \equiv \text{TT}$

sstt = $\lambda a \lambda l. (a :: l) \equiv \text{Unit}(a) @ l$

The initial simpset additionally contains the Lemmas 'AppUU', 'AppNil', 'AssocApp', Theorem 4.6, and Lemmas 4.17, 4.18, and 4.19. The lemma thmUU is obtained by specialising the lemma 'DEFisIn'; the simpset sstt contains the Lemma 'ConsUnit'.

The theorem thmtt is obtained by performing the sequence of forwards deductions suggested at the end of the proof discussed above. This involves specialising, in turn, Lemmas 4.21, 4.20, and Theorem 4.16. Various antecedents are then eliminated using Modus Ponens and simplification with a simpset containing the available assumptions (including the induction hypothesis). Finally, thmtt is obtained after simplifying with the associativity of "@".

The final theorem for this part of the case study, shows that EqBA is a congruence for the Cons operation; from this, it is easy to show that it is also a congruence for the Unit function.

Lemma 4.23

$\lambda a_1 \lambda a_2 \lambda l_1 \lambda l_2. (\alpha) L.$

$(a_1 \equiv_{\text{At}} a_2) \equiv \text{TT} \ \& \ (l_1 \equiv_{\text{BA}} l_2) \equiv \text{TT} \supset$

$(a_1 :: l_1) \equiv_{\text{BA}} (a_2 :: l_2) \equiv \text{TT}$

(on the assumption that both 'DEFEqAt' and 'SymEqAt' hold.)

Proof By simplification, based upon the assumptions that

$(a_1 \equiv_{\text{At}} a_2) \equiv \text{TT}$, and that $(l_1 \equiv_{\text{BA}} l_2) \equiv \text{TT}$. Using 'SymEqAt', we also have that $(a_2 \equiv_{\text{At}} a_1) \equiv (a_1 \equiv_{\text{At}} a_2) \equiv \text{TT}$. Moreover, $\text{TT} \equiv$

$\partial(l_1 \equiv_{\text{BA}} l_2) \equiv \partial(l_1) \text{ and } \partial(l_2)$, using the 'DEFEqBA' lemma. Now,

applying the 'AndAnalysis' lemma, we have that $\partial(l_1) = TT$ and also that $\partial(l_2) = TT$.

So, computing with these (derived) assumptions, we have:-

$$\begin{aligned}
 (a_1 :: l_1) &=_{BA} (a_2 :: l_2) \\
 &= (\partial(l_1) \text{ and } (a_1 \leftarrow (a_2 :: l_2))) \text{ and} \\
 &\quad (l_1 =_{BA} ((a_2 :: l_2) - a_1)) \\
 &= (TT \text{ and } (a_1 \leftarrow (a_2 :: l_2))) \text{ and} \\
 &\quad (l_1 =_{BA} ((a_2 :: l_2) - a_1)) \\
 &= (a_1 \leftarrow (a_2 :: l_2)) \text{ and } (l_1 =_{BA} ((a_2 :: l_2) - a_1)).
 \end{aligned}$$

Now, $a_1 \leftarrow (a_2 :: l_2) = \partial(l_2) \text{ and } ((a_1 =_{At} a_2) \text{ or } (a_1 \leftarrow l_2)) =$
 $TT \text{ and } (TT \text{ or } (a_1 \leftarrow l_2)) = TT$, and also $(a_2 :: l_2) - a_1 =$
 $(a_2 =_{At} a_1) \Rightarrow l_2 \mid a_2 :: (l_2 - a_1) = l_2$.

$$\begin{aligned}
 \text{So, } (a_1 :: l_1) &=_{BA} (a_2 :: l_2) \\
 &= (a_1 \leftarrow (a_2 :: l_2)) \text{ and } (l_1 =_{BA} ((a_2 :: l_2) - a_1)) \\
 &= TT \text{ and } (l_1 =_{BA} l_2) \\
 &= l_1 =_{BA} l_2 \\
 &= TT.
 \end{aligned}$$

This completes the proof of Lemma 4.23.

QED

A tactic which proves this Lemma is:-

```

GENDEFCASESTACS*
IMPTAC.*S
CALCTAC' ssl

```

where:-

```

ssl = { .. ]- a2 =At a1 = TT }

```

The simprule in ssl is obtained by specialising 'SymEqAt' and simplifying with the assumption that $a_1 =_{At} a_2 = TT$.

4.6.5 Multisets: Discussion.

A detailed presentation of this case study has been given indicating the kind of interaction with LCF that was performed to generate the proofs of the required theorems.

The less formal, discursive proofs of lemmas, as well as their more formal tactical counterparts have evolved over several stages of refinement and development. A rough draft of the entire proof was done first by hand, in which many of the dependencies between

lemmas were determined. Here are two examples of the kind of problem encountered:-

(1) The "Cons" case of the symmetry theorem, Theorem 4.9, proved to be a major stumbling block, until the form of Lemma 4.8 (and its non-trivial proof) was discovered. Other hold-ups were encountered in sketching a proof of the transitivity theorem, Lemma 4.16, which, in turn, depended upon the discovery of the key Lemmas 4.15 and 4.14.

(2) In showing the commutativity theorem, Theorem 4.22, a certain amount of effort was expended in spotting Lemma 4.21 and its role there. This proof also required the transitivity theorem, and various congruence-like properties of the operators:- " $\equiv$ ", " $\leftarrow$ " and " $-$ " with respect to the equivalence predicates " $=_{At}$ " and " $=_{BA}$ " as appropriate.

Once the informal sketch proof was completed, the lemma sequence was determined, and eventually, successful tactical proofs of these Lemmas were performed, interactively. During the course of executing these tactical proofs, several errors were brought to light in the sketch proofs, and then corrected.

Some attempt was made to simplify the form of the tactical proofs; this involved choosing more convenient case analyses, simplrules and the order of quantifiers in a goal formula in order to determine induction variables. Finally, the detailed discussive proofs and accompanying tactics presented above were developed for presentation.

Note that many of the tactical case analyses performed involve the equivalence predicate, " $=_{At}$ ". In each instance, the undefined case was solved by using a specialisation of the definedness property, 'DEF Eq_{At} ', to derive a contradictory antecedent:-

4.7 Case studies using resolution-oriented tactics.

Most of the proofs in the case study given in this chapter have had the following general structure:—

Decompose the goal formula by following syntactic structure, eliminating implications by assuming the antecedent, eliminating quantifiers by simple generalisation, or by applying appropriate case-analysis or structural induction tactics.

New simprules may be added to local simpsets as a result of these eliminations of form from the goal. Each of these stages can be interleaved with (conditional) simplification, as necessary.

Eventually, atomic subgoals are obtained consisting of equations (or in-equations), to which further truth-valued case analysis of well-chosen subterms maybe applied.

Finally, if any subgoals remain, apply some well-chosen sequence of specific lemmas by substitution, or otherwise, to finish off.

The question that is studied in this Section concerns the extent to which tactical methods can be applied to "finishing off" residual subgoals at the end of a proof.

Typically, such "finishing off" could involve finding certain equations to be substituted into the goal (as with Theorem 4.22, for example). It may also involve applying well-chosen collections of simprules to the goal as well. Such substitutions and simprules will generally be provable from the local assumptions, and previously established lemmas and theorems by a short "forwards" derivation.

The approach pursued here is to attempt the application of basic ideas from the resolution-based school of theorem proving (see [Robinson65]) in order to automate this kind of detailed forwards reasoning within LCF's tactical proof methodology.

Avra Cohn originally developed a tactic, called RESTAC, which implemented a basic form of the Resolution Principle, and applied it in several case studies of her own (see [Cohn81], [CohnMilner82], [Cohn82]). The original version of RESTAC does not play a direct role in the presentation below, but served as a starting point and motivation for our resolution based tactics to be used later. A brief discussion of Cohn's RESTAC appears in Section 5.4.1.

The basic method used by both of the resolution tactics used

later is now briefly described. A detailed technical discussion can be found in Section 5.4.3. The idea is, broadly speaking, to derive new consequences by "forwards chaining"; this process tries to find some kind of match of local assumptions against antecedents of formulae from a specified collection of implicative formulae (some of which could also be assumptions). This kind of derivation is applied between each possible pairing of (equational) assumption with implicative formula, so giving a "breadth first" development of consequences.

In general, the resolution tactics do not affect the goal formula itself, but derive consequences from the goal's assumption list, and add them for later use. The validation component for the resolution tactic's simply take the new consequences generated, and discharge any that appear as hypotheses of the input theorem. This process effectively replaces all hypotheses corresponding to "new consequences" by those assumptions from which they were derived.

The main resolution tactic used below is called LINEARRESTAC and has ML type `:simpset # thm list -> tactic`. It incorporates aspects of the "Linear Input" refinement of the pure resolution principle. For standard text-book renditions of "Linear Input" resolution, see, for example, [Bundy83] and [ChangLee73].

This tactic is parameterised by a simpset and a list of theorems. The list of theorems is the collection of implicative formulae against which (equational) assumptions will be resolved. The simpset is used to simplify resolvents. Those resolvents that simplify to "TRUTH" are discarded. This provides facilities for filtering out "unwanted" consequences and also ensuring that resolvents have a certain shape. A more detailed description of LINEARRESTAC and its implementation in ML is given in Section 5.4.3.

In the studies described below, a re-implemented version of RESTAC is made use of. This is also described in Section 5.4.3.

4.7.1 Auxiliary tactics.

Several further tactics are now presented for use in the following resolution case studies. The first three tactics given

are more especially for use with RESTAC.

The first tactic, called USELEMMASTAC :thm list $\rightarrow$ tactic, adds the conclusion of each given theorem as an assumption of the goal. It also checks that the hypotheses of all the given theorems are already available as assumptions. The validation part for this tactic eliminates any of the given theorems from the hypotheses of the input theorem by discharging it and then using Modus Ponens with the corresponding theorem. The functional behaviour of this tactic is:-

w
ss
fml

USELEMMASTAC ([H_1]-fm₁; H_2]-fm₂; ... ; H_n]-fm_n])

w
ss
[fm ₁ ; fm ₂ ; ... ; fm _n] @ fml

(on the proviso that the union of all the hypotheses sets H_i is contained (up to alpha-conversion) within the assumption list fml. This is always the case when all of the hypotheses sets are empty.)

The second tactic is called CANONTAC and it ensures that all implicative assumptions are transformed into (possibly a conjunctive list of) implicative formulae of the form:-

$$\forall v_1 v_2 \dots v_n. (w_1 \& w_2 \& \dots \& w_m) \supset a$$

where each of the w_i 's can be an arbitrary PPLAMBDA formula and where the consequent formula, a , is atomic (i.e. an equation or inequation). Note that the consequent is quantifier free and atomic. The appropriate transformation from an arbitrary quantified implicative formula to such formulae can be specified by a derived inference rule, HNF : thm $\rightarrow$ thm list, and is formally defined in Section 5.4.6. The behaviour of this tactic is:-

w
ss
fml

CANONTAC

w
ss
fml'

where each formula in fml' is obtained as a result of applying the canonicalisation process informally indicated above. The length of fml' may be greater than that for fml , since conjunctive formulae are decomposed into their basic conjuncts. More formally, the list fml' is equal to:-

```
map concl (flat(map (HNF o ASSUME) fml))
```

The validation component for this tactic is as for RESTAC and LINEARRESTAC in that canonicalised hypotheses of an incoming theorem are eliminated by discharging them and applying Modus Ponens with a theorem proven by using HNF with an "old" assumption. The effect of this is to replace each canonicalised hypothesis by the original assumption from which it was derived using the HNF inference rule.

CANONTAC was not needed in order to use Cohn's original version of RESTAC. This is because the canonicalisation process is built into it directly, and consequently applied everytime the tactic is applied. This effort is wasted when RESTAC is applied repeatedly, since repeated canonicalisation of formulae and, more importantly, their consequences changes nothing. So, to reduce this overhead, the process has been factored out as a separate tactic that need only be applied once before the first application of RESTAC.

The third auxiliary tactic is called FINDGNDSSSTAC, which searches the assumption list for equational assumptions whose rhs is a named constant term of non-functional type (like Nil, or TT). In addition, the lhs of the equation must not be such a term; this is in order to reject instances of the reflexivity of equality, for example. Any such equations found may then be safely incorporated into the local simpset for use as simprules. The tactic only affects the local simpset within the goal.

FINDGNDSSSTAC is typically used to add new simprules of the above form that have been derived by application of a resolution tactic. The original RESTAC incorporated a similar process for adding new simprules after a resolution step.

The next two tactics presented are more general. The tactic ATOMTAC is a general purpose decomposition tactic defined by the following tactic expression:-

```
(GENTAC ? IMPTAC ? CONJTAC)*
```

As can be seen, this simply decomposes the goal according to its composition in terms of logical connectives, until atomic formulae are reached.

The tactic TRIVTAC examines the goal to see if it is an easily recognised tautology, with failure if it is not. The class of tautologies recognised is given in terms of the following tactic expression:-

(TRUETAC ? CONTRTAC ? ASSUMETAC ? (SYMTAC THEN ASSUMETAC))

The first tactic, TRUETAC simply checks whether the goal is "TRUTH"; the second tactic, CONTRTAC, checks whether there is a standard contradiction in the assumption list of the goal. The tactic ASSUMETAC scans the assumption list to see if there is a formula either exactly matching the goal or whose quantifiers can be instantiated to exactly match the goal; this must use one-way matching. The final tactic assumes that the goal is equational, applies the symmetry of equality, via SYMTAC, and then tries ASSUMETAC again.

4.7.2 The resolution case studies.

Before giving these case studies in detail, a few general remarks concerning the application of resolution tactics.

In general, a resolution tactic will be applied to particular subgoals, about which certain definite information is already known. This information may be in the form of assumptions introduced earlier (e.g. by an induction) or it may be in the form of already proven lemmas which are, at least, suspected of being related to the goal. Hence, collections of possibly useful lemmas are selected, in advance, for later use within resolution. When using RESTAC, such lemmas could be introduced with USELEMMASTAC, and when using LINEARRESTAC, these lemmas could form the collection of theorems to be resolved against.

All of this implies that the user of a resolution tactic has to know roughly which lemmas could be of benefit during the proof of a given subgoal from given assumptions. However, it is not necessary to select only those theorems which are known to participate in some proof; more theorems can be included as desired.

Both of the studies that are given below were based on the tactical proofs of various Theorems from the above case study and contain a variety of opportunities for the application of resolution tactics.

During the course of the application of a resolution tactic, various consequences will be derived. To assist reference to such consequences, they may be labelled by a token of the form 'Cn.m', where both n and m are numbers.

On occasion, the following instance of the TRANS rule for equality between truth-values was included among the lemmas:-

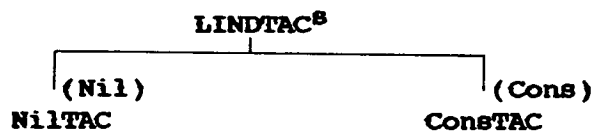
$$\vdash \forall t_1 t_2:tv. (t_1 = t_2) \ \& \ (t_2 = \text{TT}) \supset (t_1 = \text{TT})$$

This transitivity lemma will be referred to below as 'Transtv'.

By iterating a resolution tactic with this single implicative theorem, the transitive closure of the available equational assumptions, for a given goal, would be computed. The symmetry of equality has been taken into account within the matching processes of the resolution tactics (see Section 5.4.3).

4.7.2.1 First resolution case study.

The first study given here proves Theorem 4.16 from Section 4.6.3, the transitivity theorem for EqBA, and uses both RESTAC and LINEARRESTAC. The goal used is as for the original tactical proof, and hence includes all of the standard equivalence assumptions for the predicate $=_{At}$. The tactic is as follows:-



where:-

```

NilTAC  =  GENTAC*
           IMPTAC
           LINEARRESTAC( BASICSS , [ ThmA ] )
           FINDGNDSSSTAC8

ConstAC =  ATOMTAC
           USELEMMASTAC Lemmalst
           CANONTAC
           ( RESTAC
             (TRIVTAC ? FINDGNDSSSTAC8)
           )*
```

where:-

ThmA is the 'NullIMPNil' lemma from the theory 'LA'; Lemmalst contains the transitivity lemma, 'Transtv' mentioned above, the 'AndAnalysis' lemma and Lemmas 4.12 and 4.15.

(Also, recall from Section 4.5.4 that "?" means the "ORELSE"

tactical, in the context of the tactic composition notation).

The original tactic for this goal appears, at first sight, to be easier than the above. However, more is done there by explicit forwards deduction. In each case, an appropriate simprule is "invented" by application of resolution tactics using some less specific theorems. In the Nil case, LINEARRESTAC is applied once with a single theorem in order to generate the required simprule. Note that only one of the goal's antecedents is discharged, since as before the goal is reduced to a classical tautology by simplifying the remaining antecedent.

In the Cons case, RESTAC is used in order to allow every implicative assumption, like the induction hypothesis for example, to be resolved against. Before this, the goal is straightforwardly reduced by an application of ATOMTAC and some possibly useful lemmas added with USELEMMASTAC. It is interesting to note that, in this tactical proof, a single simprule of a simple form will be discovered to solve this case. In the previous tactical proof, four explicitly given rules are applied, not including the induction hypothesis, which is also assumed there for use as a simprule.

A more detailed discussion of this tactical proof is now given, commencing with a discussion of the Nil case. The tactic Niltac is applied to the goal:-

$\forall l_2 l_3. \text{Null } l_2 = \text{TT} \ \& \ (l_2 =_{\text{At}} l_3) = \text{TT} \supset \text{Null } l_3 = \text{TT}$

After applying GENTAC twice and IMPTAC once, we get the goal:-

$(l_2 =_{\text{At}} l_3) = \text{TT} \supset \text{Null } l_3 = \text{TT}$

The assumption list now includes the formula "Null $l_2 = \text{TT}$ " as well as the standard assumptions concerning the predicate $=_{\text{At}}$. The resolution tactic LINEARRESTAC is now applied (with arguments as specified above). The form of the 'NullIMPNil' lemma is:-

$\lambda l. \forall l_1. (\alpha) L. \text{Null } l_1 = \text{TT} \supset l_1 = \text{Nil}$

The single antecedent of this lemma is then successfully unified with the assumption "Null $l_2 \equiv TT$ " to produce the consequence:-

Cl.1 $l_2 \equiv Nil$

To justify the derivation process for this consequence, the following theorem is proven by using Modus Ponens:-

.]- $l_2 \equiv Nil$

on the hypothesis that " $Null\ l_2 \equiv TT$ ". No other consequences were found at this stage.

Finally, this new assumption is found by FINDGNDSTAC and included as a simprule; an application of SIMPTAC then solves this subgoal. The tactic ConstAC is now applied to the (simplified) Cons case goal:-

" $\forall\ l_2\ l_3.$

$(a' \leftarrow l_2) \text{ and } (l_1' \equiv_{BA} (l_2 - a')) \equiv TT \ \& \ (l_2 \equiv_{BA} l_3) \equiv TT \supset$

$(a' \leftarrow l_3) \text{ and } (l_1' \equiv_{BA} (l_3 - a')) \equiv TT$ "

with the assumptions that:-

" $\forall\ l_2\ l_3.$

$(l_1' \equiv_{BA} l_2) \equiv TT \ \& \ (l_2 \equiv_{BA} l_3) \equiv TT \supset (l_1' \equiv_{BA} l_3) \equiv TT$ "

" $\partial(l_1') \equiv TT$ "

" $\partial(a) \equiv TT$ "

plus the standard equivalence assumptions for $\equiv_{At}$. By applying the decomposition tactic, ATOMTAC, the two quantifiers and antecedents for the goal are removed giving:-

" $(a' \leftarrow l_3) \text{ and } (l_1' \equiv_{BA} (l_3 - a')) \equiv TT$ "

as the goal, with the additional assumptions:-

" $(a' \leftarrow l_2) \text{ and } (l_1' \equiv_{BA} (l_2 - a')) \equiv TT$ "

" $(l_2 \equiv_{BA} l_3) \equiv TT$ "

Next, USELEMMASTAC adds the following assumptions (e.g. 'Transtv', and the 'AndAnalysis' lemma):-

" $\forall\ t_1\ t_2. t_1 \equiv t_2 \ \& \ t_2 \equiv TT \supset t_1 \equiv TT$ "

" $\forall\ t_1\ t_2. t_1 \text{ and } t_2 \equiv TT \supset (t_1 \equiv TT) \ \& \ (t_2 \equiv TT)$ "

as well as Lemmas 4.12 and 4.15:-

" $\forall\ l_1\ l_2:(\alpha)L\ a:\alpha. l_1 \equiv_{BA} l_2 \equiv TT \supset (a \leftarrow l_1) \equiv (a \leftarrow l_2)$ "

" $\forall\ a\ a':\alpha\ l\ l':(\alpha)L.$

$(a \equiv_{At} a') \equiv TT \ \& \ (l \equiv_{BA} l') \equiv TT \supset$

$(l - a) \equiv_{BA} (l' - a') \equiv TT$ "

Note that any standard assumptions concerning $=_{At}$ that these Lemmas depend upon need not be included, as they are already in force within the present goal.

Applying CANONTAC leaves all of the assumptions alone, except for the 'AndAnalysis' lemma, which is split into the following two assumptions:-

$$\forall t_1 t_2:tv. t_1 \text{ and } t_2 = TT \supset t_1 = TT$$

$$\forall t_1 t_2:tv. t_1 \text{ and } t_2 = TT \supset t_2 = TT$$

The resolution tactic RESTAC is now applied. All of the implicative formulae in the present assumption list are then available to be matched against by all of the equational assumptions. In the first iteration of the derivation process, 12 new consequences are found, 3 of which are equational formulae.

Among the new consequences were:-

$$Cl.2 \quad (a' \leftarrow l_2) = TT$$

$$Cl.3 \quad (l_1' =_{BA} (l_2 - a')) = TT$$

$$Cl.4 \quad \forall a. (a \leftarrow l_2) = (a \leftarrow l_3)$$

$$Cl.5 \quad (l_1' =_{BA} l_2) = TT \supset (l_1' =_{BA} l_3) = TT$$

The first two came from the (split) 'AndAnalysis' lemma using the assumption that:-

$$\forall (a' \leftarrow l_2) \text{ and } (l_1' =_{BA} (l_2 - a')) = TT$$

The consequence Cl.4 is derived via Lemma 4.12, and Cl.5 from the original induction hypothesis. The consequence Cl.5 is given as an example of an inference which is potentially useful, on syntactical grounds, but which does not lead anywhere (in this proof)! The rest of the consequences produced were obtained via the 'Transtv' rule.

A second iteration of the derivation process is started immediately which this time giving the single new consequence:-

$$Cl.6 \quad (a' =_{At} a') = TT$$

(using 'Transtv' and the two assumptions that " $\partial(a') = TT$ " and the reflexivity of $=_{At}$, which is " $\forall a. (a =_{At} a) = \partial(a)$ ".)

This completes the first application of RESTAC. Next, TRIVTAC is applied (see above). In the present case, the goal is not an easily recognised tautology and so the next tactic to be applied is FINDGNDSSSTAC. This adds the equational consequences mentioned

above whose rhs is "TT". However, simplification of the goal with the new simpset has no effect. This has completed one round of the final part of ConstAC.

The next tactic applied is RESTAC. The first internal iteration produces 15 new consequences, of which 4 are equational. The eventually useful consequences are:-

$$\text{Cl.7} \quad (l_2 - a') =_{\text{BA}} (l_3 - a') = \text{TT}$$

$$\text{Cl.8} \quad \forall a. (a \leftarrow l_2) = \text{TT} \supset (a \leftarrow l_3) = \text{TT}$$

$$\text{Cl.9} \quad \forall l_3. ((l_2 - a') =_{\text{BA}} l_3) = \text{TT} \supset (l_1' =_{\text{BA}} l_3) = \text{TT}$$

Consequence Cl.7 was found by matching consequence Cl.6 together with the assumption that " $l_2 =_{\text{BA}} l_3 = \text{TT}$ " against Lemma 4.15. The consequence Cl.8 is obtained by using consequence Cl.4 and resolving it against the first antecedent of 'Transtv'. Finally, the consequence Cl.9 is obtained by matching Cl.3 against the first antecedent of the induction hypothesis.

The second internal iteration produces 5 new consequences, all of which are equational, and include the following two:-

$$\text{Cl.10} \quad (a' \leftarrow l_3) = \text{TT}$$

$$\text{Cl.11} \quad (l_1' =_{\text{BA}} (l_3 - a')) = \text{TT}$$

The consequence Cl.10 was obtained by matching Cl.2 against Cl.8. The other consequence above is obtained by matching Cl.7 against Cl.9. This completes the second internal iteration of the second application of RESTAC.

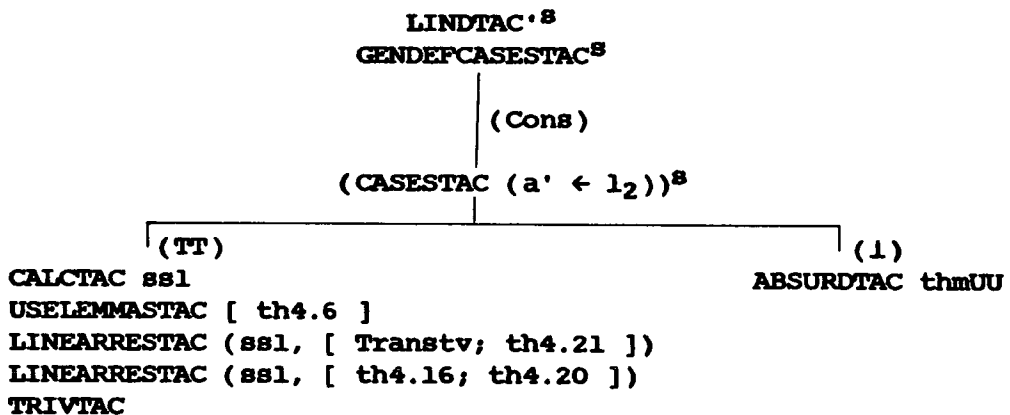
The next tactic applied is TRIVTAC which fails for the same reasons as before. Instead, the tactic FINDGNDSSSTAC is applied. This adds all of the equations having "TT" as their rhs to the local simpset. But now, using SIMPTAC, the final two consequences mentioned above transform the goal to:-

$$\text{"TT and TT} = \text{TT"}$$

which is then easily solved by simplification. This completes the entire proof of Theorem 4.16 using the tactic above.

4.7.2.2 Second resolution case study.

The second case study involving resolution tactics is concerned with Theorem 4.22, the commutativity of $\otimes$ with respect to the equivalence predicate $\approx_{BA}$. The goal used here is as for the previously given tactical proof. The initial simpset included the Lemmas 4.6, 4.17, 4.18, 4.19; the assumption list consists of the standard equivalence assumptions concerning $\approx_{At}$. The tactic is as follows:-



where:-

$thmUU = .]- \partial(a' \leftarrow l_2) = \partial(a') \text{ and } \partial(l_2)$.

ssl contains the 'ConsUnit' lemma, and the associativity of $\otimes$.

The Lemmas Transtv , th4.6 , th4.16 , th4.20 , and th4.21 all correspond in an obvious way to Lemmas proven independently of Theorem 4.22.

Most of the above tactic is as for the previously given tactical proof of Theorem 4.22, including the "I" case from the application of CASESTAC (but see Section 4.7.3). The only difference is that resolution tactics are being applied to calculate the substitution required and also arrange for its application. The two applications of the resolution tactics given above could be combined into a single application, where the union of the two theorem lists is used. However, this lead to the generation of an excessive number of intermediate lemmas, due to the presence of Theorems 4.16, 4.20 and Transtv together, giving rise to significant storage problems in the machine. By splitting the

lemmas into two, the generation of many irrelevant lemmas is avoided.

The remainder of the discussion focuses upon the detailed evaluation of the tactic in the "TT" case. The goal at this point has the form:—

$$"(l_1' @ l_2) \text{=BA} ((l_2 - a') @ (a' :: l_1')) \equiv \text{TT}"$$

with the assumptions:—

$$"(a' \leftarrow l_2) \equiv \text{TT}"$$

$$"\forall l_2. ((l_1' @ l_2) \text{=BA} (l_2 @ l_1')) \equiv \text{TT}"$$

$$"\partial(a') \equiv \text{TT}"$$

$$"\partial(l_1') \equiv \text{TT}"$$

$$"\partial(l_2) \equiv \text{TT}"$$

plus the standard equivalence assumptions concerning =At . The free variables a' and l_1' were introduced during the Cons case of the list induction tactic. CALCTAC ssl is now applied to express the occurrence of Cons in terms of "Unit" and "@":—

$$"(l_1' @ l_2) \text{=BA} ((l_2 - a') @ (\text{Unit}(a') @ l_1')) \equiv \text{TT}"$$

under the same assumptions. The next tactic is USELEMMASTAC, which then adds (the body of) Theorem 4.6 to the assumptions:—

$$"\forall l_1. (l_1 \text{=BA} l_1) \equiv \partial(l_1)"$$

The first application of LINEARRESTAC is now given with the simpset ssl as described above and the theorems 'Transtv' and Lemma 4.21:—

$$]- \forall t_1 t_2:tv. (t_1 \equiv t_2) \ \& \ (t_2 \equiv \text{TT}) \supset (t_1 \equiv \text{TT})$$

$$....]- \forall l:(\alpha)L \ a_1 \ a_2:\alpha.$$

$$(a_1 \text{=At} a_2) \equiv \text{TT} \ \& \ (a_1 \leftarrow l) \equiv \text{TT} \supset$$

$$(l \text{=BA} ((l - a_1) @ \text{Unit}(a_2))) \equiv \text{TT}$$

The first internal iteration of LINEARRESTAC produces 16 new consequences, of which 3 are equational. Consequences found at this stage that eventually prove to be useful were:—

$$\text{C2.1} \quad a' \text{=At} a' \equiv \text{TT}$$

$$\text{C2.2} \quad l_1' \text{=BA} l_1' \equiv \text{TT}$$

$$\text{C2.3} \quad l_2 \text{=BA} l_2 \equiv \text{TT}$$

$$\text{C2.4} \quad \forall a_2. (a' \text{=At} a_2) \equiv \text{TT} \supset$$

$$l_2 \text{=BA} ((l_2 - a') @ \text{Unit}(a_2)) \equiv \text{TT}$$

The consequence C2.1 was found by matching the assumption

" $\partial(a') \equiv \text{TT}$ " with the second antecedent of 'Transtv'; the first

antecedent is then resolved with the assumption:-

$$\text{"}\forall a. (a \text{ =}_{At} a) \equiv \partial(a)\text{"}$$

Similar derivations give the next two consequences. The final consequence is obtained by matching the second antecedent of Lemma 4.21 with the assumption " $(a' \leftarrow l_2) \equiv TT$ ".

Examples of other, less useful, consequences found at this stage were:-

$$C2.5 \quad \forall t_1. t_1 \equiv \partial(a') \supset t_1 \equiv TT$$

$$C2.6 \quad \forall l_1. (l_1 \text{ =}_{BA} l_1) \equiv TT \supset \partial(l_1) \equiv TT$$

produced by matching various truth-valued equations with the second antecedent of 'Transtv'.

The second internal iteration of this application of LINEARRESTAC produces 4 consequences, all of which are equational. However, three of these were produced on the last iteration as well; the single new consequence is:-

$$C2.7 \quad l_2 \text{ =}_{BA} ((l_2 - a') @ \text{Unit}(a')) \equiv TT$$

This is obtained by matching consequence C2.1 with the antecedent of consequence C2.4. This completes the first application of LINEARRESTAC, which adds the equational consequences C2.1, C2.2, C2.3 and C2.7 to the assumption list of the goal.

The second application of LINEARRESTAC is now begun, again with simpset ssl, but with Theorems 4.16 and 4.20:-

$$\dots]- \forall l_1 l_2 l_3 : (\alpha)L.$$

$$(l_1 \text{ =}_{BA} l_2) \equiv TT \quad \& \quad (l_2 \text{ =}_{BA} l_3) \equiv TT \supset$$

$$(l_1 \text{ =}_{BA} l_3) \equiv TT$$

$$\dots]- \forall l_1 l_2 l_3 l_4 : (\alpha)L.$$

$$(l_1 \text{ =}_{BA} l_2) \equiv TT \quad \& \quad (l_3 \text{ =}_{BA} l_4) \equiv TT \supset$$

$$(l_1 @ l_3) \text{ =}_{BA} (l_2 @ l_4) \equiv TT$$

The first internal iteration produces 29 new consequences, 19 of which are equational. A factor contributing to this increase in the number of equations is that the consequents of the above implicative theorems may also match any of the antecedents, thus giving rise to further opportunities for resolution. The only significant consequence drawn at this stage is:-

$$C2.8 \quad (l_2 @ l_1') \text{ =}_{BA} ((l_2 - a') @ (\text{Unit}(a') @ l_1')) \equiv TT$$

This is derived by matching the consequence C2.7 to the first

antecedent of Theorem 4.20 and matching the second antecedent with consequence C2.2. The resolvent is then simplified with the associativity of "@" from simpset ssl. This is an example of where the simpset ssl is used to guide the course of the resolution process. By ensuring that all terms involving "@" in resolvents are associated to the right, this prevents the generation of further resolvents with left-associated "@" terms. In addition, simplification filters out any resolvents that simplify to "TRUTH" (i.e. that are tautologous with respect to the given simpset).

A sample of the other consequences generated at this stage is given below:-

$$C2.9 \quad ((l_2 @ l_2) =_{BA}$$

$$(l_2 - a') @ (Unit(a') @ (l_2 - a') @ Unit(a')) =_{TT}$$

which is obtained by matching C2.7 to both antecedents of Theorem 4.20 and then applying associativity via simplification.

$$C2.10 \quad \forall l_2'. (l_1' @ (l_2' @ l_2)) =_{BA} (l_2' @ (l_1' @ l_2)) =_{TT}$$

This is obtained by resolving the induction hypothesis:-

$$"\forall l_2. (l_1' @ l_2) =_{BA} (l_2 @ l_1') =_{TT}"$$

with the first antecedent of Theorem 4.20, and the consequence, C2.2 (which is " $l_2 =_{BA} l_2 =_{TT}$ "), with it's second antecedent. As before, associativity is applied. Note that the quantified variable in C2.10 has been primed to avoid a clash of free and bound variables.

The second internal iteration of LINEARRESTAC produces 51 consequences, all of them equational. At this stage, the goal formula itself was derived as a consequence:-

$$C2.11 \quad (l_1' @ l_2) =_{BA} ((l_2 - a') @ (Unit(a') @ l_1')) =_{TT}$$

This was derived by resolving the induction hypothesis:-

$$"\forall l_2. (l_1' @ l_2) =_{BA} (l_2 @ l_1') =_{TT}"$$

with the first antecedent of Theorem 4.16, and matching consequence C2.8 with it's second antecedent.

Many other consequences were generated during the iteration, before the above was derived (the derivation process halted as soon as this was found). To take a typical example:-

$$C2.12 \quad \forall l_2'.$$

$$((l_2 @ (l_1' @ l_2')) =_{BA}$$

$$((l_2 - a') @ (Unit(a') @ (l_2' @ l_1')))) =_{TT}$$

This consequence was obtained by matching C2.7 to the first antecedent of Theorem 4.20 and the induction hypothesis to the second. This completes the second internal iteration of LINEARRESTAC. Most of the (equational) consequences are then added to the assumption list, after removing alpha-equivalent repetitions.

The next tactic applied is TRIVTAC which discovers the goal formula within the assumption list, after first scanning them for standard contradictions. This then solves the goal, and completes the entire tactic.

4.7.3 Discussion.

In both case studies, most of the consequences drawn did not actually play any part in the proof in hand, as might be expected for a "breadth-first" development. In the second case study, about 90% of the equational consequences returned did not contribute to the final result. However, in the first case study, about half the equational consequences returned did so. The number of intermediate consequences (mainly implicative formulae) is (naively) estimated to have been about double the number of equational consequences produced, since each original implication had at most two antecedents to eliminate.

Although the proofs given above made use of tactics related to the resolution principle, a considerable amount of careful planning was still necessary for the attempts to be successful. Knowledge about the proof to be conducted was represented by the choice of additional lemmas that were used. For the first study, it was anticipated that a successful outcome would be obtained through new simprules of the appropriate form being derived. In the second case, it was known in advance that a consequence matching the goal would be derived.

Additionally, knowledge about the proofs was represented by the way in which the resolution tactics were used.

In the first case study, this knowledge took the form of a calculation of a simprule from the 'NullIMPNil' lemma within the Nil case; RESTAC was used in the Cons case in order to engage the

implicative induction hypothesis.

In the second case study the resolution phase was split into two applications of LINEARRESTAC. This was done in order to curtail the number of irrelevant consequences produced. Therefore, the choice of the partition of the lemmas used reflected the sequence of substitutions to be performed.

Note that the "1" case was solved there by ABSURDTAC, could have been tackled by resolution, although at greater expense than the solution actually given. The problem is to solve the goal:-

$$"1 \equiv TT"$$

on the (contradictory) assumptions that:-

$$"a' \leftarrow l_2 \equiv 1"$$

$$"\partial(a') \equiv TT"$$

$$"\partial(l_2) \equiv TT"$$

Note that since the goal formula has the form of a standard contradiction, the only way to solve the goal is to explicitly prove that the given assumptions are inconsistent, by deriving a standard contradiction. Clearly, some extra explicit knowledge is required, and this might be given by the definedness property for "IsIn", which is the equation:-

$$"\forall a:\alpha \ 1:(\alpha)L. \ \partial(a \leftarrow 1) \equiv \partial(a) \text{ and } \partial(1)"$$

So, in order to apply resolution, some further implicative lemmas must be brought into play as well. A suitable set of such lemmas might be:-

$$"\forall x \ y \ z:tr. \ (x \equiv y) \ \& \ (y \equiv z) \supset (x \equiv z)"$$

$$"\forall x:tr. \ x \equiv 1 \supset \partial(x) \equiv 1"$$

$$"\forall x \ y:tr. \ (x \equiv TT) \ \& \ (y \equiv TT) \supset (x \text{ and } y \equiv TT)"$$

From the last two properties, the following can be derived:-

$$"\partial(a' \leftarrow l_2) \equiv 1"$$

$$"\partial(a') \text{ and } \partial(l_2) \equiv TT"$$

Using the instance of the TRANS rule given above in two ways, we would have that:-

$$"\forall x:tr \ a:\alpha \ 1:(\alpha)L. \ (x \equiv \partial(a \leftarrow 1)) \supset (x \equiv \partial(a) \text{ and } \partial(1))"$$

$$"\forall x:tr. \ (x \equiv \partial(a') \text{ and } \partial(l_2)) \supset (x \equiv TT)"$$

Note that no type instantiation is required to resolve the second antecedent with the definedness property of "IsIn". The first

antecedent of this is then eliminated to give:-

$$"1 \equiv \partial(a') \text{ and } \partial(l_2)"$$

Again, no type instantiation is required here. Finally, by applying this equation to a previously derived implication, we get the standard contradiction:-

$$"1 \equiv \text{TT}"$$

This would then be recognised as solving the goal, and the proof completed. In some sense, the manoeuvres performed by the resolution tactic are mimicking the evaluation that is performed when ABSURDTAC is applied.

4.8 Showing Simulation Correctness.

As promised earlier, we now show that the given "implementation" does, indeed, simulate the desired multiset algebra freely generated over a domain of generators, A. To simplify the discussion here, we shall assume that A is a fixed (but arbitrary), flat domain whose equality relation is represented by the continuous equivalence predicate, EqAt (or =_{At}). Formally this means that:-

$$\forall a, a': A. a = a' \Leftrightarrow \partial(a) = \partial(a') \ \& \ (a =_{\text{At}} a') \equiv \partial(a)$$

Furthermore, the flatness of A ensures that the list domain (A)L is also flat.

This simplifying assumption is justified by noting that a flat domain is simply a "lifted" set of values, and that instead of working with an equivalence relation on that set, we deal with the quotient set induced by the equivalence relation.

The method used here for showing correctness of simulation is to construct from it a space of denotations which can then be given a "natural" commutative monoid structure. This is then shown to satisfy the freeness criterion.

The first step is to introduce the relation $\approx_{\text{BA}}$ on (A)L that is represented by the continuous predicate =_{BA} :-

$$\forall l_1, l_2: (A)L.$$

$$l_1 \approx_{\text{BA}} l_2 \Leftrightarrow \partial(l_1) = \partial(l_2) \ \& \ (l_1 =_{\text{BA}} l_2) \equiv \partial(l_1)$$

Lemma 2.11 shows that $\approx_{\text{BA}}$ is a (continuous) equivalence relation on (A)L, since =_{BA} is a (continuous) equivalence predicate on (A)L.

Now, define Q_{BA} to be the quotient set induced by $\approx_{BA}$:-

$$Q_{BA} = \{ [1] \mid 1:(A)L \}$$

where, for each $1:(A)L$, $[1] = \{ 1':(A)L \mid 1 \approx_{BA} 1' \}$

The set Q_{BA} is given a flat ordering with $[1]$ as the least element. By the definition of Q_{BA} , the "natural" map, $\lambda 1:(A)L. [1]$ is onto and also continuous. Hence, we have that:-

$$\forall q \in Q_{BA}. \exists 1 \in (A)L. q = [1].$$

The following operations may be defined:-

$$V:A \rightarrow Q_{BA}$$

by the equation:-

$$\forall a:A. V(a) = [\text{Unit}(a)]$$

and the binary operation $- \circ - : Q_{BA}^2 \rightarrow Q_{BA}$

by the property:-

$$\forall l_1 l_2:(A)L. [l_1] \circ [l_2] = [l_1 \circ l_2]$$

This axiom precisely defines one function since the natural map is onto and that $\approx_{BA}$ is a congruence for the concatenation function $(- \circ -)$ on $(A)L$. These properties can show that:-

$$\forall q_1 q_2 \in Q_{BA}. \exists! q_3 \in Q_{BA}. q_1 \circ q_2 = q_3$$

Finally, we can define $[] \in Q_{BA}$ by the equation:-

$$[] = [\text{Nil}]$$

Using these introduced functions and constants, we can now prove the following lemma:-

Lemma 4.24

$([], \circ)$ is a commutative monoid on Q_{BA} .

Proof

Throughout this proof, we shall need to use the fact that $\forall q \in Q_{BA}. \exists 1:(A)L. q = [1]$. The use of this property will be implicit, and we slip between, for example, q_2 and $[l_2]$ as the need arises.

1) Bi-strictness.

$\forall q \in Q_{BA}. 1_{Q_{BA}} \circ q = [1 \circ 1] = [1] = 1_{Q_{BA}}$
and similarly for $q \circ 1_{Q_{BA}}$.

2) Left and Right identity.

$\forall q \in Q_{BA}. [] \otimes q = [Nil] \otimes [1] = [Nil \otimes 1] = [1] = q$
and similarly for $q \otimes []$

3) Associativity.

$\forall q_1, q_2, q_3 \in Q_{BA}.$

$$\begin{aligned}
 (q_1 \otimes q_2) \otimes q_3 &= ([l_1] \otimes [l_2]) \otimes [l_3] = ([l_1 \otimes l_2]) \otimes [l_3] \\
 &= [(l_1 \otimes l_2) \otimes l_3] = [l_1 \otimes (l_2 \otimes l_3)] \\
 &= [l_1] \otimes ([l_2 \otimes l_3]) = [l_1] \otimes (l_2 \otimes [l_3]) \\
 &= q_1 \otimes (q_2 \otimes q_3)
 \end{aligned}$$

4) Commutativity.

Let $q_1, q_2 \in Q_{BA}$. Suppose that either one, say q_1 , is undefined. Clearly, $q_1 = [1]$, and so $q_1 \otimes q_2 = [1 \otimes l_2] = [1] = [l_2 \otimes 1] = q_2 \otimes q_1$. On the other hand, suppose that both q_1 and q_2 are defined. Then $q_1 = [l_1]$ and $q_2 = [l_2]$ where both l_1 and l_2 are defined, by the flatness of Q_{BA} . Hence, by the commutativity property for Eq_{BA} , we have that $(l_1 \otimes l_2) =_{BA} (l_2 \otimes l_1) = \partial(l_1)$ and $\partial(l_2) = TT$. Therefore, $[l_1 \otimes l_2] = [l_2 \otimes l_1]$, and so we can calculate that:—

$$\begin{aligned}
 q_1 \otimes q_2 &= [l_1] \otimes [l_2] = [l_1 \otimes l_2] \\
 &= [l_2 \otimes l_1] = [l_2] \otimes [l_1] \\
 &= q_2 \otimes q_1
 \end{aligned}$$

This completes the proof that Q_{BA} with $([], \otimes)$ is a (continuous, bi-strict) commutative monoid.

QED

We shall now proceed to show that Q_{BA} is freely-generated by the fixed (but arbitrary) flat domain of atoms A — the freeness criterion holds for Q_{BA} .

Theorem 4.25

$(\square, @)$ is the multiset algebra freely generated over A .

Proof

Let (Z, P) be any (continuous, bi-strict) monoid (with carrier, B , say), and let $f : A \rightarrow B$ be any strict (continuous) function. Now, in Section 3.4, it was formally shown that an algebra isomorphic to $(Nil, @)$ with carrier $A(L)$ gave a Scott-continuous bi-strict monoid freely generated by the domain A . Let $m : (A)L \rightarrow B$ be defined as:

$$\forall l : (A)L. m(l) = (FreeMonoid (Z, P) f)(l)$$

So, we have immediately that:-

$$m(1) = 1$$

$$m(Nil) = Z$$

$$\forall l_1 l_2 : (A)L. m(l_1 @ l_2) = P(m(l_1), m(l_2))$$

and that $\forall a : A. m(\text{Unit}(a)) = f(a)$. The strict monoid morphism m is the unique such morphism with the above properties. In addition, we also have that:-

$$\forall a : A \ l : (A)L. m(a :: l) = m(\text{Unit}(a) @ l) =$$

$$P(m(\text{Unit}(a)), m(l)) = P(f(a), m(l))$$

In order to naturally "extend" the function $m : (A)L \rightarrow B$ to a function $m_Q : Q_{BA} \rightarrow B$, the following congruence property is required:-

$$\forall l_1 l_2 : (A)L. l_1 \approx_{BA} l_2 \supset m(l_1) = m(l_2)$$

However, because m is strict, the following statement is sufficient:-

$$\forall l_1 l_2 : (A)L. (l_1 \approx_{BA} l_2) = \text{TT} \supset m(l_1) = m(l_2)$$

This is proved later on (as Lemma 4.26); we use it here to justify the following definition of $m_Q : Q_{BA} \rightarrow B$ such that:-

$$\forall l : (A)L. m_Q([l]) = m(l)$$

Now, there is exactly one such function that satisfies this definition. This is because every element q in Q_{BA} is an image of some list l in $(A)L$ under the natural map:-

$$\forall q \in Q_{BA}. \exists l : (A)L. q = [l].$$

Moreover, if $[l_1] = [l_2]$ then $l_1 \approx_{BA} l_2$ and so, by the above, $m(l_1) = m(l_2)$. From these facts, we can show that:-

$$\forall q \in Q_{BA}. \exists! b : B. m_Q(q) = b$$

The continuity of m_Q follows from the continuity of m and that Q_{BA} is a flat domain. Now, from the definition above we can show that:-

$$(a) \ m_Q(1_{Q_{BA}}) \equiv m_Q([1]) \equiv m(1) \equiv 1.$$

$$(b) \ m_Q(\square) \equiv m_Q([Nil]) \equiv m(Nil) \equiv Z$$

$$(c) \ \forall q_1 \ q_2 \in Q_{BA}.$$

$$\begin{aligned} m_Q(q_1 \oplus q_2) &\equiv m_Q([l_1] \oplus [l_2]) \equiv m_Q([l_1 \oplus l_2]) \\ &\equiv m(l_1 \oplus l_2) \equiv P(m(l_1), m(l_2)) \\ &\equiv P(m_Q([l_1]), m_Q([l_2])) \\ &\equiv P(m_Q(q_1), m_Q(q_2)) \end{aligned}$$

Hence, m_Q is a strict commutative monoid morphism from $(\square, \oplus)$ to (Z, P) . Also, we have that for each $a : A$, $m_Q(V(a)) \equiv m_Q([Unit(a)]) \equiv m(Unit(a)) \equiv f(a)$. Hence, the morphism m_Q extends f (through V). We finally have to show uniqueness of m_Q . Suppose that $G : Q_{BA} \rightarrow B$ is any morphism from $(\square, \oplus)$ to (Z, P) which extends the valuation $f : A \rightarrow B$. This, of course, means that:-

$$G(1_{Q_{BA}}) \equiv 1_{Q_{BA}}$$

$$G(\square) \equiv Z$$

$$\forall q_1 \ q_2 \in Q_{BA}. \ G(q_1 \oplus q_2) \equiv P(G(q_1), G(q_2))$$

and finally that:-

$$\forall a : A. \ G(V(a)) \equiv f(a)$$

The function $H : (A)L \rightarrow B$ is now defined by:-

$$H(1) \equiv G([1])$$

Clearly, we have that:-

$$H(1) \equiv G([1]) \equiv G(1) \equiv 1_{BA}$$

$$H(Nil) \equiv G([Nil]) \equiv G(\square) \equiv Z$$

$$\forall l_1 \ l_2 : (A)L.$$

$$\begin{aligned} H(l_1 \oplus l_2) &\equiv G([l_1 \oplus l_2]) \equiv G([l_1] \oplus [l_2]) \equiv P(G([l_1]), G([l_2])) \\ &\equiv P(H(l_1), H(l_2)) \end{aligned}$$

Hence, H is a monoid morphism from $(Nil, \oplus)$ to (Z, P) . In addition, we also have that:-

$$\forall a : A. \ H(Unit(a)) \equiv G([Unit(a)]) \equiv G(V(a)) \equiv f(a)$$

Hence, the morphism H extends f , via $Unit$.

So, by the freeness criterion on $(A)L$ (proven, for example, in Section 3.4) we have that H is the unique monoid morphism from $(Nil, \oplus)$ to (Z, P) . Since $m : (A)L \rightarrow B$ also satisfies this property

we have that

$$\forall l:(A)L. m(l) \equiv H(l)$$

So, this in turn means that

$$\forall l:(A)L. m_Q([l]) \equiv m(l) \equiv H(l) \equiv G([l])$$

and finally we have that:-

$$\forall q \in Q_{BA}. m_Q(q) \equiv G(q)$$

i.e. m_Q is unique.

QED

Next, the proof of the lemma used above is given.

Lemma 4.26

$$\forall l_1 l_2:(A)L. (l_1 =_{BA} l_2) \equiv TT \supset m(l_1) \equiv m(l_2)$$

Proof By list induction on l_1 .

Suppose that $l_1 \equiv I$. This falsifies the assumption that $(l_1 =_{BA} l_2) \equiv TT$.

Suppose that $l_1 \equiv Nil$. We then have that $(Nil =_{BA} l_2) \equiv TT$. Therefore, $l_2 \equiv Nil \equiv l_1$. Therefore, $m(l_1) \equiv m(l_2)$.

Suppose that $l_1 \equiv (a :: l_3)$ where $\partial(a) \equiv \partial(l_3) \equiv TT$ and that we assume that $(l_1 =_{BA} l_2) \equiv TT$ holds with induction hypothesis:-

$$\forall l_4:(A)L. l_3 =_{BA} l_4 \equiv TT \supset m(l_3) \equiv m(l_4)$$

Now, evaluating $((a :: l_3) =_{BA} l_2) \equiv TT$ gives:-

$$(a \leftarrow l_2) \text{ and } (l_3 =_{BA} (l_2 - a)) \equiv TT$$

So, we then have that $(a \leftarrow l_2) \equiv TT$ and $(l_3 =_{BA} (l_2 - a)) \equiv TT$. By appealing to the induction hypothesis (with $(l_2 - a)$ for l_4) we immediately get $m(l_3) \equiv m(l_2 - a)$. By applying lemma 4.27 (see below):-

$$\forall l:(A)L \ a:A. a \leftarrow l \equiv TT \supset m(l) \equiv P(f(a), m(l - a))$$

with l_2 for l , we have that:-

$$a \leftarrow l_2 \equiv TT \supset m(l_2) \equiv P(f(a), m(l_2 - a))$$

We already know that $a \leftarrow l_2 \equiv TT$, and so after discharging we get:-

$$\begin{aligned} m(l_2) &\equiv P(f(a), m(l_3)) \\ &\equiv m(a :: l_3) \\ &\equiv m(l_1) \end{aligned}$$

This completes the proof of Lemma 4.26.

QED

We now give the proof of the lemma used above.

Lemma 4.27

$\forall l:(A)L\ a:A.$

$$a \leftarrow l \equiv TT \supset m(l) \equiv P(f(a), m(l - a))$$

Proof Without loss of generality, we may assume that $\partial(a) \equiv TT$ (since otherwise if $a = l$ then $a \leftarrow l \equiv l$, contradicting the antecedent). Now, proceed by list induction in l .

Supposing that either $l = l$ or $l = Nil$ falsify the antecedent. Hence, we may suppose that l has the form $(a' :: l_2)$ where $\partial(a') \equiv TT \equiv \partial(l_2)$ with the induction hypothesis:-

$$(a \leftarrow l_2) \equiv TT \supset m(l_2) \equiv P(f(a), m(l_2 - a))$$

So, proceeding, we may now assume that

$$a \leftarrow (a' :: l_2) \equiv TT$$

Now, by evaluation we then get that:-

$$(a' =_{At} a') \text{ or } (a \leftarrow l_2) \equiv TT$$

From this we get that either $(a =_{At} a') \equiv TT$, or $(a =_{At} a') \equiv FF$ and $(a \leftarrow l_2) \equiv TT$. (Since we already know that $\partial(a) \equiv \partial(a') \equiv TT$).

Taking the alternatives in turn, assume that $(a =_{At} a') \equiv TT$. Hence, by the equality representation assumption, we have that $a = a'$. So, $(l - a) \equiv (a' :: l_2) - a \equiv l_2$.

Hence, $m(l) \equiv m(a' :: l_2)$

$$\equiv P(f(a'), m(l_2))$$

$$\equiv P(f(a'), m(l - a)) \quad (\text{since } (l - a) \equiv l_2)$$

$$\equiv P(f(a), m(l - a)) \quad (\text{since } a = a')$$

completing this case.

On the other hand, assume that $(a =_{At} a') \equiv FF$ and that $(a \leftarrow l_2) \equiv TT$. So, $(l - a) \equiv (a' :: l_2) - a \equiv a' :: (l_2 - a)$.

Now, $m(l) \equiv m(a' :: l_2) \equiv P(f(a'), m(l_2))$. From the above, we know that $(a \leftarrow l_2) \equiv TT$, and so from the induction hypothesis we have that $m(l_2) \equiv P(f(a), m(l_2 - a))$.

$$\begin{aligned}
\text{Hence, } m(1) &\equiv P(f(a'), P(f(a), m(l_2 - a))) \\
&\equiv P(P(f(a'), f(a)), m(l_2 - a)) \\
&\equiv P(P(f(a), f(a')), m(l_2 - a)) \\
&\equiv P(f(a), P(f(a')), m(l_2 - a))
\end{aligned}$$

(since (Z, P) is a commutative monoid).

$$\text{Now, } m(1 - a) \equiv m(a' :: (l_2 - a)) \equiv P(f(a'), m(l_2 - a))$$

$$\text{Hence, } m(1) \equiv P(f(a), P(f(a'), m(l_2 - a))) \equiv P(f(a), m(1 - a))$$

So, in all cases, we have shown that the desired relationship holds and so the proof is completed.

QED

Chapter 5

Proof construction aids.

In this chapter, various LCF packages, tactics and other tools for performing and generating proofs in LCF are described. Most of them were developed for use within (and as a result of) the case studies presented above. All of the tools mentioned here are defined in ML.

We begin by describing both the axiomatisation package and the structural induction (and case analysis) package which have been used throughout the case studies above. The description follows what happens for a specific example, indicating the general pattern of behaviour as it does so.

Next, resolution-oriented tactics, such as LINEARRESTAC, are discussed. They are defined within ML in a way which allows for extensions and further parameterisation. The structure of the definitions using ML is directly related to the structure of the underlying algorithm used. Because of this, variations in the algorithm used are easily accommodated. Other resolution oriented tactics, called CONSEQTAC, EVALTAC and PROGTAC, are introduced.

LCF generally encapsulates inference in terms of functional or procedural abstraction. It turns out that, for a wide class of implicative PPLAMBDA theorems, there is a corresponding inference rule and tactic. For example, many of the basic inference rules of PPLAMBDA, with their natural tactics, could be generated in this way, from appropriate theorems. Two general purpose functions, called METARULE and METATAC, are discussed which automate this correspondence.

5.1 An LCF axiomatisation package.

Several packages have appeared for automating the domain equation technique in LCF. Typically, they are decomposed into two component packages; the first package is used to generate an axiomatisation of the domain equation specified by the user. The

second package delivers an ML function which generates an appropriate structural induction tactic for types axiomatised using the first component. In addition, the second package also provides an ML function for generating case analysis tactics.

Robin Milner implemented the first suite of packages of this kind in 1980, which was briefly reported in a short unpublished memo. A description of the user interaction required to invoke Milner's packages can also be found in the Appendix of [CohnMilner82]. The package described in detail below is derived from Milner's, insofar as the user interaction required to invoke it is similar. However, the methods of calculation used internally are different, since Milner's packages could not easily be adapted to deal with both the smash and Cartesian products of domains simultaneously.

At about the same time as the packages described below were developed, Lawrence Paulson gave a similar suite for use within Cambridge LCF. This is also capable of using strict products, as well as types defined using simultaneous recursive domain equations, and also caters for the revised PPLAMBDA (See [Paulson83a], [Paulson83b], [Paulson83e]).

The description of the package given below states the interaction with the LCF system it requires, as well as a brief indication of the method used internally. An example is used as a vehicle for the description; the general case is then indicated with respect to this. As illustrated in Section 2.1.6, the least solutions of domain equations may be axiomatised by exploiting the existence of the least fixed point of a certain continuous functional, called the "copy" functional. The form of this functional is determined by the form of the domain equation. To illustrate the operation of the package, an example is now introduced.

Suppose that we want to axiomatise a (polymorphic) type operator, called T , of arity two, such that it satisfies the following domain equation:-

$$(\alpha, \beta)T \quad \equiv \quad (\text{dot})_1 + \alpha + ((\alpha, \beta)T \otimes (\beta \# (\alpha, \beta)T)_1)$$

This equation has three summands, each of which gives rise to a corresponding constructor constant or function. In practice, the user will have particular names in mind for these; so, suppose that they are called "Empty", "Unit", and "Node". Also, in practice, it is quite convenient to give constructor functions with a "curried" type rather than that obtained directly from the equation. Hence, the constructors are assumed to have the following types:-

```
Empty : ( $\alpha, \beta$ )T
Unit  :  $\alpha \rightarrow (\alpha, \beta)T$ 
Node  : ( $\alpha, \beta$ )T  $\rightarrow \beta \rightarrow (\alpha, \beta)T \rightarrow (\alpha, \beta)T$ 
```

The user will generally need various selectors and discriminator functions (especially if, later on, strict functions are being defined). So, we assume that the following discriminators are required:-

```
NullT  : ( $\alpha, \beta$ )T  $\rightarrow tr$ 
isUnit : ( $\alpha, \beta$ )T  $\rightarrow tr$ 
```

with the selectors:-

```
Tip    : ( $\alpha, \beta$ )T  $\rightarrow \alpha$ 
Left   : ( $\alpha, \beta$ )T  $\rightarrow (\alpha, \beta)T$ 
Data   : ( $\alpha, \beta$ )T  $\rightarrow \beta$ 
Right  : ( $\alpha, \beta$ )T  $\rightarrow (\alpha, \beta)T$ 
```

The discriminators NullT and isUnit are associated with the first two summands of the domain equation; the selector Tip is associated with the second summand, and the remaining selectors are associated with the three components of the third summand. The standard relationships between these analytic functions and the synthetic constructor functions follow, in a fairly systematic way, from the form of the domain equation that defines the type operator T.

We now describe how the appropriate axiomatisation can be generated using the package. First of all, the form of the domain equation has to be specified. This is done by binding certain ML identifiers to appropriate values, before the package is invoked. These identifiers are as follows:-

```
sty : type    - This passes in a "typical" (i.e. most
general) instance of the type to be defined. From this
information, the name and arity of the type operator can be
```

determined.

shape : (token # term) list - This passes in a list of tokens and "example" terms. The idea here is that each element of the list specifies a summand of the equation. The token given is the name of the constructor (either constant or function); the "example" term is used in several ways. Firstly, it specifies the type of the summand it corresponds to, and therefore the (curried) type of it's associated constructor. Secondly, the "example" term specifies the names of variables for each of the components, to be used by the induction package later on. To achieve this, each "example" term should have a sufficiently general form. Constructor constants are indicated by "UP()" (or by "()", which is translated to the first form). The constructor names have to be unique.

discshape : token list - This consists of a list of tokens that specify particular names for the discriminators. If insufficient names are given, then a default name is used (e.g. "isC" corresponding to constructor named "C"). Again, all discriminator names have to be unique.

selshape : token list - This consists of a list of tokens specifying particular names for selector functions corresponding to components of summands. The convention for associating names with components is that the *i*th selector name corresponds to the *i*th component encountered when reading the domain equation from left to right. If no particular name for a selector is required, then a standard default name will be generated instead. These default selectors are indicated by including a corresponding '_' character. The default name for a selector corresponding to the *j*th component within a summand (when reading from left to right) is "SELCj", where C is the name of the corresponding constructor. If there are fewer names than selectors then defaults are assumed for the remainder. All selector names must be unique.

To use the package for the above example, the following values are bound to these ML identifiers:-

```

sty          = ":( $\alpha$ , $\beta$ )T"

shape        = [ ('Empty', "()")
                  ; ('Unit', "a: $\alpha$ ")
                  ; ('Node', "((t1: $\wedge$ sty) @ UP((b: $\beta$ ) , (t2: $\wedge$ sty)))"
                  ]

discshape     = ['NullT', 'isUnit']

selshape      = ['Tip', 'Left', 'Data', 'Right']

```

A general class of domain equations that this package axiomatises

is now considered. Assume that the lhs of the domain equation has the form:-

$$(\alpha_1, \alpha_2, \dots, \alpha_n)D$$

for some collection of n distinct type variables, where $n \geq 0$. The rhs of the domain equation then consists of a finite list of summands as follows:-

$$T_1 + T_2 + \dots + T_m$$

for some $m \geq 1$ and where each of the type expressions T_i assume one of the following forms:-

$$\text{dot}_1$$

or

$$(F_1 \text{ pd}_1 F_2 \text{ pd}_2 \dots \text{pd}_k F_{k+1})_1$$

or

$$(F_1 \text{ pd}_1 F_2 \text{ pd}_2 \dots \text{pd}_k F_{k+1})$$

where $k \geq 0$ and each of the pd_i is either the Cartesian product operator, "#", or the smash product type operator, "@". In order to permit nested lifting and the arbitrary association of products, each of the F_i 's may assume one of the last two forms given above, or they have one of the forms:-

$$\alpha_j, \quad (\text{a type variable from the } \underline{\text{lhs}} \text{ of the domain equation})$$

or

$$(\alpha_1, \alpha_2, \dots, \alpha_n)D, \quad (\text{the type operator being defined})$$

or

$$G, \quad \text{a previously known monotype.}$$

5.1.1 Using the axiomatisation package.

The package itself is entirely implemented in ML, and consists of a master text which successively invokes 9 other ML files. Each of these represents a separate phase in the process of axiomatising the domain equation. If erroneous input is discovered in some way,

then an exception is generated, a short message stating the reason for the exception is displayed and processing is halted. We now explain the behaviour of the package with reference to the example given above.

The first stage introduces a number of auxiliary ML functions that have general use within the package. The second stage is concerned with checking the form of the domain equation presented to ensure that it conforms to the above pattern. The checks performed here are now enumerated:-

- The rhs of the domain equation is non-empty.
- Exactly n distinct type variables occur in the lhs of the domain equation, where n is the arity of the type operator being defined.
- All of the given names for constructors, discriminators and selectors are unique.
- The form of each "example" term conforms to the requirements stated above; that is, it is composed with product pairings and the "UP" function. Each component of the summand type is indicated by a variable in the "example" term. The names of all variables occurring in an "example" term have all trailing primes (e.g. "a'") removed. Since these variable names will be used when generating new induction variables (by re-priming), it could be confusing to the user if primes were already present. Also, "example" terms of the form "()", representing constant constructors, are mapped to "UP ()".
- No variable occurs more than once within each "example" term.
- No variable name is either 'ABS', 'REP' or 'FUN' (these are standard names used in formulating axioms) or a given constructor name.

Once all of these checks are successfully completed, the third stage can begin.

This stage is concerned with formalising the isomorphism relationship between the lhs and the rhs of the domain equation. The first step here is to calculate an explicit type expression, called the "representation" type, corresponding to the rhs of the domain equation. This is easily done by determining the type of the "example" terms in sequence and then using the disjoint sum

type operator, "+". Using the resulting type expression, the isomorphism pair can now be introduced; as follows:-

$$\text{absT} : (\text{dot}_1 + \alpha + ((\alpha, \beta)T \otimes (\beta \# (\alpha, \beta)T)_1)) \rightarrow (\alpha, \beta)T$$

and

$$\text{repT} : (\alpha, \beta)T \rightarrow (\text{dot}_1 + \alpha + ((\alpha, \beta)T \otimes (\beta \# (\alpha, \beta)T)_1))$$

The defining axioms for these functions explicitly state the isomorphism relationship between them. For this example, these axioms are:-

```
'absT'  ]- "VABS. absT(repT ABS) = ABS"
'repT'  ]- "VREP. repT(absT REP) = REP"
```

The fourth stage deals with the axiomatisation of constructors. This consists of calculating the curried type for each constructor, and then formulating the required defining axiom for it. The type expression for a given constructor is extracted by examining the types of variables that occur when reading it's corresponding "example" term from left to right. The lhs of the defining equation is then built by using these variables in that order. The rhs is then constructed by applying an appropriate sequence of summand injection functions to the example term, to give a term with the "representation" type. Finally, the "abstraction" function, "absT", is applied, giving a term whose type is the same as that being defined. In the case of the example above, this process gives:-

```
'Empty' ]- "Empty = absT(INL(UP ()))"
'Unit'   ]- "Va. Unit a = absT(INL(INR a))"
'Node'   ]- "Vt1 b t2. Node t1 b t2 =
              absT(INR(INR(t1 @ UP(b , t2))))"
```

The fifth stage tackles the axiomatisation of discriminator functions. First of all, standard default names for any discriminator names omitted from "discshape" are generated as necessary. In the above example, a name for the discriminator for the final summand is obtained:-

$$\text{isCons} : (\alpha, \beta)T \rightarrow \text{tr}$$

The type for each discriminator required is trivial to generate, as they always have the form " $\text{tr} \rightarrow \text{tr}$ ". The non-trivial task here is to generate an appropriate defining axiom for a particular discriminator. These discriminator functions are always defined for defined input (and undefined for undefined input); they are strict, "total" functions. However, to determine which summand a value belongs to, it is necessary to apply the (partial) summand selection functions, OUTF and OUTR, and the summand test, ISL. Hence, some care has to be exercised in the order in which these selection functions are applied. So, the technique used is to first check whether the value belongs to a preceding summand before testing if it belongs to the summand discriminated upon. The rhs of the defining equation generated for a discriminator function takes the form of a sequence of right-nested conditionals. The condition parts consist of a sequence of summand selectors followed by an application of ISL. Note that the "representation" function, repT, must be applied first to give a value with "representation" type. The axioms generated in the case of the running example above are:-

```
'NullT'    ]- "VREP. NullT REP = ISL(repT REP)"
'isUnit'   ]- "VREP. isUnit REP =
              ISL(repT REP) = FF | ISL(OUTR(repT REP))"

'isNode'   ]- "VREP. isCons REP =
              ISL(repT REP) = FF |
              ISL(OUTR(repT REP)) = FF | TT"
```

If there is only one summand in the domain equation then the corresponding discriminator is equivalent to the definedness function, ∂ .

The next phase of the axiomatisation process is concerned with the selector functions. The first step consists of determining the selectors required, from the presented form of the domain equation. This can be done by analysis of the form of each "example" term. If a "example" term is constant, (i.e. "UP()"), then no selector function is required. Otherwise, the "example" term contains variables and, for each occurrence of a variable, a selector function is required. The next step is to determine the form of

the rhs of the defining equation of each selector. This consists of taking an arbitrary abstract value, ABS, and first applying the function, repT, to get it's "representation". Next, apply a sequence of summand selection functions to obtain the appropriate summand. Finally, apply a sequence of product projections and "DOWN" function as necessary that reaches the "example" term variable corresponding to the selector function. The next step is to associate each selector function with a name as specified by "selshape", taking into account the need to generate default selector names as well. The lhs side of the defining equations is given in the form of an application of the appropriate selector function to argument ABS. The axioms for the selector functions in the running example are:-

```
'Tip'      ]- "VABS. Tip ABS = OUTL(OUTR(repT ABS))"
'Left'     ]- "VABS. Left ABS = P1(OUTR(OUTR(repT ABS)))"
'Data'     ]- "VABS. Data ABS =
               FST(DOWN(P2(OUTR(OUTR(repT ABS)))))"
'Right'    ]- "VABS. Right ABS =
               SND(DOWN(P2(OUTR(OUTR(repT ABS)))))"
```

The seventh stage is concerned with the axiomatisation of the "copy" functional. The type required for this functional has the form " $:(\text{!sty} \rightarrow \text{!sty}) \rightarrow \text{!sty} \rightarrow \text{!sty}$ " and this gives in the case of the above example:-

$$\text{copyT} : ((\alpha, \beta)\text{T} \rightarrow (\alpha, \beta)\text{T}) \rightarrow (\alpha, \beta)\text{T} \rightarrow (\alpha, \beta)\text{T}$$

The non-trivial step is to construct the appropriate defining equation for this functional. The lhs of this equation is given as a curried application of the copy functional to a function, named "FUN : $\text{!sty} \rightarrow \text{!sty}$ ", and then applied to ABS. The rhs of this defining equation has a richer structure, consisting of a right-nested sequence of conditionals. The condition part for each conditional consists of a discriminator applied to ABS, and the affirmative case consists of a constructor term whose arguments (if any) are formed by applying corresponding selectors to ABS. In addition, if the type of a constructor argument is identical to the type being defined, then the function "FUN" is applied to the result of selecting on that argument. For the running example

above, this gives:-

```
'copyT' ]- VFUN ABS.
      copyT FUN ABS =
        ((NullT ABS)  $\Rightarrow$  Empty |
         (isUnit ABS)  $\Rightarrow$  Unit(Tip ABS) |
         (isNode ABS)  $\Rightarrow$  (Node (FUN (Left ABS))
                               (Data ABS)
                               (FUN (Right ABS))) | 1)
```

Note that all of the discriminators are used, and that the final term on the rhs is the least element, 1. It is a consequence of the axiomatisation that this alternative cannot be reached, and this is, in turn, due to the validity of the appropriate case analysis schema.

Finally, the axiomatisation itself is completed by the addition of the appropriate "fixed point" axiom for the copy functional. For the running example above, this takes the form:-

```
'FIXT' ]- "VABS. FIX copyT ABS  $\equiv$  ABS"
```

The remaining two stages prove a number of useful standard results and facts, based upon this completed axiomatisation.

The next stage now derives some simple tautologies, for "bookkeeping" purposes. The next two theorems are proven in order to pass the names of constructors and selectors for the type, to the induction (and cases) packages depending upon them. The term structure of each theorem is chosen so that these names can be extracted straightforwardly. In each case, they are appropriate instances of the reflexivity of $\equiv$. For the above example:-

```
'constructT' ]- "(Empty  $\equiv$  Empty) &
                  (Va. Unit a  $\equiv$  Unit a) &
                  (Vt1 b t2. Node t1 b t2  $\equiv$  Node t1 b t2)"
```

and

```
'selectT' ]- "(VABS. Tip ABS  $\equiv$  Tip ABS) &
                (VABS. Left ABS  $\equiv$  Left ABS) &
                (VABS. Data ABS  $\equiv$  Data ABS) &
                (VABS. Right ABS  $\equiv$  Right ABS)"
```

The last theorem proven here is a simple instance of the axiom 'copyT' and an application of the FIX rule in conjunction with the

axiom 'FIXT', as follows:-

```
'coverT'  ]- "VABS.
              ((NullT ABS)  $\Rightarrow$  Empty |
               (isUnit ABS)  $\Rightarrow$  Unit(Tip ABS) |
               (isNode ABS)  $\Rightarrow$  (Node (Left ABS)
                                     (Data ABS)
                                     (Right ABS)) | 1) = ABS"
```

This simple tautology is used by the case analysis tactic that is generated by applying the package described in Section 5.2. It is proven here, once and for all, rather than repeatedly reprove it each time it is needed.

Note that the constructor and selector information could also be obtained by analysing the form of the lhs of the copy functional axiom. This was not done here, since the above method makes a direct and explicit statement of the information required and that it is simpler to extract the desired information from these theorems.

Finally, a collection of standard lemmas of general utility are proven. Their proofs mainly go through by simplification using the appropriate axioms. It turns out that the first four lemmas (concerning simple definedness and strictness properties of the isomorphisms) are crucial to showing any strictness and definedness properties of the basic constructors and selectors. In the example above, these are:-

```
'absTUU'    ]- "absT 1 = 1"
'repTUU'    ]- "repT 1 = 1"
'DEFabsT'   ]- "VREP.  $\partial$ (absT REP) =  $\partial$ (REP)"
'DEFrepT'   ]- "VABS.  $\partial$ (repT ABS) =  $\partial$ (ABS)"
```

Next, some simple lemmas concerning the definedness of constructors, and the strictness of selectors are proven by simplification with the definitions and the four lemmas stated above. This produces, for our example:-

```
'DEFEmpty'  ]- " $\partial$ (Empty) = TT"
'DEFUnit'   ]- "Va.  $\partial$ (Unit a) =  $\partial$ (a)"
'DEFNode'   ]- "Vt1 b t2.  $\partial$ (Node t1 b t2) =  $\partial$ (t1)"
'TipUU'     ]- "Tip 1 = 1"
'LeftUU'    ]- "Left 1 = 1"
'DataUU'    ]- "Data 1 = 1"
'RightUU'   ]- "Right 1 = 1"
```

Finally, the following lemma is proven, stating the strictness of the copy functional:-

```
'copyTUU'    ]- "VFUN. copyT FUN 1 = 1"
```

This completes the performance of the axiomatisation package.

5.1.2 Discussion of the axiomatisation package.

In Milner's original axiomatisation package, the copy functional was defined by case analysis i.e. by giving an axiom for each separate constructor function of the domain. However, all of the constructor functions would also have been non-strict in each argument since the smash product of domains was not a type operator standardly available within LCF at that time. In addition, because case analysis could be used without difficulty to axiomatise copy functionals, the selectors and discriminators for the domain were not required.

When smash product is taken into account, definition by case analysis must usually be predicated on the strictness of arguments to constructors. This is because the smash product pairing function is not injective. So, taking the example given above, instead of generating the single axiom 'copyT', the following 4 axioms would have sufficed:-

```
] - VFUN. copyT FUN 1 = 1
] - VFUN. copyT FUN Nil = Nil
] - VFUN a. copyT FUN (Unit(a)) = Unit(a)
] - VFUN t1 t2 b.
    ∂(t1) = TT ⊃
    copyT FUN (Node t1 b t2) = Node (FUN(t1)) (b) (FUN(t2))
```

The strictness assumption is needed on t_1 because the constructor function Node is strict in that argument.

It is more difficult to argue the correctness of such a conditional axiomatisation than to use a single defining equation involving selectors and discriminator functions. Of course, the definitions of these functions have to be determined from the form of the domain equation, but this is not hard.

More extensive definedness and strictness theorems could have been proven at the end of the package. Note that, in general, the definedness of multi-argument constructor functions will depend upon the definedness of appropriate pairing functions. For example, the definedness of the Cartesian pairing function can be expressed by using the parallel "or" function, paror: $tr \rightarrow tr \rightarrow tr$:-

$\vdash \text{"}\partial(a,b) = \partial(a) \text{ paror } \partial(b)\text{"}$

If the constant "paror" is not available from within the present theory structure, then the following laws can be used to characterise the definedness of Cartesian pairing instead:-

$\text{]- } \partial(a) = \text{TT} \supset \partial(a,b) = \text{TT}$
 $\text{]- } \partial(b) = \text{TT} \supset \partial(a,b) = \text{TT}$
 $\text{]- } \partial(1,1) = 1$

The definedness of the smash product pairing function is easily characterised by using the function and: $tr \rightarrow tr \rightarrow tr$:-

$\text{]- } \partial(a \otimes b) = \partial(a) \text{ and } \partial(b)$

Again, if the constant "and" is not available, then the following axioms will suffice:-

$\text{]- } \partial(a) = 1 \supset \partial(a \otimes b) = 1$
 $\text{]- } \partial(b) = 1 \supset \partial(a \otimes b) = 1$
 $\text{]- } \partial(a) = \text{TT} \ \& \ \partial(b) = \text{TT} \supset \partial(a \otimes b) = \text{TT}$

Both of the truthvalued functions paror and and are defined in Section 2.1.2.

5.2 The structural induction and case analysis package.

The package described here defines ML functions which, when applied to suitable data, generate basic tactics for performing structural induction and case analysis. The package was written in conjunction with the axiomatisation package discussed above.

Throughout all of the case studies, no attempt has been made to automate the determination of which terms are to be inducted upon. Instead, induction is always used tactically and is presented as a technique for eliminating quantified variables in a type-dependant way. Hence, the order of quantified variables specifies the order in which certain structural inductions could be performed. In this way, the user can exert considerable control over the course of a tactical proof.

Automated techniques for determining which goal variables could make good candidates for induction variables have been considered in [Aubin76] and [BoyerMoore79], for example. One basic approach to this is to analyse the pattern of function calls in function definitions to determine those parameters whose structure needs to be decomposed. Another aspect of automatically generated inductive proofs is that goals frequently need to be "generalised" (by weakening an hypothesis, for example) before an inductive proof could be successful; this is discussed in the above references.

5.2.1 Generating structural induction tactics.

The ML function for generating structural induction is called `INDSCH` : token $\rightarrow$ token $\rightarrow$ tactic. The first curried argument is the name of an LCF theory and the second curried argument is the name of a known type operator. It is assumed that the given type operator satisfies a domain equation, whose axiomatisation is recorded in the stated LCF theory. Moreover this specification has to have the form as generated using the above axiomatisation package.

So, in the case of the example used previously, assume that the type operator 'T' is defined in an LCF theory named 'TREES'; the following line of ML is used to invoke the generation of `TREETAC`, the basic induction tactic for the type operator T:-

```
let TREETAC = INDSCH 'TREES' 'T'
;;
```

The behaviour of this tactic is as follows:-

" $\forall t. F[1]$ "
ss
fml

TREETAC =====

subgoal1 subgoal2 subgoal3 subgoal4

where

subgoal1 =

" $F[1]$ "
ss
fml

subgoal2 =

" $F[\text{Empty}]$ "
ss
fml

subgoal3 =

" $\forall a'. F[\text{Unit}(a')]$ "
ss
fml

subgoal4 =

" $\forall t1' t2'. F[t1'] \ \& \ F[t2'] \supset \forall b'. F[\text{Node } t1' \ b' \ t2']$ "
ss
fml

where the formula $F[1]$ is admissible in the variable 1, and the variables a' , b' , $t1'$ and $t2'$ are chosen so that they do not freely occur within the assumption list or the original goal.

Note that each new variable introduced during induction has been quantified over. This is not strictly necessary, since each such variable has a freshly-chosen name. However, it is convenient to do so, because tactics applied later on in the tactical proof may have to know which variables have been introduced (see, for example, the definition of LINDTAC in Section 5.3.2 below). A consequence of this is that the induction hypotheses (e.g. in subgoal4), will have to be included in the appropriate subgoal formula. We now briefly describe the general form that any tactic so-generated takes.

Assume that the type operator of interest is named D and has arity $r \geq 0$. Furthermore, assume that it has been axiomatised in

terms of an appropriate domain equation using the package described previously. Let $C_1, C_2, \dots, C_n$ be the names of the constructors, where, for definiteness, $C_1, \dots, C_k$ stand for constants (if any) and let $C_{k+1}, \dots, C_n$ be constructor functions (if any). The generated structural induction tactic corresponding to the type operator D then behaves as follows. First of all, the input goal formula must have the form:-

$$\text{"}\forall v. F[v]\text{"}$$

where v is a variable of type " $(\tau_1, \tau_2, \dots, \tau_r)D$ ", for some types $\tau_1, \tau_2, \dots, \tau_r$; also, the formula $F[v]$ has to admit induction in the variable v . If this is so, the tactic produces a list of $(n+1)$ subgoals, $SG_0, SG_1, \dots, SG_n$. Each of these subgoals have the same simpset and assumption list as the original goal.

The first subgoal consists of the "undefined" case with goal formula $F[1]$. Each of the remaining subgoals corresponds to one of the constructors for the type operator D . The first k of these subgoals (i.e. constant constructors) have goal formulae of the form $F[C_i]$ where $1 \leq i \leq k$.

The remaining $(n-k)$ subgoals (i.e. non-constant constructors) have more complex goal formulae, which may or may not have induction hypotheses, depending upon the type of the corresponding constructor function. Assume that the constructor function, C_i , has the following (curried) type:-

$$\text{"}\lambda A_1 \rightarrow A_2 \rightarrow \dots \rightarrow A_j \rightarrow (\alpha_1, \alpha_2, \dots, \alpha_r)D\text{"}$$

for some type expressions A_g where $1 \leq g \leq j$. If none of the A_g 's are equal to the type of the result, then no induction hypotheses for this goal are to be included. In this case, the goal formula then has the form " $\forall v_1 v_2 \dots v_j. F[C_i v_1 v_2 \dots v_j]$ " where each (fresh) variable v_g has type A_g .

On the other hand, suppose that, for definiteness, each of the types $A_1, A_2, \dots, A_p$ are type expressions different from the result type, where $0 \leq p < j$, and that $A_{p+1}, \dots, A_j$ are each equal to it. In this case, induction hypotheses are generated for each variable corresponding to the last $(j-p)$ arguments for the

constructor function, C_i . Hence, the goal formula corresponding to such constructors, has the form:-

$$\begin{array}{c} \forall v_{p+1} \dots v_j. F[v_{p+1}] \& \dots \& F[v_j] \supset \\ \forall v_1 v_2 \dots v_p. F[C_i v_1 v_2 \dots v_j] \end{array}$$

As before, each (fresh) variable v_g has type A_g .

The process by which these tactics are generated, and how they, in turn operate, is now discussed. We do so in the specific case of generating the induction tactic TREETAC defined above.

The first stage of the process is to extract, once and for all, all of the basic axioms and lemmas that are required by TREETAC. The required axioms are:-

- 'absT', 'repT', the defining axioms for the isomorphism pair.
- 'copyT', the defining axiom for the copy functional.
- 'FIXT', the "fixed point" property for the copy functional.

Also required are the two facts:-

- 'constructT', a simple tautology containing the constructors.
- 'selectT', a simple tautology containing the selectors.

The isomorphism pair axioms are used to get the original expression of the type defined, in this case " $:(\alpha, \beta)T$ ", and the original expression for the rhs of the domain equation. This ensures the correct association of type variables between the lhs and the rhs of the domain equation. The copy functional axiom will be used to relate each of the separate cases of the structural induction to the single theorem needed for the "step" case needed to apply the Computational Induction rule. The "fixed point" axiom is used to convert the result of applying Computational Induction into a theorem corresponding to the original goal.

Note that the names of these facts can be systematically obtained from the given name of the type operator whose induction tactic is required.

As mentioned in Section 5.1.1, the facts 'constructT' and 'selectT' are used to pass on the names (and types) of the constructors and selectors. In addition, the form of the 'constructT' fact is such that the original variables that appeared in the "example" terms originally used to specify the domain

equation, can be easily determined. The names of these variables form the basis for generating fresh induction variables, by using the standard ML function, variant (see [LCF], p129).

From the data contained in these facts, the appropriate induction tactic can be generated. The internal processes of the tactic when applied to a suitable goal are now described.

First of all, the form of the goal is checked to conform to the pattern as stated above. This involves determining the variable to be inducted upon (i.e. by taking the first quantified variable), checking that the resulting formula satisfies the admits induction test, and making a variant of this formula, if the induction variable occurred freely within the assumption list of the goal. Next, fresh variables are made for each of the constructor terms, before building the list of subgoals required. This step also instantiates types of the constructors and selectors, to correspond with those in the stated goal. Those subgoals that are to have induction hypotheses are determined by analysis of the types of variables in the constructor terms. Finally, the list of subgoals is computed, by instantiating the goal formula with appropriate constructors, by quantification over fresh variables and inclusion of induction hypotheses as antecedents of goals.

The definition of the validation function is more complex and depends upon several quantities computed during the subgoal generation phase. This function initially checks that the given theorems achieve the subgoals computed above. This is tested syntactically using the ML predicate function aconvform (see [LCF], p129), for checking the equivalence of formulae, up to alpha conversion.

The task that the validation has to perform* is to take the given list of theorems (which match the previously generated subgoals) and obtain theorems that can be directly supplied to the Computational Induction rule, INDUCT (See [LCF], p117). The result of that application has then to be used to derive a theorem matching the original goal (up to alpha-conversion).

* The process of validation described here is based on that described in [LCF], p98 - 99. See also [Milner76], p33-34.

The underlying idea is to use **INDUCT** to prove a theorem, involving the copy functional, whose form is based upon the original goal. In addition, the input theorems to **INDUCT** will have to be explicitly derivable from the given list of theorems.

In order to describe this process, assume that the original goal had the form " $\forall x. F[x]$ ". Now, consider the formula " $\forall x. F[\text{FUN}' x]$ " where " $\text{FUN}' : !xty \rightarrow !xty$ " is a variable that does not freely occur in $F[x]$ and xty is the object language type of the variable " x ". Finally, let $G[\text{FUN}']$ abbreviate this formula.

The main task will be to prove a theorem whose conclusion takes the form " $!G[\text{copyT FUN}']$ " with the hypothesis that $G[\text{FUN}']$ holds. We now show how that can be derived from the given list of theorems, in the specific case of the induction tactic, **TREETAC**, and the type operator '**T**'.

For the purpose of the discussion below, assume that the list of theorems below have been given the following names:-

Uuthm	=	$\vdash "F[!]"$
Emptythm	=	$\vdash "F[\text{Empty}]"$
Unitthm	=	$\vdash "\forall a. F[\text{Unit } a]"$
Nodethm	=	$\vdash "\forall t1\ t2. F[t1] \ \&\ F[t2] \supset \forall b. F[\text{Node } t1\ b\ t2]"$

The idea is to use the the above theorems, and the (derived) inference rule **CONDCASES** (see below, and [LCF], p 97) to build up a theorem whose conclusion consists of $G[\text{condtm}]$, where " condtm " is the rhs of the defining equation for **copyT**, suitably instantiated.

The inference rule **CONDCASES** behaves as follows:-

	$\text{ctm} = \text{TT} \quad \vdash w[!tm/x]$
	$\text{ctm} = \text{FP} \quad \vdash w[rtm/x]$
	$\text{ctm} = ! \quad \vdash w["!"/x]$
CONDCASES $x\ w\ t$	$\vdash w[t/x]$

where the term t has the shape " $!ctm \Rightarrow !ltm \mid !rtm$ ", the term x is an object language variable of the appropriate type, and " w " is a formula.

So, continuing with the derivation, take theorem **Nodethm** and specialise " $t1$ " and " $t2$ " with " $\text{FUN}'(\text{Left } x)$ " and " $\text{FUN}'(\text{Right } x)$ "

respectively, to give:-

$$\vdash "F[\text{FUN}'(\text{Left } x)] \ \& \ F[\text{FUN}'(\text{Right } x)] \ \supset \\ \forall b. F[\text{Node} (\text{FUN}'(\text{Left } x)) \ b \ (\text{FUN}'(\text{Right } x))]"$$

Now, the two antecedents above are instances of the formula
 $G[\text{FUN}'] = "\forall x. F[\text{FUN}' x]"$. So, by assuming this formula, and
 then specialising it appropriately we get:-

$$. \vdash F[\text{FUN}'(\text{Left } x)]$$

and

$$. \vdash F[\text{FUN}'(\text{Right } x)]$$

Applying MP twice with the above to the previous theorem produces:-

$$. \vdash "\forall b. F[\text{Node} (\text{FUN}'(\text{Left } x)) \ b \ (\text{FUN}'(\text{Right } x))]"$$

Note that there is only one hypothesis here (up to alpha-
 conversion). The next step is to specialise "b" with "Data x" to
 give:-

$$. \vdash F[\text{Node} (\text{FUN}'(\text{Left } x)) \ (\text{Data } x) \ (\text{FUN}'(\text{Right } x))]$$

This is in the right form to apply CONDCASES, where the conditional
 term is:-

$$"(\text{isNode } x) \Rightarrow \text{Node} (\text{FUN}'(\text{Left } x)) \ (\text{Data } x) \ (\text{FUN}'(\text{Right } x)) \mid \perp"$$

and the other two theorems are both UUthm. This gives the
 theorem:-

$$. \vdash F[(\text{isNode } x) \Rightarrow \\ \text{Node} (\text{FUN}'(\text{Left } x)) \ (\text{Data } x) \ (\text{FUN}'(\text{Right } x)) \mid \perp]$$

Now, by using the theorem Unitthm, specialising "a" to "Tip x", we
 get:-

$$\vdash F[\text{Unit}(\text{Tip } x)]$$

So, applying CONDCASES again, with suitable conditional term and
 the last two theorems (plus UUthm again), we have that:-

```
.⊢ F[ (isUnit x) ⇒ Unit(Tip x) |
      (isNode x) ⇒
        Node (FUN'(Left x)) (Data x) (FUN'(Right x)) | ⊥ ]
```

Finally, by applying CONDCASES a third time with the above theorem, the Emptythm and the U0thm, we reach:-

```
.⊢ F[ (isEmpty x) ⇒ Empty |
      (isUnit x) ⇒ Unit(Tip x) |
      (isNode x) ⇒
        Node (FUN'(Left x)) (Data x) (FUN'(Right x)) | ⊥ ]
```

As stated above, we can now use the defining equation for copyT to give, by substitution:-

```
.⊢ F[ (copyT FUN') x ]
```

Now, because the variable "x" does not occur freely in the hypotheses (since fresh variables were introduced), we can generalise upon "x" to get:-

```
.⊢ "∀x. F[ copyT FUN' x ]"
```

Also, we may take U0thm:-

```
⊢ F[ ⊥ ]
```

and obtain directly:-

```
⊢ "∀x. F[ (⊥ : !xty → !xty) x ]"
```

using the basic property of least functions, and generalising upon "x". Finally, by applying the INDUCT rule to the last two theorems generated, we have that:-

```
⊢ "∀x. F[ FIX copyT x ]"
```

Note that the INDUCT rule eliminates the induction hypothesis that "∀x. F[FUN' x]". Clearly, by specialising "x" again, and then, by using the "fixed point" property for the copy functional, 'FIXT', and substituting, we have that:-

```
⊢ F[x]
```

Finally, this can be generalised on "x" to give:-

```
⊢ "∀x. F[x]"
```

This completes our discussion of the induction tactic TREETAC, and the structural induction tactic generator, INDSCH.

5.2.2 Generating case analysis tactics.

In a similar fashion to the above, the package also defines an ML function, called CASESCH :token → token → tactic, for generating type-specific case analysis tactics. This generator is invoked in much the same way as for INDSCH. For example, the following brief ML text produces TREECASETAC, a case analysis tactic corresponding to the type operator 'T':-

```
let TREECASESTAC = CASESCH 'TREE' 'T'
;;
```

The functional behaviour of the tactic is given by:-

"Vt. F[t]"
ss
fml

TREECASESTAC

 [subgoal1; subgoal2; subgoal3; subgoal4]
where

subgoal1 =	F[1]
	ss
	fml

subgoal2 =	F[Empty]
	ss
	fml

subgoal3 =	"Va'. F[Unit(a')]"
	ss
	fml

subgoal4 =	"Vt1' t2' b'. F[Node (t1') (b') (t2')]"
	ss
	fml

The general form of the case analysis tactic generated is as for the structural induction tactics, except that no induction hypotheses are generated in the subgoals. In addition, the admissibility constraints upon input goal formulae are unnecessary, since generated case analysis tactics do not make any use of the

INDUCT rule.

We now briefly discuss the operation of CASESCH, by considering how the case analysis tactic TREECASETAC declared above, is generated. When CASESCH is invoked, the facts 'constructT', 'selectT' and 'coverT' are recalled from the LCF theory 'TREE'. The first two facts are used in the same way as for TREETAC; they pass on data concerning the constructors and selectors. The third fact, 'coverT', is used by the validation function for TREECASETAC.

The process for generating subgoals is as for TREETAC, except that no induction hypotheses need to be generated. As before, fresh variables are introduced as necessary, with appropriate quantifiers.

The validation function has to map a list of theorems, each corresponding to a separate case, into a theorem matching the original goal. The process used makes use of the structure of the copy functional's definition in a similar manner to that for TREETAC. The inference rule CONDCASES is used, as before, to take each case's theorem and combine them, to eventually give a theorem which matches the lhs of 'coverT', suitably instantiated. By substituting for this expression, using 'coverT', and then generalising, we obtain a theorem achieving the original goal.

5.4 Special induction tactics.

During the course of the above case studies, various special induction tactics were introduced. These were not defined directly by using the induction package described above, but required substantially different development. The first to be described is the (family of) Monoid induction tactics that were applied in Chapter 3. The other tactic described here is the list induction tactic, LINDTAC, used throughout Chapter 4.

5.4.1 Monoid Induction.

In the case studies in Chapter 3, it was shown how the Monoid Induction principle can be usefully applied within proofs concerning domains of lists. In Section 3.1, this principle was described and also shown to be valid by a reduction to the standard

structural induction principle for lists.

The proof of validity forms the basis for programming the basic Monoid Induction tactic named MONOIDTAC; the behaviour of this tactic was described in Section 3.2.3. This tactic formed the kernel of a derived tactic named MONTAC that was then directly used in the case studies within in Chapter 3.

The method used here is to first apply the schematically generated structural induction tactic for the appropriate list domain and then apply a tactic to the resulting Cons case which further decomposes that into the Unit and Append cases. The "I" and Nil cases are left unchanged.

A consequence of the method of derivation is that it can be easily used for any list domain within which the "ConsUnit" lemma can be proven (see Lemma 3.14, for example). A direct expression of this induction principle using Computational Induction avoiding a reduction to ordinary list induction is feasible in principle, but has not been attempted here.

We now turn to the expression of the tactic itself within ML, which is given in Figure 5.1 below. From the above discussion, the detailed form of the tactic depends upon the particular list theory being used. In the presentation below, the necessary details from a typical example of such a theory are made explicit (the general situation being analogous). Although the case for a polymorphic type operator is shown here, the more specific monomorphic case goes through in the same way.

Assume that the list type operator being used is named 'List' and that it is axiomatised (by some appropriate domain equation) in an LCF theory named 'L'. Suppose also that the "ConsUnit" theorem is available in that theory as:-

'ConsUnit']- $\forall a:\alpha \text{ l}:(\alpha)\text{List}. (a :: \text{l}) = \text{Unit}(a) @ \text{l}$

From the form of this theorem, the name and types of the operators 'Cons', 'Unit' and 'Append' can be determined. The ML text for extracting this information from the above theorem is routine and has not been included below. Note that this theorem is also used within the validation part of the tactic.

The details of two auxilliary ML functions used in the


```

mlc infix 'Then' % The identifier "Then" is declared as an infix %
;;

let (f Then g) = (g o f)
and PM thm impthm = MP impthm thm
;;

let alfa = ":α"
and Unittm = "Unit: α -> (α)List"
and Apptm = "$@: (α)List # (α)List -> (α)List"
and LISTTAC = INDSCH 'L' 'List'
and thConsUnit = FACT 'L' 'ConsUnit'
;;

let MONOIDTAC = (LISTTAC THENL [ IDTAC; IDTAC; CONSTAC ])

where CONSTAC (consfm, ss, fml) =
  let (l, impfm) = destquant consfm           in
  let (lfm, afm) = destimp impfm              in
  let a = fst(destquant afm)                  in
  and goalfrees = formlfrees(lfm.fml)         in
  let aty = typeof a                          in
  let insttm tm = instinterm [(aty, alfa)] tm  in
  and INSTTH thm = INSTTYPE [(aty, alfa)] thm in
  let Utm = insttm Unittm                     in
  and Atm = insttm Apptm                      in
  let subtm tm = substinform [(tm, l)] lfm     in
  and vary tm = varytm(tm, goalfrees)         in
  let l1 = vary(suffixvar '1' l)              in
  and l2 = vary(suffixvar '2' l)              in
  let fmlst =
    [ "∀ !a. !(subtm " !Utm !a)"
      ; "∀ !l1 !l2.
          !(subtm l1) & !(subtm l2) IMP !(subtm " !Atm(!l1,!l2))"
    ]
  in
    (map (λ fm. (fm,ss,fml)) fmlst), CONSRUL

where CONSRUL [Unitthm; Appthm] =
  ( (SPEC " !Utm !a" )          Then (SPEC l)          Then

    (PM (SPEC a Unitthm)) Then (PM (ASSUME lfm)) Then

    (SUBST
      [ ((SYM o (SPEC l) o (SPEC a) o INSTTH) thConsUnit, l) ]
      lfm
    ) Then

    (GEN a) Then (DISCH lfm) Then (GEN l)
  ) Appthm
;;

```

FIGURE 5.1

definition below have been omitted for clarity. They are varytm: term # term list $\rightarrow$ term, which takes a term and a list of variables, and makes variants of any free variables from the given term that appear in the given list. The other function is called suffixvar: token # term $\rightarrow$ term and this adds the given suffix to the name of the given variable, after stripping off any trailing primes. The purpose of these two functions is to ensure that variants of variables are made in a uniform and systematic way.

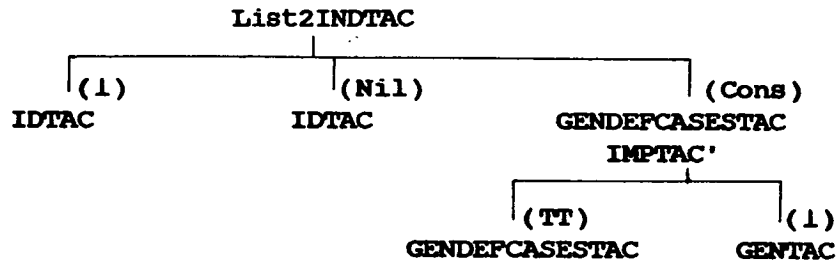
The tactic first applies list induction, LISTTAC, and then applies the tactic CONSTAC in the "Cons" case. This tactic examines the structure of the Cons case goal formula, extracting types, quantified variables and a formula equivalent to the original goal. This information is used to instantiate types of the standard Unit and Append terms, Unitm and Apptm, as well type instantiate the 'ConsUnit' theorem during the validation stage. Two fresh induction variables are generated which are made disjoint from each other by first using suffixvar and then from the other free variables occurring in the goal by using varytm. The list of two subgoals is routinely generated by using anti-quotation and then adding the simpset and assumption list components.

The validation component maps a pair of theorems corresponding to the Unit and Append cases of the monoid induction into a theorem corresponding to the appropriate Cons case for list induction and does so by formalising the proof of Lemma 3.3.

5.3.2 LINDTAC.

The list domain tactic, LINDTAC, was introduced and described in Section 4.5.3. The definition of LINDTAC' in ML is now described; it makes use of a standard, schematically generated induction tactic named List2INDTAC^{*} here. The first task to be done is to calculate the required subgoal decomposition and applies the following composite tactic to the goal:-

* The tactic List2INDTAC is defined in Section 3.4.2.



producing 5 subgoals in all, three of which are logically equivalent to the "1" case. The first two and the last subgoals then form the resulting subgoal list for LINDTAC'. The validation part for LINDTAC' uses the validation part generated above and involves constructing the two theorems corresponding to the two omitted subgoals. This, in turn, requires the form of the original goal and the specific induction variables introduced by the basic induction tactic List2INDTAC. This may all be programmed in ML, as shown in Figure 5.2 below.

5.3 Resolution oriented tactics.

Various tactics for engaging resolution oriented theorem proving techniques are discussed in detail below. Firstly, a discussion of the original resolution tactic developed by Avra Cohn, called RESTAC, is given. This is followed by a more detailed description of the resolution tactics used in the two case studies presented in Section 4.7. Finally, some ideas concerning some "back-chaining" forms of resolution tactics are discussed.

5.3.1 The original RESTAC.

An early version of this tactic was described in [Cohn81] which was later used in the parser correctness case studies given in [CohnMilner82] and [Cohn82]. Lawrence Paulson has also independently developed a version of RESTAC (see [Paulson83d]).

The resolution oriented tactics, of which Cohn's RESTAC is an example, are unlike most other tactics discussed here since the goal formula itself is not affected. Such tactics act, instead, upon the list of assumptions already available in the goal to derive logical consequences from them. In some sense, all that resolution tactics do is to find different ways of applying Modus

```

let thUUCons = FACT 'L' 'UUCons'
and thConsUU = FACT 'L' 'ConsUU'
;;

let LINDTAC' g =
  let (x,w) = (destquant o fst) g
  in
  let ([ UUCase; NilCase; ConsCase1 ], prf1) = List2INDTAC(g)
  in
  let ([ ConsCase2; (); () ], prf2) =
    (I THEN GENDEPCASESTAC THEN IMPTAC'
     THENL [ GENDEPCASESTAC; GENTAC ]
     ) ([ ConsCase1 ], hd)
  in
  let (lvar, stepfm) = (destquant o fst) ConsCase1
  in
  let avar = (fst o destquant o snd o destimp) stepfm
  in
  let defaUU = (DEFUU o ASSUME) "a(lavar) = 1"
  and deflUU = (DEFUU o ASSUME) "a(lvar) = 1"
  in
  let aUuthm = (SYM o snd o (simpterm (ssadd defaUU ssCons)))
    "Cons lavar lvar"
  and lUuthm = (SYM o snd o (simpterm (ssadd deflUU ssCons)))
    "Cons lavar lvar"
  in
  ([ UUCase; NilCase; ConsCase2 ], prf3)

where prf3 [ Uuthm; NilThm; ConsThm ] =
  let UuthmaUU = SUBST [aUuthm, x] w Uuthm
  and UuthmlUU = SUBST [lUuthm, x] w Uuthm
  in
  let ConsThml = prf2 [ConsThm; UuthmaUU; UuthmlUU]
  in
  prf1 [Uuthm; NilThm; ConsThml]
;;

```

FIGURE 5.2

Ponens between assumptions.

The behaviour of Cohn's RESTAC is now described. Each formula in the list of available assumptions is assumed and a derivation process (see below) is applied between every possible pairing of implicative formula with non implicative formula. If any new consequences are obtained by this means, then their conclusions are added to the list of assumptions. In addition, any consequences that have a certain carefully specified syntactic form may be added as *simprules* to the local *simpset*. This class, for example, rejects "obviously" looping *simprules* such as instances of

reflexivity and such like. If no consequences can be derived at all, then the tactic fails (to allow iteration using REPEAT). The validation component for RESTAC is explained below.

The derivation process is now described. Each implicative formula in the assumption list is canonicalised, as necessary, by applying the following inference rule as much as possible:-

$$\frac{\vdash \forall x_1 x_2 \dots x_n. w_1 \supset (\forall y. w_2)}{\vdash \forall x_1 x_2 \dots x_n y'. w_1 \supset w_2[Y'/Y]}$$

(where y' is chosen to be disjoint from $\{x_1, x_2, \dots, x_n\}$).

This is applied in order to bring as many universally quantified variables into the prefix as possible. (A slightly more general canonicalisation is used in CANONTAC described in Section 4.7.1; the inference rule HNF used there is described in Section 5.4.6).

Having done this, it may be assumed that all of the implicative formulae have the form:-

$$\forall v_1 v_2 \dots v_k. w_1 \supset w_2$$

where the conclusion formula, w_2 , is not a quantified formula, and because of formula identification (see [LCF], p72), it is also not an implication.

The next step is to try to find matches of the antecedent(s) of the implicative assumptions from among the already available assumptions. By the above, the implicative assumptions will have been assumed and canonicalised and have the form:-

$$\vdash \forall v_1 v_2 \dots v_k. (a_1 \& a_2 \& \dots \& a_n) \supset w$$

Now, suppose that there is an available assumption, w_3 , such that there exists an antecedent formula, a_i , and a (most general)* substitution $[t_1/v_{i1}, t_2/v_{i2}, \dots, t_h/v_{ih}]$ such that the formula:-

$$a_i[t_1/v_{i1}, t_2/v_{i2}, \dots, t_h/v_{ih}]$$

is (α -convertible) to the assumption, w_3 . It is assumed that $\{v_{i1}, v_{i2}, \dots, v_{ih}\} \subseteq \{v_1, v_2, \dots, v_k\}$ and has cardinality h .

* Substitutions may be quasi-ordered by composition (See [Buet80]).

Putting $\{x_1, x_2, \dots, x_j\} = \{v_1, \dots, v_k\} \setminus \{v_{i1}, \dots, v_{ih}\}$, where $j = k - h$, we may then derive, by specialisation, Modus Ponens and then generalisation that:-

$$\dots \vdash \forall x_1 x_2 \dots x_j. ((a_1 \& \dots a_{i-1} \& a_{i+1} \dots \& a_n) \supset w)[t_1/v_{i1}, \dots, t_h/v_{ih}]$$

For example, suppose that the given goal contains the following two assumptions:-

$$\begin{aligned} \forall x_1 x_2. F(x_1) \equiv x_2 \quad \& \quad G(x_2) \equiv F(z) \supset x_2 \equiv F(G(z)) \\ G(F(q)) \equiv F(z) \end{aligned}$$

where F and G are some suitable functional constants. By applying the matching process described above, the following result would be obtained:-

$$\dots \vdash \forall x_1. F(x_1) \equiv F(q) \supset F(q) \equiv F(G(z))$$

The variable " x_2 " is matched to the term " $F(q)$ ", and this matching is used to specialise the theorem formed by assuming the implicative assumption. The second antecedent is then eliminated by applying a slightly generalised form of Modus Ponens (using the assumed equation). Finally, any previously quantified variables not involved in the match are then generalised upon. The two hypotheses of the derived result are the original assumptions from which it was derived.

Note that the pattern matching method used is simple one-way matching in which only one of the formulae is regarded as a template and some of whose free variables may be instantiated to achieve a match with the other formula. In the above, the antecedent formulae of implicative assumptions are used as templates, with those variables that are explicitly quantified, at the outermost level, giving the instantiable variables.

The validation component for RESTAC has to "undo" the effect of the derivation of "new" consequences from "old" assumptions. The situation is sketched in Figure 5.3 below. The task that is performed is the replacement of any hypotheses of the (single) input theorem that are alpha-equivalent to derived consequences, by those assumptions from which these consequences were derived. This

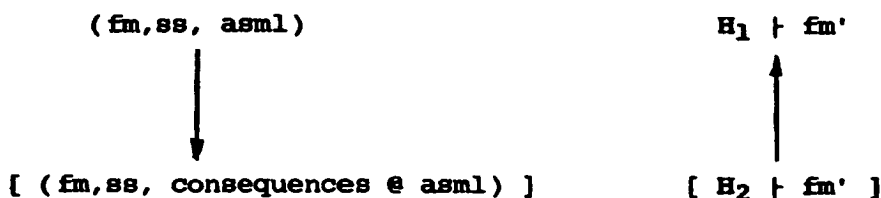


FIGURE 5.3

does not affect the conclusion formula of the input theorem.

Hence, any formulae contained in $(consequences \cap H_2)$ are eliminated in favour of those assumptions in $asml$ from which they were derived. So, we have the relationship that $H_1 \subseteq asml$, modulo alpha-equivalence.

5.3.2 Limitations of Cohn's RESTAC.

The matching technique used in Cohn's RESTAC is, as mentioned above, one-way matching. This is well-known to be incomplete; as a very simple example of this, consider the following:-

$$\begin{array}{l}
 \forall x y. x + 0 = y \supset x = y \\
 \forall m n. m + n = n + m
 \end{array}$$

From this, we should be able to deduce that:-

$$\forall x. x = 0 + x$$

However, to do so, both formulae have to be matched and specialised simultaneously. This kind of matching is known as unification and was first reported in Section 5 of [Robinson65] in connection with formulating the Resolution Principle. Other kinds of unification are now known that take into account certain properties possessed by particular functions, such as associativity and commutativity (see [HuetOppen80], [Plotkin72] and [Stickel81]).

Note that in the case studies given in Section 4.7, unification matching was needed to obtain several consequences (e.g. consequences C1.8, C2.1 and C2.11).

Not using unification is clearly a limitation in that not every possible match can be computed. But note the following:- if the proof can be found by only using one-way matching, then fewer extraneous consequences are necessarily developed, giving a more

tightly constrained search. Unification matching, being complete in some sense, develops all possible matches for all possible proofs from the available assumptions and not just the particular one that is being considered at the moment.

It is noted that Edinburgh LCF does not directly provide a unifying pattern matcher for PPLAMBDA constructs; instead, only one-way matching is given via the two ML functions formmatch and termmatch (see [LCF], p132). Efficient algorithms for performing unification have only been found comparatively recently (see [Paterson78], [Martelli82], [CorbinBidoit83], for example).

In addition, the symmetry of equivalence, $\equiv$, is not taken into account when matching equational antecedents; suppose that within the illustrative example used previously in Section 5.3.1 above, we had only the equation " $F(z) \equiv G(F(q))$ " instead of " $G(F(q)) \equiv F(z)$ " then matching and derivation could not have taken place.

Also, antecedents are eliminated one at a time; hence, implicative assumptions with two or more antecedents available would require more than one full iteration of Cohn's RESTAC for them to be completely discharged. In addition, each different way of eliminating an antecedent gives rise to separate consequences. Again, using the illustrative example above, suppose that we have that:-

$$\begin{aligned} \forall x_1 x_2. F(x_1) &\equiv x_2 \quad \& \quad G(x_2) \equiv F(z) \quad \supset \quad x_2 \equiv F(G(z)) \\ G(F(q)) &\equiv F(z) \\ F(G(r)) &\equiv F(q) \end{aligned}$$

then Cohn's RESTAC would derive the following two implicative formulae:-

$$\begin{aligned} \forall x_1. F(x_1) &\equiv F(q) \quad \supset \quad F(q) \equiv F(G(z)) \\ G(F(q)) &\equiv F(z) \quad \supset \quad F(q) \equiv F(G(z)) \end{aligned}$$

Another full iteration of Cohn's RESTAC is then required to produce the equation:-

$$F(q) \equiv F(G(z))$$

However, this would also then be derived twice, once from each of the above formulae. In computing the final list of consequences to

be placed in the assumption list, any repetitions, (up to alpha equivalence), are reduced.

Finally, when iterating Cohn's RESTAC, all consequences from previous iterations are recomputed at each iteration.

5.3.3 LINEARRESTAC and a (new) RESTAC.

A re-development of Cohn's RESTAC was undertaken which attempted to remedy some of the limitations mentioned above. A general tactic, LINEARRESTAC1, was developed for performing the derivation process. The two tactics LINEARRESTAC and a (new) RESTAC were then given in terms of this one. The main points are as follows:-

Unification matching (at 1st order) for PPLAMBDA data types type, term and form was provided in ML (see below).

The symmetry of equivalence is recognised by trying both possible ways of matching two equations together and returning any results found.

As in Cohn's RESTAC, non-implicative assumptions are matched against the antecedents of implicative assumptions, using the "Linear, Input" strategy for "eagerly" eliminating as many available antecedents as possible in one attempt. This process is iterated internally (using any derived implicants) until no further consequences can, or need, be derived. Only the equational consequences derived during these iterations are returned for addition to the assumption list of the goal.

Each derived consequence is checked if it establishes the goal in some way. If it does, then the derivation process is immediately halted.

In LINEARRESTAC, a collection of implicative lemmas is input against which all resolutions may be performed.

Some additional "features" were incorporated, such as the application of simplification (with a specified simpset) to resolvents (i.e. the products of derivation). For RESTAC, the local simpset is used; for LINEARRESTAC, the given simpset is used. Also, the list of intermediate results produced during internal iterations are sorted, using the "quicksort" algorithm by formula size, to produce shorter consequences before longer ones.

By performing internal iterations of the derivation process, more equational consequences can be derived within a single application

of a resolution tactic. This reduces the frequency of "loosing one's place" within the derivation and having to re-establish it by repeating derivations. Because more equational conclusions are drawn, the need to iterate these tactics is reduced; however, when they do need to be iterated, previously derived consequences will still be inferred over again, as happened with Cohn's RESTAC.

The functional behaviour of the two tactics RESTAC and LINEARRESTAC is now described in broad terms. This is then followed by a detailed presentation of the implementation of these tactics within ML.

fm
ss
asml

RESTAC

fm
ss
(conseq @ asml)

where conseq is the formula list formed from the conclusions of the consequences newly derived from the assumptions. The derivation process uses the assumption list asml to provide the implicative formulae for resolution. The local simpset, ss, is used to simplify resolvents (i.e. the products of derivation).

fm
ss
asml

LINEARRESTAC (ss1,thml)

fm
ss
(conseq @ asml)

where conseq is the formula list formed from the conclusions of the consequences newly derived from the assumptions. The derivation process uses the conclusions of the theorems in thml to provide the implicative formulae for resolution. The given simpset, ss1, is used to simplify resolvents.

Neither of the tactics above add any derived consequences to the local simpset at any stage; this may be catered for by using tactics like FINIXGNDSSSTAC (see Section 4.7.1).

The implementation of these resolution tactics was given within ML in a structured and modular way. This structure exposes the strategy used to control the pattern of search, so that modifications to this strategy could be easily made.

The presentations below consists of ML text interspersed with discursive passages, and to aid readability, a top-down sequence of definitions is employed. However, various ML function definitions are not given formally when they would distract attention from the central thread of the presentation; in any case, these may all be found in Appendix 4 and the role played by such functions is described informally in the discussion.

In Figure 5.4 below, the two tactics RESTAC and LINEARRESTAC are defined in terms of LINEARRESTAC1. For RESTAC, each of the implicative formulae in the assumption list are assumed, and passed on. For LINEARRESTAC, the given list of theorems are first canonicalised using the ML function HNF (discussed in Section 5.4.6). Next, the hypotheses of these theorems are checked to ensure that each is alpha-convertible to assumptions already available within the goal. This task is done using the ML function CheckThml, and uses the ML predicate aconvform. The ML function openq strips away leading quantifiers from formulae.

The tactic LINEARRESTAC is now defined in Figure 5.5 below, to be a composition of several different functions. The tactic part of this definition is provided by GENDERIVETAC (see later), which

```

let RESTAC (fm,ss,asml) =
  let impthml =
    mapfilter
      (\fm1. (isimp o openq) fm1 ⇒ ASSUME fm1 | fail)
      asml
  in
    LINEARRESTAC1 (ss,impthml) (fm,ss,asml)

and LINEARRESTAC (ss1,thml) =
  let thml' = flat(map HNF thml)
  in
    λ g. (CheckThml thml' g) ⇒ LINEARRESTAC1 (ss1,thml') g
      | failwith 'LINEARRESTAC'
;;

```

FIGURE 5.4

```

let simpfilter ss thm =
  let thm' = SIMP ss thm in
    (istruth o concl) thm' ⇒ fail | thm'
;;

let (GenDERIVETAC inferfn);tactic (fm,ss,asml) =
  let (ss', thl') = inferfn (ss,asml) in
    (null thl') ⇒ failwith 'GenDERIVETAC' |
      ( [ (fm, ss', (map concl thl') @ asml) ]
        , {DERIVE thl'} o hd
        )
  ;;

let LINEARRESTAC1 (ss1, thm1) (fm,ss2,asml) =
  GenDERIVETAC
    (InferFn
      (StepFn
        ( ResLoop (UnionFn, (GoalChk fm), (simpfilter ss1))
          , quicksort (Thequiv, ThOrd)
          )
        )
      thm1
    )
  (fm,ss2,asml)
;;

```

FIGURE 5.5

takes a functional parameter that defines the derivation process to be performed. In this case, the ML functions Inferfn, Stepfn, and ResLoop are used to represent the derivation process. The function Unionfn performs the set-theoretic union of two lists of theorems, taking alpha conversion into account; the function GoalChk is used to check freshly derived consequences to see if they could establish the goal (either by being a standard contradiction or by being alpha equivalent to the goal itself). The ML function quicksort is parameterised by an equivalence function and an ordering function, which should be compatible. In this instance, the equivalence function is Thequiv, which takes into account alpha conversion and the symmetry of PPLAMBDA equivalence, $\equiv$; the ordering function, ThOrd, simply compares the textual size of the conclusion formulae of the given pair of theorems.

The ML function simpfilter simplifies the given theorem with the given simpset, and if it's conclusion formula is equal to "TRUTH" then simpfilter fails. Otherwise, the simplified theorem

is returned.

The parameterised tactic GenDERIVETAC applies the derivation function specified by parameter "inferfn", to the goal's simpset and assumption list. This returns a simpset and a list of theorems representing the consequences. In the derivation function actually used in LINEARRESTAC, the simpset returned is the same as that passed in. The derivation function deliberately does not depend upon the goal formula; this prevents the goal from being inadvertently included as an hypothesis of a consequence. The validation part applies the ML function DERIVE to the consequences and the (single) input theorem (see Figure 5.6 below).

The functional behaviour of the ML function DERIVE as an inference rule is now described:-

$$\text{DERIVE thml} \frac{\{fm_1, fm_2, \dots, fm_n\} \cup H \vdash w}{H_1 \cup H_2 \cup \dots \cup H_n \cup H \vdash w}$$

whenever the list of theorems, thml, contains theorems of the form $H_i \vdash fm_i'$ where fm_i' is alpha-convertible to the hypothesis fm_i . The set of hypotheses, H , contains the remaining hypotheses of the given theorem.

The algorithm used to implement this rule is to find all those theorems in thml whose conclusions are alpha-convertible to some hypothesis of the given theorem, and then repeatedly apply the discharge rule, DISCH, and Modus Ponens, MP, with this list and the given theorem.

The ML predicate function isatomic recognises basic equational or inequational formulae.

The ML function InferFn first assumes all the (possibly quantified) atomic assumptions in the given formula list, fml, to form the theorem list, Lthl. This forms the first component of the input to the derivation process. The second component is the given list of (implicative) theorems, Ithl, and the third component is a truth-valued "flag" (initially false) stating whether the derivation should halt.

This data is passed into the derivation process, which simply iterates the given parameter function, stepfn, until the predicate

```

let DERIVE th1 th =
  let hyp1 = hyp th in
  let disch1 = filter
    (λth. exists (λfm'. aconvform(fm, fm')) hyp1
     where fm = concl th
    )
    th1
  in
    itlist DERIVESTEP disch1 th

where DERIVESTEP alteration th =
  MP (DISCH (concl alteration) th) alteration
;;

let InferFn stepfn lth1 (ss, fml) =
  let lth1 = mapfilter
    (λfm. (isatomic o openq) fm ⇒ ASSUME fm | fail)
    fml
  in
    ( ss
    , ( (filter (NewThmChk lth1))
      o fst
      o (Until nomore stepfn)
      ) (lth1, lth1, false)
    )
  where
    nomore (lth1, rth1, done) = (null lth1) or (null rth1) or done
;;

```

FIGURE 5.6

nomore becomes true. This predicate simply checks whether either of the first two lists become empty or if the third component (named done) is true. The ML functional Until iterates the given functional body over the given argument, until the given predicate function gives true of the result.

Next, the first component of the resulting triple is then passed on to the final stage, which simply filters out theorems already present in the list Lth1, by using the ML function NewThmChk. This checks for theorems that are equivalent to those in Lth1, taking into account alpha-equivalence as well as the symmetry of equality for (quantified) equational theorems. Note that the simpset component returned by InferFn is the same as the given simpset parameter.

The ML function divide given in Figure 5.7 below uses the given predicate function, pred, to divide the given list, l, into those elements that satisfy pred and those that do not.

```

let divide pred l = itlist arbitrator l ([],[])
  where arbitrator x (inlst,outlst) =
    pred(x) => (x.inlst,outlst) | (inlst,x.outlst)
;;

let StepFn (resloop, oracle) (Lthl, Rthl, done) =
  let (newthml, done) = resloop (Lthl, Rthl) in
  let (newLthl, newRthl) =
    (divide (isatomic o openq o concl)) o oracle) (newthml @ Lthl)
  in
  (newLthl, newRthl, done)
;;

let Guard fn (x,tv) = tv => (x, true) | fn(x)
;;

let ResLoop (unionfn, satchk, simpfn) (Lthl, Rthl) =
  revitlist
    (\lth.
      revitlist
        (\rth. Guard (InnerResLoop lth rth))
        Rthl
      )
    Lthl
    ([], false)

where InnerResLoop lth rth resl =
  let infer rthl = mapfilter simpfn (RESOLVE lth rthl) in
  let AddFn rthl =
    let reslst = infer rthl in
    Guard
      (\resl'.
        (unionfn(reslst, resl'), exists satchk reslst)
      )
  in
  itlist AddFn (rth . resl) (resl, false)
;;

```

FIGURE 5.7

The ML function StepFn applies the given function resloop to the theorem lists Lthl and Rthl to produce a list of consequences, newthml, and a truth-value to say whether the task had been achieved during the application of resloop. Next, the new theorems, newthml, and Lthl are supplied to an "oracle" function. In the particular application, this simply sorts the theorems using an appropriate measure. The result of this process is then divided into (possibly quantified) atomic or non-atomic theorems which then form newLthl and newRthl respectively.

Note that only Lth1, and not Rth1, is included for later processing. This is vital to prevent re-computation of already derived consequences from any previous internal iterations. In addition, it assures the termination of the derivation process, as follows.

All of the consequences produced using resolution result from eliminating at least one antecedent of an implicative theorem. Hence any theorem so produced has at least one less antecedent than the theorem from which it was derived using resolution. Therefore, the number of antecedents each theorem in newRth1 has is strictly less than the maximum number of antecedents of any implicative theorem from Rth1. Since the test in InferFn checks when either list of theorems produced is empty, this shows termination.

The ML function Guard extends the given parameter function, fn : $\alpha \rightarrow (\alpha \# \text{tv})$ to a function with type $:(\alpha \# \text{tv}) \rightarrow (\alpha \# \text{tv})$. This returns its argument if the second component is true; otherwise, it applies fn to the first component.

The ML function ResLoop is used to apply resolution between each pair of theorems in Lth1 and Rth1, as well as any intermediate results. This function Guard is used in inhibiting further computation of derivations once the "flag" shows that the task has been achieved.

In more detail, the result of applying ResLoop is a pair consisting of a list of derived theorems and a truth-value. The algorithm starts off initially with the empty list and false, and progressively adds derived theorems to the list.

Essentially, each theorem in Lth1 is paired up with each theorem in Rth1 for the purpose of generating consequences. In addition, the ML function InnerResLoop ensures that any consequences already obtained during this process are compared with the present choice from Lth1. This, potentially, allows several antecedents of an implicative theorem to be eliminated within a single iteration. The order in which antecedents are eliminated can be significant if instantiable variables have several occurrences in separate antecedents.

At the centre of the algorithm, the ML function infer is used

within the function AddFn to generate new consequences. This in turn uses the function RESOLVE which is applied to attempt to resolve left-hand theorems against the antecedents (if any) of right-hand theorems. The given parameter function simplfn is applied to any consequences, or resolvents, obtained and the list of successful results from this process is returned.

The function AddFn firstly uses the function infer to obtain a result list, reslst. This is then added to the accumulated list of consequences, using the given parameter function unionfn. In addition, a check is made to see if the task has been achieved using the given parameter function satchk.

5.4.4 The function RESOLVE.

The ML function RESOLVE :thm $\rightarrow$ thm $\rightarrow$ thm list, is briefly discussed below; its definition in ML may be found in Appendix 4. The value of RESOLVE (lth) (rth) is the list of resolvents obtained by attempting to unify lth with any antecedent of rth, taking symmetry of PPLAMBDA equivalence into account when lth is a (possibly quantified) equation. A list of theorems is returned since more than one antecedent of rth may be successfully matched by lth.

The basic matching process is described below; suppose that the second argument, rth, is an implicative theorem of the form:-

$$\vdash \forall v_1 v_2 \dots v_n. (a_1 \& \dots \& a_k) \supset w$$

and suppose that the conclusion of the first argument, lth, has the form:-

$$\forall u_1 u_2 \dots u_m. fm$$

To simplify the discussion, assume that the two sets of quantified variables, $V = \{v_1, \dots, v_n\}$ and $U = \{u_1, \dots, u_m\}$ are disjoint. This requirement is easily enforced by renaming as necessary. Observe that the notation "fm[ψ]" is used to mean "apply the substitution ψ to formula fm". Finally, it is assumed that unit substitutions (i.e. those of the form " v/v ") are omitted.

The theorem lth resolves with the theorem rth when there

exists an antecedent a_i of rth which unifies with (a specialisation of) lth.

More precisely, this means that there exists a (most general) substitution $\theta = [t_1/x_1, \dots, t_j/x_j]$ such that the set, $X = \{x_1, \dots, x_j\} \subseteq U \cup V$, and such that the formula:-

$$a_i[\theta_V]$$

is alpha-equivalent to the formula:-

$$\forall y_1 \dots y_r. \text{fm}[\theta_U]$$

where θ_U is equal to the restriction of substitution θ (as a function on variables) to the set U , and similarly for θ_V . Also, the set $\{y_1, \dots, y_r\} = U \setminus X$, for some $r \geq 0$. The order of the quantified variables $y_1, \dots, y_r$ is chosen to match leading quantifiers in a_i , if possible.

Each resolution gives rise to a resolvent of the following form by specialisation, Modus Ponens and then generalisation:-

$$\vdash \forall z_1 \dots z_s. ((a_1 \& \dots a_{i-1} \& a_{i+1} \dots \& a_n) \supset w)[\theta_V]$$

where the set $\{z_1, \dots, z_s\} = V \setminus X$, where $s \geq 0$.

When lth is a (quantified) equation, the symmetry of PPLAMBDA equivalence is dealt with by matching the equation in each orientation, and returning any results obtained.

As a simple illustrative example, suppose that rth is:-

$$\vdash \forall x y. (\forall p. x + p \equiv p + 0) \supset x + y \equiv y$$

and that lth is:-

$$\vdash \forall m n. m + n \equiv n + m$$

The single antecedent of rth is unified with lth by substitution $[0/x, 0/m]$ and produces the resolvent:-

$$\vdash \forall y. 0 + y \equiv y$$

5.4.5 Unification.

In order to implement the RESOLVE function described above, it was necessary to implement, in ML, unification functions for the PPLAMBDA data types term and form. (Unification for type was also implemented, but not required here.)

The unification algorithm used is based upon a refinement of Robinson's original algorithm (see [Robinson65] or [Robinson79]). The design of the algorithm was influenced by the use of sequential, or monic, substitutions in the efficient pointer-oriented algorithm developed by Paterson and Wegman (see [Paterson78]). By carefully representing monic substitutions as lists of term/variable pairs whose elements are kept in a certain order, it is possible to avoid substituting partial unifiers into terms and avoiding the potential exponential size increase of sub-terms during the unification process.

Once the unifier has been determined, it is necessary to convert the monic substitution into a corresponding simultaneous substitution, as used by the PPLAMBDA inference rules SUBST and INST. This could, of course, lead to the exponential increase in size of substitutions that is avoided by using monic substitutions in computing the unifier. Example 3 of [Paterson78] illustrates this increase in size.

In unifying PPLAMBDA formulae, no attempt was made to take into account properties of the logical connectives such as commutativity and associativity of conjunction.

5.4.6 Canonicalisation.

The function $\text{HNF: thm} \rightarrow \text{thm list}$ is a derived inference rule that is used to put implicative theorems into a canonical form before applying resolution tactics; the functional behaviour of HNF was informally referred to in Section 4.7.1. HNF can be defined in ML as shown in Figure 5.8. Various auxiliary functions are used whose definition can be found in Appendix 4.

The algorithm recursively decomposes the theorem over the syntactic structure of its conclusion formula. The derived inference rule $\text{DESTQUANTL : thm} \rightarrow (\text{term list} \# \text{thm})$ ensures that suitable care is taken when stripping quantifiers (by specialising them) to use variables not already free in the hypotheses.

```

let HNF th = map (uncurry GENL) (H [] th)

whererec
  H ql th =
    if (isquant fm)
    then
      (let (ql', th') = DESTQUANTL (th)
       in
        H (ql @ ql') th'
      )
    else if (isconj fm)
    then flat(map (H ql) (DESTCONJL (th)))
    else if (isimp fm)
    (let afml = (destconjl o fst o destimp o concl) th
     in
      let thl = revitlist PM (map ASSUME afml) th'
      in
        map
          (\(ql', th'). (ql', itlist DISCH afml th'))
          (H ql thl)
      )
    else
      [ (ql, th) ]

where fm = concl th
//

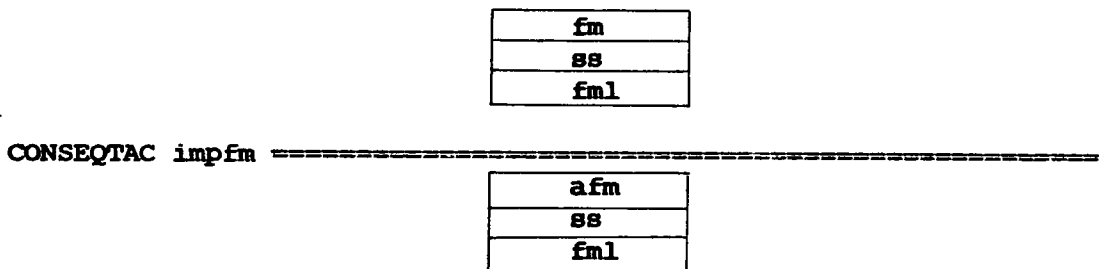
```

FIGURE 5.8

5.4.7 Backchaining forms of resolution tactics.

The tactics presented here are related to Cohn's USEIMPASSUMPTAC (See [Cohn81]), which scanned the assumption list for implicative assumptions whose consequent could be matched to the goal, by instantiating quantified variables appropriately. The replacement of the goal by the (instantiated) antecedent is justified by an application of Modus Ponens.

In Figure 5.9 below, three similar parameterised tactics are presented in terms of ML code. The first such tactic, called CONSEQTAC :form $\rightarrow$ tactic, and behaves as follows:-



(where impfm has the form " $\forall v_1 \dots v_k. afm' \supset fm'$ " and there exists an instantiation of the variables $v_1 v_2 \dots v_k$ such that fm' will match fm and afm is the result of instantiating the antecedent afm' . The symmetry of equality is also taken into account when matching the goal formula (assuming that fm is atomic). Otherwise failure.)

Note that if there are no quantified variables in $impfm$ then the consequent of $impfm$ must match the goal exactly for success, modulo alpha-conversion. Clearly, CONSEQTAC can then give Cohn's USEIMPASSUMPTAC, by simply iterating this over the assumption list of the goal, as follows:-

```
let USEIMPASSUMPTAC (w,ss,fml) =
  tryfind.(λfm. CONSEQTAC fm (w,ss,fml)) fml
;;
```

The remaining two parameterised tactics, EVALTAC :tactic $\rightarrow$ tactic and PROGTAC :thm list $\rightarrow$ tactic $\rightarrow$ tactic, perform computations which resemble the evaluation of a set of logical clauses as a Prolog program (see [Clocksin81]). So, considering EVALTAC, the idea is to first apply CONSEQTAC in turn to each of the present goal's assumptions. If any succeed then the tactic parameter, T, is

```

let (CONSEQTEC fm):tactic =
  let (qv, fm') = destquantl fm      in
  let (afm, cfm) = destimp fm'      in
  let afmfv = formfrees afm
  and cfmfv = formfrees cfm
  in
  (($not o null o intersect) (subtract(afmfv,cfmfv),qv)
  => failwith 'CONSEQTAC: Hidden Variables in Match Formula'
  | (let CEQTAC (w,ss,asml) =
      (let match = filter ($not o eq) (formmatch cfm w) in
       (forall (λ(tm,v). (mem v qv)) match)
       => (let th1 = ((INST match) o OPENQ o ASSUME) fm in
          let (afm',cfm') = (destimp o concl) th1 in
            (aconvform(w,cfm'))
            => ([afm',ss,asml]), (MP th1) o hd)
          | fail
        )
       | fail
      ) ? failwith 'CONSEQTAC'
    in
      (CEQTAC ORELSE (SYMTAC THEN CEQTAC))
  )
;;

letrec EVALTAC T (w,ss,fml) =
  tryfind
    (λfm. ((CONSEQTAC fm) THEN T THEN (EVALTAC T)) (w,ss,fml))
    fml
  ;;

let PROGTAC thml T g =
  (let fml = map concl thml in
   (CheckThml thml g) => (progtac g) | fail
  )
where rec progtac g =
  tryfind (λfm. ((CONSEQTAC fm) THEN T THEN progtac) g) fml
  ;;

```

FIGURE 5.9

applied. Finally, any result obtained is passed onto (EVALTAC T).

Because of the recursive application of (EVALTAC T) itself, the tactic T is interleaved with applications of CONSEQTAC, and as failures occur, remaining possibilities are tried. Note that the tactic T could modify both goal formula and the assumption list over which CONSEQTAC is iterated. For this tactic to terminate with a definite result, the empty goal list must eventually be generated by an application of the tactic T to an intermediate goal.

The tactic PROGTAC is similar to EVALTAC, except that a fixed

collection of theorems are given to provide the alternatives over which CONSEQTAC is iterated. The ML function CheckThm1 ensures that the hypotheses of the given set of theorems lie among the formulae in the assumption list.

5.5 METARULE and METATAC.

In this section, two useful ML functionals are briefly presented which can be used to convert a wide class of (quantified) PPLAMBDA theorems either into a tactic or an inference rule. The idea is that the theorem provides the justification for either forwards or backwards inference by use of Modus Ponens and instantiation. In order to do this, polymorphic matching of forms is necessary to construct appropriate instantiations of the given theorem. The ML descriptions of both of these functions is given in Appendix 4.

The ML function METARULE :thm $\rightarrow$ thm list $\rightarrow$ thm, is a parameterised inference rule generator which takes a (quantified) implicative PPLAMBDA sentence and produces the derived Natural Deduction style inference rule corresponding to the theorem. For example, consider the following sentence:-

]- $\forall x y : \alpha. x \in y \ \& \ y \in x \supset x = y$

The partial application of METARULE to the above produces a function of type :thm list $\rightarrow$ thm, representing the following rule:-

[$\vdash "tm_1 : \tau \in tm_2"$; $\vdash "tm_2 \in tm_1"$] $\Rightarrow$ $\vdash "tm_1 = tm_2"$

This rule is similar to the standard SYNTH rule for the partial ordering. The produced function fails if not given the right number of input formulae, or if any of the formulae are not of the expected shape. Note that, for example, the particular term tm_1 is polymorphically matched to the variable "x" with type ": α ". The function METARULE checks, before constructing the inference rule, that there are no free term or type variables in the conclusion which are not matched by occurrences in the antecedents.

A limitation is that schematic families of inference rules dependent upon formula parameters (e.g. such as GEN, the

generalisation inference rule) cannot be represented this way since PPLAMBDA does not possess object language variables ranging over properties.

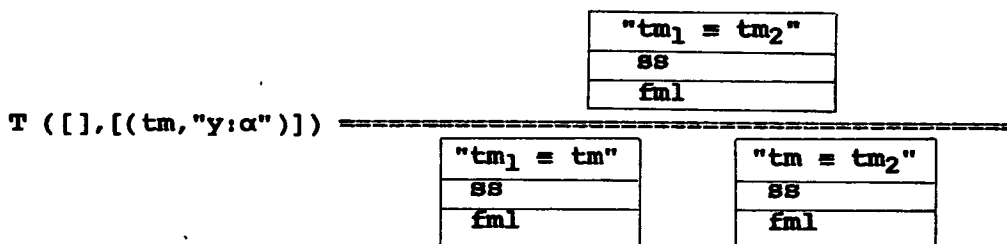
The ML functional METATAC is used to generate a tactic corresponding to a wide class of PPLAMBDA sentences and has the ML type $\text{thm} \rightarrow ((\text{type} \# \text{type})\text{list} \# (\text{term} \# \text{term})\text{list}) \rightarrow \text{tactic}$. The idea is to take a sentence and "invert" it, to give a tactic mapping goals which correspond to the conclusion into subgoals composed from the antecedents of the given theorem.

To do this, a polymorphic matching of the goal to the conclusion of the given theorem is attempted. Any free term or type variables matched in this way are then instantiated in the theorem. However, many theorems require a slightly more general approach because the antecedents may contain occurrences of free term and type variables that do not occur in the conclusion. In such cases, a pair of type and term instantiations can be used to give these variables particular assignments.

For example, consider the general polymorphic theorem stating transitivity of equality:-

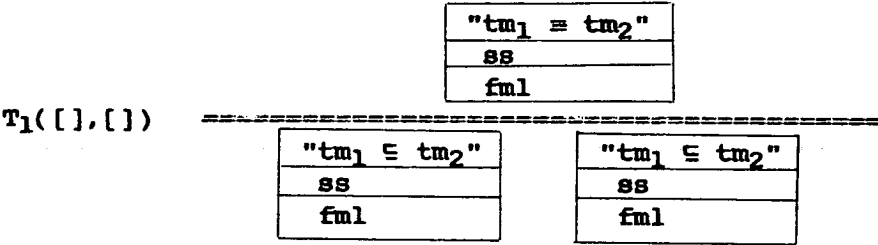
$\text{]- } \forall x \ y \ z : \alpha. \ x = y \ \& \ y = z \supset x = z$

Note that the term variable "y:α" occurs free in the antecedents but not in the conclusion. A partial application of METATAC to this theorem produces a parameterised tactic (called T below) which is dependent upon instantiations. When it comes to apply T as a tactic, a term instantiation is given specifying a value for the variable "y". Hence, the behaviour of T may be given by:-



Of course, if all the term and type variables occurring free in the antecedents of the theorem also occur in the conclusion, then the polymorphic matching process will determine all of the appropriate

information for the subgoals. Hence, taking the previous theorem above, the behaviour of it's corresponding tactic (called T_1 below) is:-



One-way polymorphic matching is used in both of the above functionals. This notion is defined as follows. Suppose that ptm is a pattern term containing either free term and type variables to be matched, and that mtm is the term to be matched against. We say that mtm polymorphically matches ptm if and only if there exists a pair of simultaneous instantiations for types and terms, $(tyinst, tminst)$ such that the following holds:-

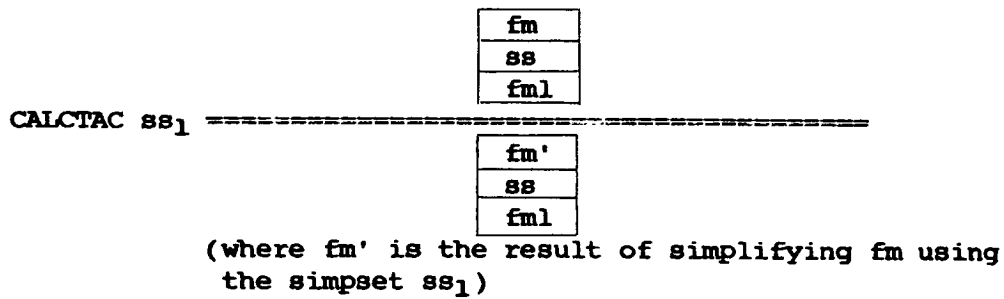
$$mtm = substinterm\ tminst\ (instinterm\ tyinst\ ptm)$$

where the equality is taken to mean syntactic identity. This definition extends gracefully to the formula case. Various ML functions were written for calculating such matchings; these in general can take the context into account, in the form of a list of bound variables and given term and type instantiations.

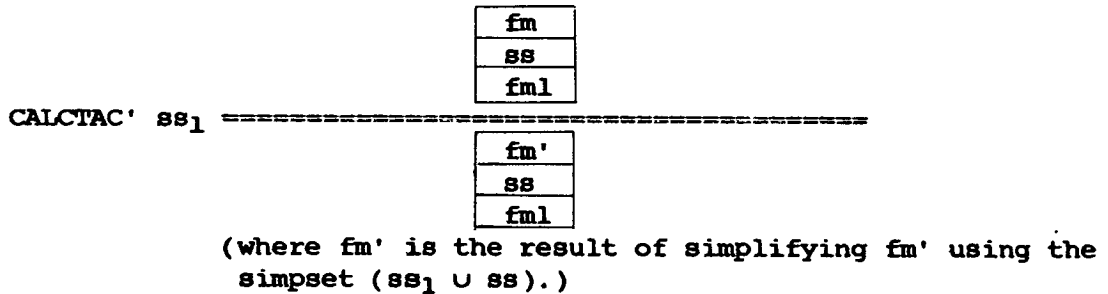
5.6 Simplification tactics.

It is occasionally useful to engage a specific, non-local simpset during a tactical proof. It may happen that particular simpsets are not confluent when extra assumptions are added or that a collection of simprules conspire not to terminate when applied to certain goals. In such cases, the following parameterised tactics can be used.

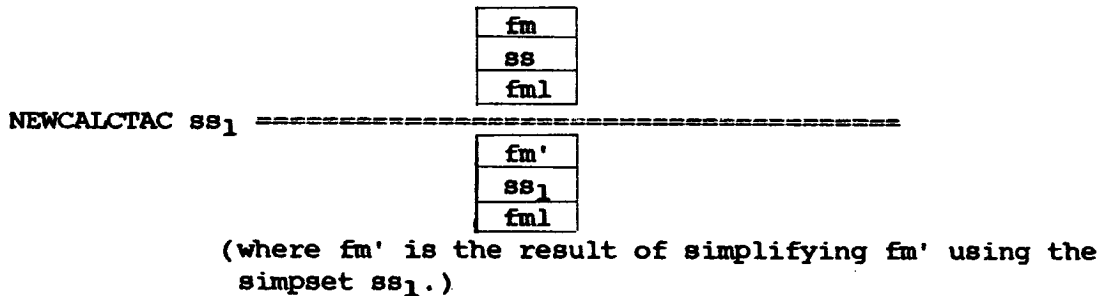
The first is called `CALCTAC :simpset → tactic`, and this applies the given simpset to the goal, ignoring the goal's local simpset. It's behaviour can be defined as follows:-



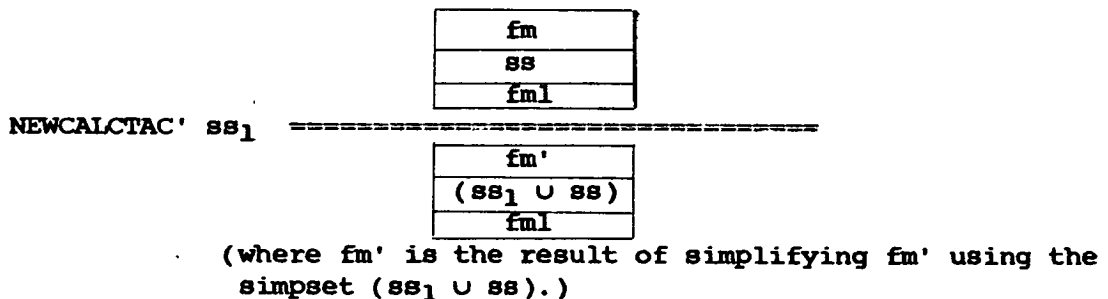
The next tactic, CALCTAC', acts in a similar way except that the union of the local and given simpsets is used to simplify the goal.



The above two tactics do not change the simpset component of the goal. This facility is provided by the following two tactics:-



The above tactic simplifies the goal formula using the given simpset, which also replaces the previous local simpset.



The above tactic simplifies the goal with the union of the given and local simpset, which then replaces the previous local simpset.

Conclusions.

Edinburgh LCF is a second generation proof assistant of a highly programmable kind. By using strong typing, it is possible to just store the result of doing a proof without the overhead of storing the proof itself. The separation of object language from meta language give deduction rules (written in ML) the freedom to analyse linguistic objects and perform calculations about them in a logically secure way. The tactical methodology enables the user to formulate high-level, goal directed strategies for the construction of proofs. Domain specific reasoning is easily accommodated within this framework, using induction, case analysis and simplification tactics. The higher type capability and the exception mechanism of ML fundamentally contributes to the successful application of this methodology.

One general motivation for doing case studies using LCF is to gain insight into the pragmatic issues surrounding the generation and manipulation of complex, linguistic objects when using a functional programming style. At the heart of the LCF philosophy is the idea that functional parameterisation and abstraction is a powerful technique for describing highly complex forms and relationships. Another goal is to study the ease with which fully justified mathematical reasoning can be simulated or generated by using a programmable theorem proving system such as Edinburgh LCF.

LCF case studies are usually concerned with proving some well-constrained collection of facts about a (structured) collection of functions defined axiomatically. Hence, LCF does not directly address the question of general software validation as it stands. Instead, such validation problems have to be cast, or formalised, in PPLAMBDA terms and the corresponding correctness statements verified. Hence, LCF is more suited for addressing the correctness of algorithms, rather than the correctness of particular programs in some specific programming language.

Several interesting issues are raised by such attempts to generate and construct formal proofs. One major aspect concerns the "global" structure of the theories used to decompose, or

factorise, the problem into smaller, more manageable parts. This then leads to considering how this global structure can be effectively used to simplify deduction within and between theories. Another aspect concerns the "micro-structure" of the deductive system used and how well it can be applied to express certain valid arguments, or patterns of inference. This issue is more concerned with the form that deductions could take, rather than with which statements can be proven in some arbitrary way.

The Case studies.

The aim of the case studies was to formulate and prove the correctness of some simple equationally specified data types using LCF. Such results can be thought of as relative consistency proofs which explicitly show the existence of models of the particular data type.

The three case studies given in Chapter 3 considered the problem of showing that the standard algebra of lists with concatenation operation and empty list form a free (left-strict) monoid with respect to the singleton, or Unit, list function. The different case studies arise by varying the underlying domain of lists used. This variation changes the strictness properties of the Cons function in a natural way, and gave rise to domains of either infinite or finite lists.

The formalisation of the freeness criterion in PPLAMBDA borrows an idea of Burstall and Landin's (see [BurstallLandin69]) which uses a second-order, polymorphic functional giving the extension morphism for any suitable valuation of generators. The "second-order" aspect allows parameterisation by the given operations of a monoid and the "polymorphism" aspect allows parameterisation by the carrier of a monoid.

The studies showed that the statement of a freeness criterion could be established in each case, with possibly some additional strictness assumptions about the given monoid's binary operation and the valuation function. Moreover, by using the tactic GENDEFCASESTAC, in conjunction with the appropriate structural induction tactic, the introduction of strictness properties for

variables could be systematically achieved (see also [Fisher84]), The third case study, which dealt with the (polymorphic) domain of finite lists, needed definedness assumptions the most. This used a bi-strict Cons function that gives rise to definedness pre-conditions when applying either of the selection functions, Head and Tail.

Two tactical proofs of the Uniqueness theorem were given in each case study; the first used structural induction over the list domain, and the second used "Monoid Induction" with respect to the derived concatenation function. The tactic for Monoid Induction systematically made use of the list induction using a uniform method across each of the case studies.

The Multiset case study presented within Chapter 4 formulated the correctness of a simulation of an algebra of multisets. This simulation used a flat domain of lists, quotiented by an effectively given equivalence function, called EqBA. In some sense, this function represents a tiny theorem prover for deciding whether two given lists represent the same multiset.

An interesting aspect of the multiset case study is that the multiset equivalence function makes use of a function, EqAT, which is merely assumed to be an equivalence predicate on the domain of elements. Many of the lemmas proven do not depend upon the full strength of this assumption; often only strictness is required which, in turn, is derived from the assumption characterising definedness (see Lemma 2.7). The theory EQFUN contains some general theorems about functions assumed to be equivalence predicates, and these were specialised to give theorems about EqAT (using the appropriate assumptions). Of course, these general theorems could also be specialised for EqBA itself, once all the appropriate correctness criteria have been established for discharging the hypotheses.

Note also that the explicit mentioning of an equivalence predicate easily permits these theorems to be instantiated for the case of multisets of multisets, and so on.

The correctness criteria that were proven in LCF showed that the given algorithmically specified function represented an

equivalence predicate, in the sense of Section 2.4. Also required were the congruence properties for the concatenation and Unit functions and, finally, that the multiset equations held (i.e. the simulation is valid). This final requirement principally consists of showing that the concatenation function is commutative, with respect to the multiset equivalence predicate.

The programming approach taken here is somewhat similar in philosophy to the work of Robert Cartwright as presented in [Cartwright82]. In this approach, a concept of "domain construction" and "data domain" is also introduced and requires explicitly defined equality predicates to characterise equivalences that are specified logically. Cartwright also points out that the given predicate must represent an equivalence relation on the representation domain, and furthermore, that it is "extensional". This presumably means the equivalence has to be a congruence, with respect to some particular collection of operators.

The proofs of the lemmas were all conducted by various combinations of structural induction over lists, Boolean case analysis, definedness case analysis, simplification and substitution. A major complication was the amount of explicit reasoning required to deal with the equivalence assumptions concerning EqAT. This was, in general, not performed tactically, but by short stretches of forwards deduction. This frequently consisted of specialising a property of EqAt (e.g. 'GenTrans') for particular variables (possibly introduced automatically by induction), and then simplifying the result, before making use of it in the main proof.

Various automated techniques are known for determining certain properties of congruences (see [Shostak78], [NelsonOppen80], for example). However, such methods seem to be restricted to first order, quantifier free, theories with uninterpreted function symbols.

Both informal and formal (i.e. tactical) proofs were given in considerable detail, in order to compare the kind of reasoning that was invoked. In general, the overall pattern of inductions and case analyses corresponded closely, but this is to be expected

since the detailed development of proofs in either style affected the other.

The freeness criterion for multisets was not established formally in PPLAMBDA, and a separate proof was given, making use of a standard set theoretic model construction (see Section 4.8). The heart of the matter is whether or not the specification of multisets is consistent (i.e. whether it has a model). The simulation given in the case study given shows only that some commutative monoid is represented upon lists and leaves the question of which commutative monoid unanswered. Hence, a denotational model is constructed making essential use of the simulation given and the freeness property of the underlying (flat) domain of lists, proven in Chapter 3. It is an interesting question to see how naturally this reasoning could be embedded into PPLAMBDA by erecting some kind of formal theory of (arbitrary) set-like objects.

Two case studies were performed in which resolution-oriented tactics were applied to obtain theorems already proven within the multiset case study (see Section 4.7). In general, various domain-specific tactics (e.g. structural induction and case analyses) are applied to enrich the assumption list of the remaining subgoals first of all. Next, some combination of resolution tactics are applied to "finish off" these subgoals, by purely logical inference. The resolution tactics proceed by performing a "breadth first" search that resolves equational assumptions against the antecedents of specified implicative theorems.

As is well-known, the resolution process is expensive in terms of machine resources. However, case studies such as the above show that the tactical framework can be used to incorporate tactical agents that work successfully on the assumptions rather than the goal (see also [Schmidt83a]). This suggests that the assumptions might be considered as a "dual" kind of goal information which represents presently known, or assumed information about the requirement represented by the goal formula. Such a stance is more respectful of the Natural Deduction style of proof adopted in Edinburgh LCF. A promising direction would be to find useful

classes of proof operators which use this broader concept of goal.

PPLAMBDA issues.

PPLAMBDA is a small calculus that is designed to give a formal counterpart of programming language theory in the mathematical framework of denotational semantics. Within the natural applications of this theory, it is mainly partial functions that are encountered (e.g compiler and interpreter correctness).

However, for the case studies considered above, nearly all of the functions considered were total and strict (i.e the result is defined for defined input, and otherwise undefined). In general, the undefined case of inductions or case analyses of propositions using such functions would usually follow by simplification using explicit strictness theorems. However, there does not seem to be a general, efficient way to incorporate this "meta" information into LCF, and prevent the generation of such goals in the first place.

The term language of PPLAMBDA is the typed λ -calculus which makes it, theoretically speaking, sufficiently powerful to express all possible algorithms. However, this is no basis for claiming its practicality, and perhaps incorporating a richer syntax for function definitions which includes both let and cases constructs would help to make such expression clearer and conciser. A cases construct in particular would remove much of the present need for selector and discriminator functions. Lawrence Paulson has extended the logical language of PPLAMBDA in Cambridge LCF to include more propositional connectives and the existential quantifier, with appropriate inference rules for them.

It is certainly of theoretical interest that domain equations can be axiomatised purely on the basis of the Computational Induction principle and fixed points. The various induction and axiomatisation packages that have been independently developed in LCF automate the required translation processes completely. Note that each time a structural induction is used, some kind of systematic reduction to Computational Induction is performed.

However, in practice, this axiomatisation could be made clearer and simpler if the structural induction principle could be stated

more directly. Since structural induction generally places a kind of continuous reachability constraint upon the domain, it would be sufficient to state this as an atomic property of an (algebraically specified) type. The form of the required structural induction principle could then be efficiently inferred from the constructor signature together with this atomic assertion.

In order to apply proof assistants like LCF to the formal specification and verification of systems and software, careful consideration must be given to the underlying logical framework used. Any such system should be carefully designed, as PPLAMBDA was, and might also contain a logical sub-language for expressing propositions, a term sub-language for expressing values, and a type sub-language for expressing well-formedness constraints. Other choices for the semantical structure of the specification language are possible, ranging from a purely type theoretic approach (e.g. [Constable83], [ConstableZlatin84], [MartinLof79]), where propositions are themselves values, to a more "model-oriented" approach (e.g. [Sufrin82], [BjornerJones82]) where all entities are values, or denotations, of some description. More importantly, the formal semantics of the specification language should be used to demonstrate the soundness of a well-designed collection of inference rules. Among the design criteria of such systems should be the naturalness with which standard proof methods can be applied by the theorem prover, human or otherwise.

Concerning the general development of theories using LCF, it seems that the present approach of developing theories usually proceeds forwards from the axioms. In order to prove a lemma, it is necessary to have established (or at least isolated) those additional facts that your proof will need. This is because it is not a valid tactical step to simply add arbitrary formulae as assumptions without giving a justification for doing so. Note that USELEMMASTAC (see Section 4.7.1) always adds the conclusions of lemmas already established.

So, an LCF user is almost required to have much of the formal proof firmly in mind, at least in the form of a lemma sequence, before approaching the machine. The tactical approach encourages a

top-down approach to the generation of individual theorems, but does not assist the top-down development of theories. What seems to be needed is some way of deferring proofs of auxiliary lemmas until later.

One possibility might be to record a formula as being a putative result. This states a promise that the given formula will be established later, with hypotheses drawn from some stated set of formulae. These putative results could then stand as theorems within the proofs of lemmas. Of course, this brings with it the obligation to record when putative results have been used in proofs. It should then be possible to arrange for lemmas dependent upon some putative result to automatically release that dependence as soon as the putative result is fully established (using no more hypotheses than those originally stated).

At present, LCF faithfully simulates all of the inferences required to establish a particular result. Within simplification, for example, all the inferences required to show that two terms are equal (e.g. by transitivity, symmetry, substitution, etc) are actually performed. It is sometimes possible to characterise the result of a sequence of inferences by reference to non-logical features, such as the distribution of variables in a formula. For example, the right-nested normal form for an associative function can be obtained by successive rewritings by an appropriate associativity theorem. Such characterisations are simple examples of meta theorems, that is, theorems about theorems (or their proofs). More sophisticated examples are decision procedures for recognising particular classes of valid formulae (e.g equivalence of regular expressions, Presburger arithmetic, etc.) which use more efficient techniques than by the construction of formal proofs (see [BundySterling81], [Weyhrauch78]). LCF would not allow these techniques to produce theorems unless they also provide a means of providing the corresponding formal justification in each case, which is precisely the computational overhead that was to be avoided. Boyer and Moore have suggested their "metafunctions" approach as a way of enriching the fundamental set of inference rules (see [BoyerMoore80]). Their suggestion is that an arbitrary

LISP function may be admitted to the set of basic "proof procedures" only if a proof of its correctness (i.e soundness) accompanies it.

A subtle variant of this problem also arises in connection with the use of valid tactics. By definition, the validation function of such tactics always produces a theorem achieving the original goal when applied to theorems satisfying the refined list of subgoals. However, if it is known beforehand that a certain tactic is valid, then the act of applying the validation function is, in a sense, redundant since the validation will always succeed. It is not possible within the present Edinburgh LCF system to express the formal proposition that a given tactic is valid in such a way that this knowledge can be used by the LCF system to optimise the validation function. A semantically sound approach to this could involve formulating an LCF theory of LCF tactics, or even of LCF itself!

In some sense, the functions METARULE and METATAC go some way to providing this kind of efficient attachment of new rules within LCF as it stands. Both ML functions use a proven theorem to construct an inference rule and tactic based upon it. However, the present scheme is certainly limited in that schematic rules like universal generalisation cannot be represented as a single theorem (in a logic without predicate variables, such as PPLAMBDA). In addition, it is difficult to see how these functions could be used to establish as a bone fide theorem some "fact" that is efficiently recognised by an arbitrary decision procedure.

Some design limitations of the Edinburgh LCF system are briefly mentioned below:-

Theorem schemes (i.e meta abbreviations of theorems) in LCF cannot be stored directly, because these in general depend upon certain ML bindings having to exist in scope.

Formulae cannot be "re-abbreviated" once anti-quotations have been expanded.

There is no formal relationship between peices of ML code and the LCF theories that they could be run against. For example, the Monoid Induction tactic, MONOIDTAC, could be invoked in the context of any theory at all. This does not cause any particular harm, except that it wastes time on an evaluation which has to fail because the appropriate constants may either not exist, or they have the wrong semantics. More subtly, theory specific ML code also represents valuable information about the theory in any case, and so has just as much right to be included, in some sense, in the database of LCF theories.

LCF does not keep track of which (recorded) theorems or axioms are used in the proofs of theorems (note that this is different from recording hypotheses, which LCF does deal with). Because of this, it is difficult to arrange for some kind of minimal recomputation of proofs when minor changes are made to a specification. This could therefore lead to arbitrary amounts of unnecessary repetition of previous work when developing and verifying a large specification. Such information could be used to help determine which tactically generated proofs might need modification as a result of reformulations of the theory database.

The user interface to the Edinburgh LCF system is not as helpful and informative as it might be. When loading ML files to the system, evaluation of definitions will halt upon the first error encountered. The development of good programming support tools such as structure editors and other interactive data management aids could enhance the perception of how systems like LCF could be applied.

The underlying structure of LCF theories forms a hierachical database describing a simple inheritance relationship. With this very simple structure, it is difficult to arrange for the re-use of theories in widely different

contexts and to impose some kind of information hiding regime giving conventional data abstraction (see also the work of [BurstallSannella83]).

Future directions.

There are many interesting avenues of research for the continued development of proof generation systems like LCF, some of which might consider the following.

At present, LCF provides only marginal support for the creative discovery of mathematical theorems. To some extent, LCF's tactics provide a way of presenting certain kinds of high-level proof plans. However, they do not help with the discovery of the required lemma sequence. Aids for sketching incomplete fragments of formal proofs would be extremely useful. In addition, LCF has no formal representation of which tactics might be of assistance in tackling particular goals. In general, machine assistance for the discovery and planning of proof attempts would be desirable.

The kind of theorem that LCF seems to be good at are long, tedious, but essentially shallow proofs. The fact that they can be generated at all by a regular language of tactics tends to suggest this. Future builders of systems for proving interesting theorems might consider, as a long term goal, the attempt to formalise and verify some well known difficult theorems from general mathematics, such as the Prime Number Distribution Theorem^{*} (see [Apostol76]). This theorem is interesting because it is a well-known example of a number-theoretic theorem whose "easiest" proofs make use of certain properties of Riemann's zeta function and uses the theory of complex analytic functions. This indicates that, in general, proofs of mathematical theorems may use techniques that are widely separated from the theory within which they are stated.

The use of tactics could have serious application elsewhere in computing. For example, the stepwise refinement of programs (see [Dahl et al 72], [Jones83], for example) could be easily cast in this form, where goals correspond to specifications of systems to

* There are "elementary" proofs of this theorem due independently to Erdos and Selberg (see [Apostol76] for references).

be implemented and events correspond to program fragments (i.e. implementations of systems). The appropriate achievement relation then says when a program fragment meets, or satisfies, its specification. The application of a "tactic" to a specification would produce a collection of sub-specifications and a validation part that shows how to fit together program fragments satisfying sub-specifications to form a program fragment satisfying the original specification. Note that in this case the validation part is of intrinsic value, since the program would essentially be "accumulated" within it, as decomposition and data refinement proceeds.

Another interesting direction is in contributions to the mathematical theory of control strategies used in automatic theorem provers. One approach to the theory might take the LCP work on tactics as its starting point to show how to compose tactically represented strategies to build better, more flexible, theorem provers. Some examples of such strategies might include the general matings approach (see [Andrews80], [Andrews81]), and the further exploration of the resolution method, using both forwards and backwards chaining forms of resolution tactics.

References

Abbreviations.

CACM	Communications of the ACM
EUCSD	Edinburgh University Computer Science Department
JACM	Journal of the ACM
JCSS	Journal of Computer and System Science
LNCS	Lecture Notes in Computer Science
POPL	Principles of Programming Languages conference
SIAM	Society for Industrial and Applied Mathematics
ToPLaS	Transactions on Programming Languages and Systems

[ADJ77]

Goguen J A, Thatcher J, Wagner E, Wright J, *Initial algebra semantics and continuous algebras*, JACM, Vol 24, No 1, pp 68-95, January 1977

[Andrews80]

Andrews P B, *Transforming matings into natural deduction proofs*, Proc. 5th Conf. Automated Deduction, (ed: Bibel, Kowalski), Les Arcs, LNCS 87, pp 281-292, 1980

[Andrews81]

Andrews P B, *Theorem Proving via general matings*, JACM, Vol 28, No 2, pp 193-214, 1981

[Apostol76]

Apostol T M, *Introduction to Analytic Number Theory*, Undergraduate texts in Mathematics, Springer Verlag, 1976

[Aubin76]

Aubin R, *Mechanizing Structural Induction*, PhD Thesis, Edinburgh University, 1976

[Backus78]

Backus J, *Can Programming be Liberated From the Von Neumann Style?*, A Functional Programming Style and it's Algebra of Programs, CACM, Vol 21, No 8, 1978

[Bird76]

Bird R, *Programs and Machines:- An Introduction to the Theory of Computation*, John Wiley and Sons, London, 1976

[Birkhoff67]

Birkhoff G, *Lattice Theory*, American Mathematical Society Providence Rhode Island, 1967

[BjornerJones82]

Bjorner D, Jones C B, *Formal Specification and Software Development*, Prentice Hall, 1982

[BoyerMoore79]

Boyer R S, Moore J S, *A Computational Logic*, Academic Press, 1979

[BoyerMoore80]

Boyer R S, Moore J S, *Metafunctions: proving them correct and using them efficiently as new proof procedures*, from "The Correctness Problem in Computer Science", (ed. Boyer, Moore), pp 103-184, Academic Press, 1980

[BroyWirsing81]

Broy M, Wirsing M, *On the algebraic extensions of abstract data types*, Proc. Int. Coll. Formalisation of Programming Concepts, LNCS 107, (ed Diaz, J and Ramos, I), April 1981

[Bundy83]

Bundy A, *The Computer Modelling of Mathematical Reasoning*, Academic Press, 1983

[BundySterling81]

Bundy A, Sterling L S, *Meta-level inference in Algebra*, Research paper 164, Dept. of Artificial Intelligence, Edinburgh, September 1981

[Burstall80]

Burstall R M, *Electronic Category Theory*, Proc. 9th Int. Symp. on Mathematical Foundation of Computer Science, Rydzyna, Poland, LNCS 88, pp 22-40, 1980

[BurstallGoguen80a]

Burstall R M, Goguen J A, *An informal introduction to specifications using Clear*, from "The Correctness Problem in Computer Science", (ed. Boyer, Moore), pp 185-213, Academic Press, 1980

[BurstallGoguen80b]

Burstall R M, Goguen J A, *The Semantics of Clear - A Specification Language*, Proc. Advanced Course on Abstract Software Specifications, LNCS 86, (ed: Bjorner, D), pp 292-332, 1980

[BurstallGoguen82]

Burstall R M, Goguen J A, *Algebras, Theories and Freeness - An Introduction for Computer Scientists*, Theoretical Foundations of Programming Methodology, (ed: Broy, M, Schmidt, E), pp 329-350, D. Reidel Pub. Co, Dordrecht, 1982

[BurstallLandin69]

Burstall R M, Landin P J, *Programs and their Proofs:- an Algebraic Approach*, Machine Intelligence 4, pp 17-43, Edinburgh University Press, 1969

[BurstallSannella83]

Burstall R M, Sannella D T, *Structured Theories in LCF*,
Proc. 8th Coll. on Trees in Algebra and Programming, L'Aquila,
Italy, March 1983 (also:- EUCSD Report, CSR-129-83, February
1983)

[Cartwright82]

Cartwright R, *Towards a logical theory of program data*,
Proc. Workshop on "Logics of Programs", (ed: Kozen, D), LNCS
131,, pp 37-51, 1982

[ChangLee73]

Chang C-H, Lee R C-T, *Symbolic Logic and Mechanical Theorem
Proving*, Academic Press, 1973

[Clocksin81]

Clocksin W F, Mellish C S, *Programming in PROLOG*, Springer
Verlag, New York, 1981

[Cohn65]

Cohn P M, *Universal Algebra*, Harper and Row, 1965

[Cohn79]

Cohn A, *Machine Assisted Proofs of Recursion Implementation*,
PhD thesis, University of Edinburgh, 1979

[Cohn81]

Cohn A, *The Equivalence of Two Semantic Definitions:- a Case
Study in LCF*, SIAM J. of Computing, Vol 12, pp 267-285, May
1983

[Cohn82]

Cohn A, *The Correctness of a Precedence Parsing Algorithm in
LCF*, Technical Report 21, Computer Laboratory, University of
Cambridge, April 1982

[CohnMilner82]

Cohn A, Milner R, *On Using Edinburgh LCF to Prove the
Correctness of a Parsing Algorithm*, EUCSD Report, CSR-113-82,
1982

[Constable83]

Constable R L, *Programs as proofs: a synopsis*, Inf. Proc.
Letters, Vol 16, pp 105-112, 1983

[ConstableZlatin84]

Constable R L, Zlatin D R, *The Type Theory of PL/CV3*, ACM
ToPLaS, Vol 6, No 1, pp 94-111, January 1984

[CorbinBidoit83]

Corbin J, Bidoit M, *A Rehabilitation of Robinson's Unification Algorithm*, IFIP 83, (ed:- Mason, R.E), North Holland, Amsterdam, pp 909-914, 1983

[CourcelleRaoult80]

Courcelle B, Raoult J C, *Completions of Ordered Magmas*, Annales Societatis Mathematicae Polonae series vi, Fundamenta Informaticae iii.i, pp 105-116, 1980

[Dahl et al 72]

Dahl O-J, Dijkstra E W, Hoare C A R, *Structured programming*, Academic Press, 1972

[DeBakker80]

de Bakker J, *Mathematical Theory of Program Correctness*, Prentice-Hall, 1980

[Fisher84]

Fisher M, *An investigation into the use of Edinburgh LCF in VDM data type refinement*, MSc dissertation, University of Manchester, October 1984

[Gerhart et al 80]

Gerhart S, Musser D R, Thompson D H, Baker D A, Bates R L, Erickson R W, London R L, Taylor D G, Wile D S, *An overview of AFFIRM: A specification and verification system*, IFIP'80 (ed:Lavington, S), pp 343-348, North Holland, Amsterdam, 1980

[Giles78]

Giles D A, *The Theory of LISTS in LCF*, EUCSD Report, CSR-31-78, October 1978

[Gloess80]

Gloess P, *An experiment with the Boyer-Moore Theorem Prover; A proof of the correctness of a Simple Parser of Expression*, Proc. 5th Conf. Automated Deduction, (ed: Bibel, Kowalski), Les Arcs, LNCS 87, pp 154-169, 1980

[GoguenTardo77]

Goguen J A, Tardo J J, *An introduction to OBJ: A language for writing and testing formal algebraic program specifications*, Proc. Conf. on "Specification of Reliable Software", Cambridge, Massachusetts, 1977

[Gordon79]

Gordon M J C, *The Denotational Description of Programming Languages*, Springer Verlag, 1979

- [Gordon81]
Gordon M J C, *A Model of Register Transfer Systems with Applications to Microcode and VLSI Correctness*, EUCSD Report, CSR-82-81, 1981
- [Gordon82]
Gordon M J C, *Representing a Logic in the LCF Metalanguage, Tools and Notations for Program Construction*, (ed: Neel, D), pp 163-185, Cambridge University Press, 1982
- [Gordon83a]
Gordon M J C, *LCF-LSM*, Technical Report 41, Computer Laboratory, University of Cambridge, 1983
- [Gordon83b]
Gordon M J C, *Proving a Computer Correct with the LCF-LSM Hardware Verification System*, Technical Report 42, Computer Laboratory, University of Cambridge, 1983
- [GordonMilner78]
Gordon M, Milner R, Morris L, Newey M, Wadsworth C, A *Metalanguage for Interactive Proof in LCF*, 5th ACM POPL at Tucson, Arizona, 1978
- [Gratzner79]
Gratzner G, *Universal Algebra*, (2nd. edition), Springer Verlag, 1979
- [Guttag75]
Guttag J V, *The Specification and Application to Programming of Abstract Data Types*, PhD Thesis, University of Toronto, 1975
- [HOPE80]
Burstall R M, MacQueen D B, Sannella D T, *HOPE - An Experimental Applicative Language*, Proc. 1980 LISP Conference, Stanford, California, pp 136-143, 1980 (also:- EUCSD Report, CSR-62-80, 1980 (revised 1981)),
- [Hoare69]
Hoare C A R, *An Axiomatic Basis for Computer Programming*, CACM, Vol 12, pp 576-580, 1969
- [Hoare72]
Hoare C A R, *Proof of Correctness of Data Representations*, Acta Informatica, Vol 1, pp 271-281, 1972
- [Huet80]
Huet G, *Confluent Reductions - Abstract Properties and Applications to Term Rewriting Systems*, JACM, Vol 27, No 4, pp 797-821, October 1980

[HuetOppen80]

Huet G, Oppen D C, *Equations and Rewrite Rules:- A survey*, in "Formal Languages:- Perspective and Open Problems", (ed:Book, R), Academic Press, 1980 (also in SRI International Report CSL-111)

[Jones80]

Jones C B, *Software Development:- A Rigorous Approach*, Prentice Hall, 1980

[Jones83]

Jones C B, *Systematic Program Development*, invited paper, Symposium "Wiskunde en Informatica", Centrum voor Wiskunde en Informatica (formerly Mathematical Center), Amsterdam, November 1983

[Kleene52]

Kleene S C, *Introduction to Metamathematics*, Studies in Logic, North Holland, Amsterdam, 1952

[LCF]

Gordon M J C, Milner A J R G, Wadsworth C P, *Edinburgh LCF: A mechanised logic of computation*, LNCS 78, Springer Verlag, New York, 1979

[Landin66]

Landin P J, *The Next 700 Programming Languages*, CACM, Vol 9, No 3, pp 157-166, March 1966

[LehmannSmyth78]

Lehmann D J, Smyth M B, *Algebraic Specifications of Data Types:- A Synthetic Approach*, Mathematical Systems Theory, Vol 14, pp 97-139, 1978 (also:- Report 115, Department of Computer Studies, University of Leeds)

[LeszczylowskiWirsing82]

Leszczylowski J, Wirsing M, *A System for Proving within and about Abstract Data Types*, Proc. 5th Int. Symp. on Programming, Turin, (ed:Dezani-Ciancaglini, M, Montanari, V), 1982

[Leszczylowski80a]

Leszczylowski J, *An Experiment with Edinburgh LCF*, Proc. 5th Conf. Automated Deduction, (ed: Bibel, Kowalski), Les Arcs, LNCS 87, 1980

[Leszczylowski80b]

Leszczylowski J, *Theory of FP Systems in Edinburgh LCF*,
EUCSD Report, CSR-61-80, April 1980

[LevyMaibaum82]

Levy M, Maibaum T S E, *Continuous Data Types*, SIAM J. on
Computing, Vol 11, No 2, pp 201-216, May 1982

[Loeckx80a]

Loeckx J, *Proving properties of Algorithmic Specifications of
Abstract Data Types in AFFIRM*, AFFIRM Memo-29-JL, Program
Verification Project, USC ISI,, 4676 Admiralty Way, Marina del
Rey, CA 90291, July 1980

[Loeckx80b]

Loeckx J, *Algorithmic specifications of abstract data types*,
Fachbereich 10-Informatik, Universitat des Saarlandes, 6600
Saarbrücken, West Germany, 1980

[MB]

Mac Lane S, Birkhoff G, *Algebra*, Macmillan, 1979

[Manna74]

Manna Z, *Mathematical theory of computation*, McGraw Hill,
1974

[MannaWaldinger81]

Manna Z, Waldinger R, *Deductive synthesis of the Unification
algorithm*, Science of Computer Programming, Vol 1, pp 5-48,
1981

[Martelli82]

Martelli A, Montanari U, *An efficient unification algorithm*,
ACM ToPLaS, Vol 4, No 2, pp 258-282, 1982

[MartinLof79]

Martin Lof P, *Constructive Mathematics and computer
programming*, Proc. 6th Int. Congress for Logic, Methodology,
and Philosophy, of Science, Hannover, 1979

[McCarthyPainter67]

McCarthy J, Painter J A, *Correctness of a compiler for
arithmetic expressions*, Proc. Symp. Applied Mathematics, Vol
19,, Mathematical aspects of computer science, pp 33-41, (ed:
Schwartz, J T), American Mathematical Society, Providence,
Rhode island, 1967

[Milner71]

Milner R, *An algebraic definition of simulation between
programs*, Proc. 2nd IJCAI, Imperial College, London,
September 1971

[Milner72a]

Milner R, *Logic for computable functions. Description of a machine implementation*, Artificial Intelligence Memo 169, Computer Science department, Stanford University, 1972

[Milner72b]

Milner R, *Implementation and applications of Scott's logic of computable functions*, Proc. Conf. on "Proving assertions about programs", SIGPLAN Notices 7, Vol 1, 1972

[Milner72c]

Milner R, *Models of LCF*, Artificial Intelligence Memo 186, Computer Science Department, Stanford University, 1972 (also:- Mathematical Centre Tracts 82, pp 49-63, Amsterdam, 1976)

[Milner76]

Milner R, *Program semantics and mechanised proof*, Mathematical Centre Tracts 82, pp 3-44, Amsterdam, 1976

[Milner77]

Milner R, *Flowgraphs and Flow Algebras*, JACM, Vol 26, No 4, 1979

[Milner78]

Milner R, *A theory of type polymorphism in programming*, JCSS, Vol 17, No 3, pp 348-375, December 1978

[Milner83]

Milner R, *How ML evolved*, "Polymorphism", Vol 1, No 1, Bell Laboratories, Murray Hill, January 1983

[MilnerWeyhrauch72]

Milner R, Weyhrauch R, *Proving compiler correctness in a mechanised logic*, Machine Intelligence 7, (ed:- Michie, D), Edinburgh University Press, 1972

[Milner et al 75]

Milner R, Morris L, Newey M, *A logic for computable functions with reflexive and polymorphic types*, Proc. Conf. on "Proving and improving programs", Arc-et-Senans,, 1975

[Mulmuley84]

Mulmuley K, *Mechanisation of existence proofs of recursive functions.*, Proc. 7th Int. Conf. Automated deduction, LNCS 170 (ed: Shostak, R), Springer Verlag, New York, pp 460-475, 1984

[Musser80]

Musser D, *Abstract data type specification in the AFFIRM system*, IEEE transactions on Software Engineering, Vol SE-6, No 1, pp 24-32, January 1980

[NelsonOppen80]

Nelson G, Oppen D.C, *Fast Decision Procedures based on Congruence Closure*, JACM, Vol 27, No 2, pp 356-364, April 1980

[Park70]

Park D, *Fixpoint induction and proofs of program properties*, Machine Intelligence 5, pp 59-78, Edinburgh University Press, 1969

[Paterson78]

Paterson M S, Wegman M N, *Linear Unification*, JCSS, Vol 16, pp 158-197, 1978

[Paulson83a]

Paulson L, *Structural induction in LCF*, Proc. Conf. on "Semantics of Data Types", (ed: Kahn, G, MacQueen, D B, Plotkin, G), LNCS 173, pp 197-214, June 1984 (also:- Technical report 44, Computer Laboratory, University of Cambridge, 1983)

[Paulson83b]

Paulson L, *The revised logic PPLAMBDA: A reference manual*, Technical report 36, Computer Laboratory, University of Cambridge, 1983

[Paulson83c]

Paulson L, *A higher order implementation of rewriting*, Science of Computer Programming, Vol 3, No 2, pp 119-149, August 1983 (also:- Technical report 35, Computer Laboratory, University of Cambridge, 1983)

[Paulson83d]

Paulson L, *Tactics and tacticals in Cambridge LCF*, Technical report 39, Computer Laboratory, University of Cambridge, 1983

[Paulson83e]

Paulson L, *Recent developments in LCF: examples of structural induction*, Technical report 34, Computer Laboratory, University of Cambridge, 1983

[Paulson84]

Paulson L, *Verifying the unification algorithm in LCF*, Technical report 50, Computer Laboratory, University of Cambridge, March 1984 (to appear in "Science of Computer Programming")

[Plotkin72]

Plotkin G D, *Building in equational theories*, Machine Intelligence 7, pp 73-90, Edinburgh University Press, 1972

- [Plotkin76]
Plotkin G D, *A powerdomain construction*, SIAM J. on Computing, Vol 5, No 3, pp 452-487, September 1976
- [Plotkin78]
Plotkin G D, T^ω as a Universal Domain, JCSS, Vol 17, No 2, pp 209-236, October 1978
- [Prawitz65]
Prawitz D, *Natural Deduction - A proof theoretical study*, Stockholm studies in Philosophy 3, Almqvist & Wiksell, Uppsala, 1965
- [Quine81]
Quine W V, *Mathematical Logic*, (revised edition) Harvard University Press, 1981 (first pub. 1940)
- [Reynolds70]
Reynolds J, *GEDANKEN: A simple typeless programming language, based on the principle of completeness and the reference concept*, CACM, Vol 13, No 5, 1970
- [Robinson65]
Robinson J A, *A machine oriented logic based on the Resolution principle*, JACM, Vol 12, No 1, pp 23-41, January 1965
- [Robinson79]
Robinson J A, *Logic: Form and Function, the mechanisation of deductive reasoning*, Edinburgh University Press, 1979
- [Russell77]
Russell B, *Implementation correctness involving a language with GOTO statements*, SIAM J. of Computing, Vol 6, No 3, September 1977
- [Rydeheard81]
Rydeheard D, *Applications of category theory to programming and program specification*, PhD Thesis, University of Edinburgh, 1981
- [Sannella82]
Sannella D S, *Semantics, Implementations and Pragmatics of Clear, A Program Specification Language*, PhD Thesis, University of Edinburgh, 1982
- [Schmidt83a]
Schmidt D, *A programming notation for tactical reasoning*, Proc. 7th Int. Conf. Automated Deduction, LNCS 170, (ed: Shostak, R E), pp 445-458, February 1984

- [Schmidt83b]
Schmidt D, *Natural deduction theorem proving in set theory*,
EUCSD Report, CSR-142-83, 1983
- [Schoenfield67]
Schoenfield J R, *Mathematical Logic*, Addison-Wesley, 1967
- [Scott71]
Scott D, *Continuous Lattices*, Technical Monograph PRG 7,
Programming Research Group, Oxford, August 1971
- [Scott76]
Data types as lattices, SIAM J. of Computing, Vol 5, No 3,
pp 522-587, Sept 1976
- [Scott82]
Scott D S, *Domains for Denotational Semantics*, Proc.
ICALP'82, LNCS 140, (ed:Neilsen and Schmidt), July 1982
- [ScottDeBakker69]
Scott D S, DeBakker J W, *A theory of programs*, (unpublished
seminar notes), IBM seminar, Vienna, 1969
- [ScottStrachey71]
Scott D S, Strachey C, *Towards a mathematical semantics of
computer languages*, Proc. Symp. Computers and Automata,
Brooklyn, pp 19-46, April 1971
- [Shostak78]
Shostak R.E, *An algorithm for reasoning about equality*,
CACM, Vol 21, No 7, pp 583-585, July 1978
- [Smyth78]
Smyth M B, *Power Domains*, JCSS, Vol 16, pp 23-36, 1978
- [SmythPlotkin82]
Smyth M B, Plotkin G D, *The Category theoretic solution of
recursive domain equations*, EUCSD CSR-102-82, University of
Edinburgh, 1982
- [Sokolowski83a]
Sokolowski S, *A note on tactics in LCF*, EUCSD Report, CSR-
140-83, 1983
- [Sokolowski83b]
Sokolowski S, *An LCF proof of soundness of Hoare's logic - a
paper without a happy ending*, EUCSD Report, CSR-146-83, 1983
- [Stickel81]
Stickel M E, *A unification algorithm for associative -
commutative functions*, JACM, Vol 28, No 3, pp 423-434, July
1981

[Stoy77]

Stoy J, *Denotational Semantics: The Scott-Strachey approach to programming language theory*, MIT Press, Massachusetts, 1977

[Suftrin83]

Suftrin B, *Formal Specification of a Display Oriented Text Editor*, Science of Computer Programming, Vol 1, Number 3, May 1983

[Wadsworth78]

Wadsworth C P, *Domain theory course notes*, Postgraduate computation theory course notes, unpublished, 1978

[Weyhrauch78]

Weyhrauch R, *Prolegomena to a theory of formal reasoning*, Artificial Intelligence Memo 315, Stanford University, December 1978

[WeyhrauchMilner72]

Weyhrauch R, Milner R, *Program semantics and correctness in a mechanised logic*, Proc. USA-Japan Computer Conference, Tokyo, 1972

Appendix 1.A Table of PPLAMBDA ConstantsGeneral

$1 : \alpha$	- least element
$\partial : \alpha \rightarrow \text{tr}$	- definedness or self-identity
$\text{COND} : \text{tr} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$	- conditional, usually written as (- $\Rightarrow$ - -).
$\text{FIX} : (\alpha \rightarrow \alpha) \rightarrow \alpha$	- least fixed point

(Internal) Truth values

$\text{TT} : \text{tr}$	- true
$\text{FF} : \text{tr}$	- false

Cartesian Product

$\text{PAIR} : \alpha \rightarrow \beta \rightarrow (\alpha \# \beta)$	- pairing, usually written as $(-, -)$
$\text{FST} : (\alpha \# \beta) \rightarrow \alpha$	- first component
$\text{SND} : (\alpha \# \beta) \rightarrow \beta$	- second component

Smash Product

$(- \otimes -) : \alpha \rightarrow \beta \rightarrow (\alpha \otimes \beta)$	- smash pairing
$\text{P1} : (\alpha \otimes \beta) \rightarrow \alpha$	- first smash component
$\text{P2} : (\alpha \otimes \beta) \rightarrow \beta$	- second smash component

Coalesced Sum

$\text{INL} : \alpha \rightarrow (\alpha + \beta)$	- left injection
$\text{INR} : \beta \rightarrow (\alpha + \beta)$	- right injection
$\text{OUTL} : (\alpha + \beta) \rightarrow \alpha$	- left selection
$\text{OUTR} : (\alpha + \beta) \rightarrow \beta$	- right selection
$\text{ISL} : (\alpha + \beta) \rightarrow \text{tr}$	- left discrimination
$\text{ISR} : (\alpha + \beta) \rightarrow \text{tr}$	- right discrimination

Lifting

$\text{UP} : \alpha \rightarrow \alpha_1$	- lift (or "freeze")
$\text{DOWN} : \alpha_1 \rightarrow \alpha$	- drop (or "thaw")

Appendix 2.Machine processable concrete syntax for PPLAMBDA.

Throughout this document, a formal notation for PPLAMBDA has been used that is quite pleasant to write in. However, this is not technically convenient from the machine interaction point of view, (at least with standard computer terminal technology). Hence, it is necessary to give various conventions and translations of special symbols into more suitable codes. The basic concrete syntax is:-

Written notation.Machine notation.Forms.

V

!

D

IMP

=

==

E

<<

Terms. ∂

DEF

 λ

\

 $\Rightarrow$ $\Rightarrow$ Types.

#

ℓ

 $\rightarrow$ $\rightarrow$ ty₁(ty)_u $\alpha, \beta, \dots$ (type variables.)

*, **, ...

or

*id, **id, ...

During the case studies, a theory of smash product called 'SMASH' was defined and used (see Section 2.1.2 and Appendix 3). The concrete syntax actually used for this is as follows:-

(ty₁ @ ty₂)(ty₁, ty₂)X

(x @ y)

(x ℓ y)

Unfortunately, Edinburgh LCF does not permit user-defined type operators to be infix symbols. Note also that the symbol 'ℓ' is used in two different ways in the machine syntax; as the infix Cartesian product type operator in PPLAMBDA and also as the infix smash pairing function.

Appendix 3.

Listing of LCF theories used and generated.

A listing is given of the LCF theories generated during the course of the work presented above. All of the theorems shown have been generated through using the Edinburgh LCF system. As an aid to presentation, the names of theorems have been emboldened and offset into the margin. The structure of this listing is as follows:-

(a) Standard theories.

SMASH, KERNEL, MORPH, NAT, NATFUN, PL, BASIC

(b) The theories for the Monoid case studies.

- 1) 1st case study: L, LFUN, LFREE
- 2) 2nd case study: L1, LFUN1, LFREE1
- 3) 3rd case study: L2, LFUN2, LFREE2

(c) The theories for the Multiset case study.

EQFUN, ATOM, LA, LAFUN

THEORY smash

```

newtype 2 'X' ;;
newolcifix ( 'f' , " :*->( **->(*,**)X)" ) ;;
newconstant ( 'P1' , " :(*,**)X->*" ) ;;
newconstant ( 'P2' , " :(*,**)X->***" ) ;;
NEWAXIOMS( );;
S1  " !p:(*,** )X. (P1 p :*) f (P2 p :**) == p"
S2  " !a:*. !b:**. P1((a f b):(*,** )X) == DEF b=>a|UU:*"
S3  " !a:*. !b:**. P2((a f b):(*,** )X) == DEF a=>b|UU:***"

```

FACT smash

```

F1  "(UU:*) f (UU:**) == UU:(*,** )X"
F2  " !a:*. !b:**. DEF b == TT IMP P1((a f b):(*,** )X) == a"
F3  " !a:*. !b:**. DEF a == TT IMP P2((a f b):(*,** )X) == b"
F4  "P1 (UU:(*,** )X) == UU:*"
F5  "P2 (UU:(*,** )X) == UU:***"
F6  " !a:*. a f (UU:**) == UU:(*,** )X"
F7  " !b:**. (UU:*) f b == UU:(*,** )X"
F8  " !p:(*,** )X. DEF(P1 p :*) == DEF p :tr"
F9  " !p:(*,** )X. DEF(P2 p :**) == DEF p :tr"
F10 " !a:*. !b:**. DEF((a f b):(*,** )X) == DEF a=>DEF b|UU:tr"
F11 " !x:(*,** )X. !y:(*,** )X. P1 x == P1 y :* & P2 x == P2 y
    :** IMP x == y"

```

P12 "la:*. lb:**. DEF((a £ b):(**,*)X) == DEF((b £ a):(**,*)
X) :tr"

THEORY kernel

newparent 'smash' ;;

FACT kernel

condUU "lt:tr. t=>UU|UU == UU:*

condeQ "lt:tr. lx:*. DEF t == TT IMP t=>x|x == x"

DEFCOND "lt:tr. lx:*. ly:*. DEF(t=>x|y) == t=>DEF x|DEF y :tr"

BooleanCond "lt:tr. t=>TT|FF == t"

DEFDEF "lx:*. DEF(DEF x :tr) == DEF x :tr"

DEFXTT "lx:*. lt:tr. DEF x=>TT|t == DEF x :tr"

DEFXUU "lx:*. lt:tr. DEF x=>UU|t == UU:tr"

TTDEft "lt:tr. (t=>TT|TT) == DEF t :tr"

DEFaDEFab "la:*. lb:**. DEF a == TT IMP DEF(a, b) == TT"

DEFbDEFab "lb:**. la:*. DEF b == TT IMP DEF(a, b) == TT"

DEFabDEFba "la:*. lb:**. DEF(a, b) == DEF(b, a) :tr"

gUU "lg:**->*. lf:**->*. (lx:**. g(f x) == x) IMP g UU == UU
:**"

DEFf "lg:**->*. lf:**->*. (lx:*. g(f x) == x) & (ly:**. f(g
y) == y) IMP (lx. DEF(f x) == DEF x :tr)"

MIN "lx:*. UU << x"

```

SYNTH  "Ix:*. Iy:*. x << y & y << x IMP x == y"

TRANSEQ "Ix:*. Iy:*. Iz:*. x == y & y == z IMP x == z"

TRANSINEQ "Ix:*. Iy:*. Iz:*. x << y & y << z IMP x << z"

SYM  "Ix:*. Iy:*. x == y IMP y == x"

MONOEQ "Ix:*. Iy:*. IH:*->*. x == y IMP H x == H y"

MONOINEQ "Ix:*. Iy:*. IH:*->*. x << y IMP H x << H y"

EXTEQ "IF:*->**. IG:*->**. (Ix:*. F x == G x) IMP F == G"

EXTINEQ "IF:*->**. IG:*->**. (Ix:*. F x << G x) IMP F << G"

FIXPT "IH:*->*. H(FIX H) == FIX H ;*"

FIXUU "FIX (UU:*->*) == UU:*"

MINFIX "IH:*->*. Ix:*. H x == x IMP FIX H << x"

MONIC "IV:**->*. IU:*->**. Ix:*. Iy:*. (Ix. V(U x) == x) & U
      x << U y IMP x << y"

EQMONIC "IV:**->*. IU:*->**. Ix:*. Iy:*. (Ix. V(U x) == x) &
      U x == U y IMP x == y"

INEQ "Ix:*. Iy:*. (Iz:*. z << x IMP z << y) IMP x << y"

IsoFixThm "IV:**->*. IU:*->**. If:*->*. (Ia:*. V(U a) == a)
      & (Ib:**. U(V b) == b) IMP FIX(\b.U(f(V b))) ==U(FIX f)"

BooleanCondAssoc "Ia:tr. Ib:tr. Ic:tr. a=>TT|(b=>TT|c) == (a=>TT|b)
      =>TT|c"

```

THEORY morph

```

newparent 'kernel' ;;

newconstant ( 'Id' , " :*Id->*Id" ) ;;

newolcinflix ( 'o' , " :(**o->***o)->((*o->**o)->(*o->***o))" ) ;;

NEWAXIOMS();;

```


Id "**!x:***Id. Id x == x"

Comp "**!f:**o->***o. !g:*o->**o. \$o f g == \y:*o.f(g y)**"

FACT morph

CompAssoc "**!f:***->****. !g:**->***. !h:*->**. \$o f(\$o g h
:*->***) == \$o(\$o f g :**->****)h :*->****"**

LeftId "**!f:*->**. \$o (Id:**->**) f == f"**

RightId "**!f:*->**. \$o f (Id:*->*) == f"**

THEORY nat

newparent 'kernel' ;;

newtype 0 'Nat' ;;

newconstant ('NatABS' , ":(.)u+Nat->Nat") ;;

newconstant ('NatREP' , ":Nat->(.)u+Nat") ;;

newconstant ('O' , ":Nat") ;;

newconstant ('Succ' , ":Nat->Nat") ;;

newconstant ('NatFUN' , ":(Nat->Nat)->(Nat->Nat)") ;;

NEWAXIOMS();;

NatABS "**!ABS:Nat. NatABS(NatREP ABS) == ABS"**

NatREP "**!REP:(.)u+Nat. NatREP(NatABS REP) == REP"**

O "**O == NatABS(INL(UP () :(.)u))"**

Succ "**!n:Nat. Succ n == NatABS(INR n)"**

NatUU "IF:Nat->Nat. NatFUN F UU == UU:Nat"

NatO "IF:Nat->Nat. NatFUN F O == O"

NatSucc "IF:Nat->Nat. In:Nat. NatFUN F(Succ n) == Succ(F n)
"

NatFIX "IABS:Nat. FIX NatFUN ABS == ABS"

FACT nat

NatABSUU "NatABS UU == UU:Nat"

Nat "O == NatABS(INL(UP () :(.)u)) & (In:Nat. Succ n == NatABS
(INR n))"

NatREPUU "NatREP UU == UU:(.)u+Nat"

DEFNatREP "Ix:Nat. DEF(NatREP x) == DEF x :tr"

DEFNatABS "Ix:(.)u+Nat. DEF(NatABS x) == DEF x :tr"

THEORY natfun

newparent 'nat' ;;

newconstant ('Zero' , ":Nat->tr") ;;

newconstant ('Pred' , ":Nat->Nat") ;;

newolcinf ('+' , ":Nat->(Nat->Nat)") ;;

newolcinf ('*' , ":Nat->(Nat->Nat)") ;;

NEWAXIOMS();;

Zero "Zero == So (ISL:(.)u+Nat->tr) NatREP :Nat->tr"

Pred "Pred == So (OUTR:(.)u+Nat->Nat) NatREP :Nat->Nat"

Sum " $\lambda n1:Nat. \lambda n2:Nat. \$+ n1 n2 == Zero n1 \Rightarrow n2 \mid Succ(\$+(Pred$
 $n1)n2)$ "

Times " $\lambda n1:Nat. \lambda n2:Nat. \$* n1 n2 == Zero n1 \Rightarrow 0 \mid \$+ n1(\$*(Pred$
 $n1)n2)$ "

FACT natfun

Zero " $\lambda x:Nat. Zero x == ISL(NatREP x) :tr$ "

Pred " $\lambda x:Nat. Pred x == OUTR(NatREP x) :Nat$ "

PredSucc " $\lambda n:Nat. Pred(Succ n) == n$ "

PredO " $Pred 0 == UU:Nat$ "

ZeroO " $Zero 0 == TT$ "

ZeroSucc " $\lambda n:Nat. DEF n == TT \text{ IMP } Zero(Succ n) == FF$ "

THEORY pl

newparent 'kernel' ;;

newolcinflix ('and' , " $:tr \rightarrow (tr \rightarrow tr)$ ") ;;

newolcinflix ('or' , " $:tr \rightarrow (tr \rightarrow tr)$ ") ;;

newolcinflix ('imp' , " $:tr \rightarrow (tr \rightarrow tr)$ ") ;;

newolcinflix ('iff' , " $:tr \rightarrow (tr \rightarrow tr)$ ") ;;

newconstant ('not' , " $:tr \rightarrow tr$ ") ;;

newolcinflix ('paror' , " $:tr \rightarrow (tr \rightarrow tr)$ ") ;;

NEWAXIOMS() ;;

and " $\lambda t1:tr. \lambda t2:tr. t1 \text{ and } t2 == t1 \Rightarrow t2 \mid FF$ "

```

or "t1:tr. t2:tr. t1 or t2 == t1=>TT|t2"
imp "t1:tr. t2:tr. t1 imp t2 == t1=>t2|TT"
iff "t1:tr. t2:tr. t1 iff t2 == t1=>t2|not t2"
not "t1:tr. not t1 == t1=>FF|TT"
TTparor "t1:tr. TT paror t1 == TT"
FFUUparor "t1:tr. t2:tr. t1 << FF IMP t1 paror t2 == t2"

```

FACT p1

```

TTand "t1:tr. TT and t1 == t1"
FFand "t1:tr. FF and t1 == FF"
UUand "t1:tr. UU and t1 == UU:tr"
TTor "t1:tr. TT or t1 == TT"
FFor "t1:tr. FF or t1 == t1"
UUor "t1:tr. UU or t1 == UU:tr"
TTimp "t1:tr. TT imp t1 == t1"
FFimp "t1:tr. FF imp t1 == TT"
UUimp "t1:tr. UU imp t1 == UU:tr"
TTiff "t1:tr. TT iff t1 == t1"
FFiff "t1:tr. FF iff t1 == t1=>FF|TT"
UUiff "t1:tr. UU iff t1 == UU:tr"
notTT "not TT == FF"
notFF "not FF == TT"
notUU "not UU == UU:tr"
FFparor "t1:tr. FF paror t1 == t1"

```

UUparor "l t1:tr. UU paror t1 == t1"

andAssoc "l t1:tr. l t2:tr. l t3:tr. (t1 and t2) and t3 == t1 and
(t2 and t3)"

orAssoc "l t1:tr. l t2:tr. l t3:tr. (t1 or t2) or t3 == t1 or
(t2 or t3)"

andRef1 "l t1:tr. t1 and t1 == t1"

orRef1 "l t1:tr. t1 or t1 == t1"

andDist "l t1:tr. l t2:tr. l t3:tr. t1 and (t2 or t3) == (t1 and
t2) or (t1 and t3)"

orDist "l t1:tr. l t2:tr. l t3:tr. t1 or (t2 and t3) == (t1 or
t2) and (t1 or t3)"

andAnalysis "l t1:tr. l t2:tr. t1 and t2 == TT IMP t1 == TT &
t2 == TT"

orAnalysis "l t1:tr. l t2:tr. t1 or t2 == FF IMP t1 == FF & t2
== FF"

AndComm "l t1:tr. l t2:tr. l t3:tr. DEF t1 == TT & DEF t2 == TT
IMP t1 and (t2 and t3) == t2 and (t1 and t3)"

notnot "l t1:tr. not(not t1) == t1"

andExMid "l t1:tr. t1 and (not t1) == not(DEF t1)"

orExMid "l t1:tr. t1 or (not t1) == DEF t1 :tr"

andDeMorg "l t1:tr. l t2:tr. t1 and t2 == not((not t1) or (not
t2))"

orDeMorg "l t1:tr. l t2:tr. t1 or t2 == not((not t1) and (not
t2))"

THEORY basic

```

newparent 'pl' ;;
newparent 'morph' ;;
newparent 'natfun' ;;

```

FACT basic

```

DEFPair "la:*ss. lb:**ss. DEF((a ∈ b):(*ss,**ss)X ) == (DEF
  a) and (DEF b)"

```

THEORY 1

```

newparent 'basic' ;;
newtype 1 'List' ;;
newconstant ( 'absList' , ":(.)u + (*a ∈ (*a)List)u → (*a)List" ) ;;
newconstant ( 'repList' , ":(*a)List → (.)u + (*a ∈ (*a)List)
  u" ) ;;
newconstant ( 'Nil' , ":(*a)List" ) ;;
newconstant ( 'Cons' , ":*a → ((*a)List → (*a)List)" ) ;;
newconstant ( 'Null' , ":(*a)List → tr" ) ;;
newconstant ( 'isCons' , ":(*a)List → tr" ) ;;
newconstant ( 'Head' , ":(*a)List → *a" ) ;;
newconstant ( 'Tail' , ":(*a)List → (*a)List" ) ;;
newconstant ( 'copyList' , ":((*a)List → (*a)List) → ((*a)
  List → (*a)List)" ) ;;
NEWAXIOMS( );

```

```

absList  "!ABS:(*a)List. absList(repList ABS :(. )u + (*a £ (*a)
List)u) == ABS"

repList  "!REP:(. )u + (*a £ (*a)List)u. repList(absList REP
:(*a)List) == REP"

Nil      "Nil == absList(INL(UP ( ) :(. )u) :(. )u + (*a £ (*a)List)
u) :(*a)List"

Cons     "!a:*a. !l:(*a)List. Cons a l == absList(INR(UP(a,l) :
(*a £ (*a)List)u) :(. )u + (*a £ (*a)List)u) :(*a)List"

Null     "!ABS:(*a)List. Null ABS == ISL(repList ABS :(. )u + (*a
£ (*a)List)u) :tr"

isCons   "!ABS:(*a)List. isCons ABS == (ISL(repList ABS :(. )u
+ (*a £ (*a)List)u)=>FF|TT)"

Head     "!ABS:(*a)List. Head ABS == FST(DOWN(OUTR(repList ABS
:(. )u + (*a £ (*a)List)u) :(*a £ (*a)List)u) :*a £ (*a)List)
:*a"

Tail     "!ABS:(*a)List. Tail ABS == SND(DOWN(OUTR(repList ABS
:(. )u + (*a £ (*a)List)u) :(*a £ (*a)List)u) :*a £ (*a)List)
:(*a)List"

copyList "!FUN:(*a)List -> (*a)List. !ABS:(*a)List. copyList
FUN ABS == (Null ABS=>Nil|Cons(Head ABS :*a)(FUN(Tail ABS)
) :(*a)List)"

FIXList  "!ABS:(*a)List. FIX (copyList:((*a)List -> (*a)List)
-> ((*a)List -> (*a)List)) ABS == ABS"

```

FACT 1

```

constructList "Nil == Nil:(*a)List & (!a:*a. !l:(*a)List. Cons
a l == Cons a l :(*a)List)"

selectList  "(!ABS:(*a)List. Head ABS == Head ABS :*a) & (!ABS.
Tail ABS == Tail ABS :(*a)List)"

coverList   "!ABS:(*a)List. (Null ABS=>Nil|Cons(Head ABS :*a)
(Tail ABS :(*a)List)) == ABS"

absListUU   "absList (UU:(. )u + (*a £ (*a)List)u) == UU:(*a)List"

```

```

repListUU  "repList (UU:(*a)List) == UU:(.)u + (*a £ (*a)List)
            u"

DEFPabsList  "!REP:(.)u + (*a £ (*a)List)u. DEF(absList REP :
            (*a)List) == DEF REP :tr"

DEFrepList  "!ABS:(*a)List. DEF(repList ABS :(.)u + (*a £ (*a)
            List)u) == DEF ABS :tr"

DEFNil  "DEF (Nil:(*a)List) == TT"

DEFCons  "!a:*a. !l:(*a)List. DEF(Cons a l :(*)List) == TT"

NullUU  "Null (UU:(*a)List) == UU:tr"

isConsUU  "isCons (UU:(*a)List) == UU:tr"

HeadUU  "Head (UU:(*a)List) == UU:*a"

TailUU  "Tail (UU:(*a)List) == UU:(*a)List"

copyListUU  "!FUN:(*a)List -> (*a)List. copyList FUN (UU:(*a)
            List) == UU:(*a)List"

NullNil  "Null (Nil:(*a)List) == TT"

NullIMPNil  "!l:(*a)List. Null l == TT IMP l == Nil:(*a)List"

NullIMPCons  "!l:(*a)List. Null l == FF IMP l == Cons(Head l
            :*a)(Tail l :(*)List) :(*)List"

NullCons  "!a:*a. !l:(*a)List. Null(Cons a l :(*)List) == FF"

HeadCons  "!a:*a. !l:(*a)List. Head(Cons a l :(*)List) == a"

TailCons  "!a:*a. !l:(*a)List. Tail(Cons a l :(*)List) == l"

```


THEORY lfun

```

newparent 'l' ;;
newconstant ( 'Unit' , " :(*a) -> (*a)List" ) ;;
newolinfix ( '@' , " :(*a)List & (*a)List -> (*a)List" ) ;;
NEWAXIOMS();;

Unit " !a:*a. Unit a == Cons a (Nil:(*a)List) :(*a)List"

Append " !l1:(*a)List. !l2:(*a)List. l1 @ l2 == (Null l1=>l2|Cons
  (Head l1 :a)((Tail l1 :(*a)List) @ l2):(*a)List) :(*a)List"

```

FACT lfun

```

UUApp " !l:(*a)List. (UU:(*a)List) @ l == UU:(*a)List"

NilApp " !l:(*a)List. (Nil:(*a)List) @ l == l"

ConsApp " !a:*a. !l1:(*a)List. !l2:(*a)List. (Cons a l1 :(*a)
  List) @ l2 == Cons a((l1 @ l2):(*a)List) :(*a)List"

AppNil " !l:(*a)List. l @ (Nil:(*a)List) == l"

Assoc " !l1:(*a)List. !l2:(*a)List. !l3:(*a)List. ((l1 @ l2)
  :(*a)List) @ l3 == (l1 @ ((l2 @ l3):(*a)List)):(*a)List"

ConsUnit " !a:*a. !l:(*a)List. Cons a l == ((Unit a :(*a)List)
  @ l):(*a)List"

```

THEORY lfree

newparent 'lfun' ;;

newconstant ('FreeMonoid' , " : *b $\in$ (*b $\in$ *b $\rightarrow$ *b) $\rightarrow$ ((*a $\rightarrow$ *b) $\rightarrow$ ((*a)List $\rightarrow$ *b))") ;;

NEWAXIOMS(;;)

FreeMonoid " !Z: *b. !P: *b $\in$ *b $\rightarrow$ *b. !f: *a $\rightarrow$ *b. !l: (*a)List.
FreeMonoid(Z,P)f l == (Null l $\Rightarrow$ Z | P(f(Head l),FreeMonoid(Z,P)
f(Tail l : (*a)List)))"

FACT lfree

FreeMonUU " !Z: *b. !P: *b $\in$ *b $\rightarrow$ *b. !f: *a $\rightarrow$ *b. FreeMonoid
(Z,P)f (UU: (*a)List) == UU: *b"

FreeMonNil " !Z: *b. !P: *b $\in$ *b $\rightarrow$ *b. !f: *a $\rightarrow$ *b. FreeMonoid
(Z,P)f (Nil: (*a)List) == Z"

FreeMonCons " !Z: *b. !P: *b $\in$ *b $\rightarrow$ *b. !f: *a $\rightarrow$ *b. !a: *a. !l
: (*a)List. FreeMonoid(Z,P)f (Cons a l : (*a)List) == P(f a,FreeMonoid
(Z,P)f l)"

FreeMonUnit " !P: *b $\in$ *b $\rightarrow$ *b. !Z: *b. (!x: *b. P(x,Z) == x) IMP
(!f: *a $\rightarrow$ *b. !a: *a. FreeMonoid(Z,P)f (Unit a : (*a)List) ==
f a)"

FreeMonAppend " !P: *b $\in$ *b $\rightarrow$ *b. !Z: *b. !f: *a $\rightarrow$ *b. (!x: *b.
P(UU,x) == UU: *b) & (!x. P(Z,x) == x) & (!x. !y: *b. !z: *b.
P(P(x,y),z) == P(x,P(y,z))) IMP (!l1: (*a)List. !l2: (*a)List.
FreeMonoid(Z,P)f ((l1 @ l2): (*a)List) == P(FreeMonoid(Z,P)
f l1,FreeMonoid(Z,P)f l2))"

FreeMonUnique " !f: *a $\rightarrow$ *b. !G: (*a)List $\rightarrow$ *b. !Z: *b. !P: *b
 $\in$ *b $\rightarrow$ *b. f == \a: *a. G(Unit a) & G UU == UU: *b & G Nil
== Z & (!x: *b. P(UU,x) == UU: *b) & (!x. P(Z,x) == x) & (!x.
!y: *b. !z: *b. P(P(x,y),z) == P(x,P(y,z))) & (!x. P(x,Z) ==
x) & (!x1: (*a)List. !x2: (*a)List. G(x1 @ x2) == P(G x1,G x2)
) IMP (!l: (*a)List. FreeMonoid(Z,P)f l == G l)"

THEORY 11

```

newparent 'basic' ;;

newtype 1 'List1' ;;

newconstant ( 'absList1' , ":(.)u + (*a,((*a)List1)u)X → (*a)
  List1" ) ;;

newconstant ( 'repList1' , ":(*a)List1 → (.)u + (*a,((*a)List1)
  u)X" ) ;;

newconstant ( 'Nil' , ":(*a)List1" ) ;;

newconstant ( 'Cons' , ":(*a → ((*a)List1 → (*a)List1)" ) ;;

newconstant ( 'Null' , ":(*a)List1 → tr" ) ;;

newconstant ( 'isCons' , ":(*a)List1 → tr" ) ;;

newconstant ( 'Head' , ":(*a)List1 → *a" ) ;;

newconstant ( 'Tail' , ":(*a)List1 → (*a)List1" ) ;;

newconstant ( 'copyList1' , ":((*a)List1 → (*a)List1) → ((*a)
  List1 → (*a)List1)" ) ;;

NEWAXIOMS();;

absList1 "!(ABS:(*a)List1. absList1(repList1 ABS :(. )u + (*a,
  ((*a)List1)u)X) == ABS"

repList1 "!(REP:(.)u + (*a,((*a)List1)u)X. repList1(absList1
  REP :(*a)List1) == REP"

Nil "Nil == absList1(INL(UP ( ) :(. )u) :(. )u + (*a,((*a)List1)
  u)X) :(*a)List1"

Cons "!(a:*a. !l:(*a)List1. Cons a l == absList1(INR((a f (UP
  l :(*a)List1)u)):(*a,((*a)List1)u)X) :(. )u + (*a,((*a)List1)
  u)X) :(*a)List1"

Null "!(ABS:(*a)List1. Null ABS == ISL(repList1 ABS :(. )u +
  (*a,((*a)List1)u)X) :tr"

isCons "!(ABS:(*a)List1. isCons ABS == (ISL(repList1 ABS :(.
  )u + (*a,((*a)List1)u)X)=>FF|TT)"

```

```

Head  "!ABS:(*a)List1. Head ABS == P1(OUTR(repList1 ABS :(. )
      u + (*a,((*a)List1)u)X) :(*a,((*a)List1)u)X) :*a"

Tail  "!ABS:(*a)List1. Tail ABS == DOWN(P2(OUTR(repList1 ABS
      :(. )u + (*a,((*a)List1)u)X) :(*a,((*a)List1)u)X) :(((*a)List1)
      u) :(*a)List1"

copyList1  "!FUN:(*a)List1 -> (*a)List1. !ABS:(*a)List1. copyList1
      FUN ABS == (Null ABS=>Nil|Cons(Head ABS :*a)(FUN(Tail ABS)
      ) :(*a)List1)"

FIXList1  "!ABS:(*a)List1. FIX (copyList1:((*a)List1 -> (*a)
      List1) -> ((*a)List1 -> (*a)List1)) ABS == ABS"

FACT 11

constructList1  "Nil == Nil:(*a)List1 & (!a:*a. !l:(*a)List1.
      Cons a l == Cons a l :(*a)List1)"

selectList1  "(!ABS:(*a)List1. Head ABS == Head ABS :*a) & (
      !ABS. Tail ABS == Tail ABS :(*a)List1)"

coverList1  "!ABS:(*a)List1. (Null ABS=>Nil|Cons(Head ABS :*a)
      (Tail ABS :(*a)List1)) == ABS"

absList1UU  "absList1 (UU:(. )u + (*a,((*a)List1)u)X) == UU:(*a)
      List1"

repList1UU  "repList1 (UU:(*a)List1) == UU:(. )u + (*a,((*a)List1)
      u)X"

DEFabsList1  "!REP:(. )u + (*a,((*a)List1)u)X. DEF(absList1 REP
      :(*a)List1) == DEF REP :tr"

DEFrepList1  "!ABS:(*a)List1. DEF(repList1 ABS :(. )u + (*a,(
      (*a)List1)u)X) == DEF ABS :tr"

DEFNil  "DEF (Nil:(*a)List1) == TT"

DEFCons  "!a:*a. !l:(*a)List1. DEF(Cons a l :(*a)List1) == (DEF
      a=>TT|UU:tr)"

NullUU  "Null (UU:(*a)List1) == UU:tr"

isConsUU  "isCons (UU:(*a)List1) == UU:tr"

```

```

HeadUU  "Head (UU:(*a)List1) == UU:*a"

TailUU  "Tail (UU:(*a)List1) == UU:(*a)List1"

copyList1UU  "|FUN:(*a)List1 -> (*a)List1. copyList1 FUN (UU
              :(*a)List1) == UU:(*a)List1"

NullNil  "Null (Nil:(*a)List1) == TT"

NullIMPNil  "|l:(*a)List1. Null l == TT IMP l == Nil:(*a)List1"

NullIMPCons  "|l:(*a)List1. Null l == FF IMP l == Cons(Head l
              :(*a)(Tail l :(*a)List1) :(*a)List1"

NullCons  "|a:*a. l:(*a)List1. DEF a == TT IMP Null(Cons a l
              :(*a)List1) == FF"

TailCons  "|a:*a. l:(*a)List1. DEF a == TT IMP Tail(Cons a l
              :(*a)List1) == l"

HeadCons  "|a:*a. l:(*a)List1. Head(Cons a l :(*a)List1) ==
a"

UUCons  "|l:(*a)List1. Cons (UU:*a) l == UU:(*a)List1"

```

THEORY lfun1

```

newparent 'l1' ;;

newconstant ( 'Unit' , " :*a -> (*a)List1" ) ;;

newolinfix ( '@' , " :(*a)List1 & (*a)List1 -> (*a)List1" ) ;;

NEWAXIOMS( );;

Unit  "|a:*a. Unit a == Cons a (Nil:(*a)List1) :(*a)List1"

Append  "|l1:(*a)List1. l2:(*a)List1. l1 @ l2 == (Null l1=>l2|Cons
          (Head l1 :a)((Tail l1 :(*a)List1) @ l2):(*a)List1) :(*a)
          List1)"

```

FACT lfun1

```

UUApp  "l1:(*a)List1. (UU:(*a)List1) @ 1 == UU:(*a)List1"

NilApp  "l1:(*a)List1. (Nil:(*a)List1) @ 1 == 1"

ConsApp "la:*a. l1:(*a)List1. l2:(*a)List1. (Cons a l1 :(*a)
  List1) @ l2 == Cons a((l1 @ l2):(*a)List1) :(*a)List1"

AppNil  "l1:(*a)List1. 1 @ (Nil:(*a)List1) == 1"

Assoc  "l1:(*a)List1. l2:(*a)List1. l3:(*a)List1. ((l1 @ l2)
  :(*a)List1) @ l3 == (l1 @ ((l2 @ l3):(*a)List1)):(*a)List1"

UnitUU  "Unit (UU:*a) == UU:(*a)List1"

ConsUnit "la:*a. l1:(*a)List1. Cons a 1 == ((Unit a :(*a)List1)
  @ 1):(*a)List1"

```

THEORY lfree1

```

newparent 'lfun1' ;;

newconstant ( 'FreeMonoid' , " :*b f (*b f *b -> *b) -> ((*a ->
  *b) -> ((*a)List1 -> *b))" ) ;;

NEWAXIOMS( );;

FreeMonoid "lZ:*b. lP:*b f *b -> *b. lf:*a -> *b. l1:(*a)List1.
  FreeMonoid(Z,P)f 1 == (Null l=>Z|P(f(Head l),FreeMonoid(Z,P)
  f(Tail l :(*a)List1)))"

```

FACT lfree1

```

FreeMonUU "lZ:*b. lP:*b f *b -> *b. lf:*a -> *b. FreeMonoid
  (Z,P)f (UU:(*a)List1) == UU:*b"

FreeMonNil "lZ:*b. lP:*b f *b -> *b. lf:*a -> *b. FreeMonoid
  (Z,P)f (Nil:(*a)List1) == Z"

FreeMonCons "lf:*a -> *b. lP:*b f *b -> *b. lZ:*b. f UU == UU
  :*b & (lx:*b. P(UU,x) == UU:*b) IMP (la:*a. l1:(*a)List1.
  FreeMonoid(Z,P)f(Cons a 1 :(*a)List1) == P(f a,FreeMonoid
  (Z,P)f 1))"

```

```
FreeMonUnit "lP:*b f *b -> *b. lZ:*b. lf:*a -> *b. (lx:*b. P
(x,Z) == x) & f UU == UU:*b IMP (la:*a. FreeMonoid(Z,P)f(Unit
a :(*a)List1) == f a)"
```

```
FreeMonAppend "lP:*b f *b -> *b. lZ:*b. lf:*a -> *b. (lx:*b.
P(UU,x) == UU:*b) & (lx. P(Z,x) == x) & f UU == UU:*b & (
lx. ly:*b. lz:*b. P(P(x,y),z) == P(x,P(y,z))) IMP (l1:(*a)
List1. l2:(*a)List1. FreeMonoid(Z,P)f((l1 @ l2):(*a)List1)
== P(FreeMonoid(Z,P)f l1,FreeMonoid(Z,P)f l2))"
```

```
FreeMonUnique "lf:*a -> *b. lG:(*a)List1 -> *b. lZ:*b. lP:*b
f *b -> *b. f == \a:*a.G(Unit a) & G UU == UU:*b & G Nil
== Z & (lx:*b. P(UU,x) == UU:*b) & (lx. P(Z,x) == x) & (lx.
ly:*b. lz:*b. P(P(x,y),z) == P(x,P(y,z))) & (lx. P(x,Z) ==
x) & (xl:(*a)List1. lx2:(*a)List1. G(xl @ x2) == P(G xl,G
x2)) IMP (l1:(*a)List1. FreeMonoid(Z,P)f l == G l)"
```

THEORY 12

```
newparent 'basic' ;;
```

```
newtype 1 'List2' ;;
```

```
newconstant ( 'absList2' , ":(.)u + (*a,(*a)List2)X -> (*a)List2" ) ;;
```

```
newconstant ( 'repList2' , ":(*a)List2 -> (.)u + (*a,(*a)List2)
X" ) ;;
```

```
newconstant ( 'Nil' , ":(*a)List2" ) ;;
```

```
newconstant ( 'Cons' , ":*a -> ((*a)List2 -> (*a)List2)" ) ;;
```

```
newconstant ( 'Null' , ":(*a)List2 -> tr" ) ;;
```

```
newconstant ( 'isCons' , ":(*a)List2 -> tr" ) ;;
```

```
newconstant ( 'Head' , ":(*a)List2 -> *a" ) ;;
```

```
newconstant ( 'Tail' , ":(*a)List2 -> (*a)List2" ) ;;
```

```
newconstant ( 'copyList2' , ":((*a)List2 -> (*a)List2) -> ((*a)
List2 -> (*a)List2)" ) ;;
```

```
NEWAXIOMS( );;
```

```

absList2  "!(ABS:(*a)List2. absList2(repList2 ABS :(. )u + (*a,
      (*a)List2)X) == ABS"

repList2  "!(REP:(. )u + (*a,(*a)List2)X. repList2(absList2 REP
      :(*a)List2) == REP"

Nil  "Nil == absList2(INL(UP ( ) :(. )u) :(. )u + (*a,(*a)List2)
      X) :(*a)List2"

Cons  "!(a:*a. !l:(*a)List2. Cons a l == absList2(INR((a l)
      :(*a,(*a)List2)X) :(. )u + (*a,(*a)List2)X) :(*a)List2"

Null  "!(ABS:(*a)List2. Null ABS == ISL(repList2 ABS :(. )u +
      (*a,(*a)List2)X) :tr"

isCons  "!(ABS:(*a)List2. isCons ABS == (ISL(repList2 ABS :(.
      )u + (*a,(*a)List2)X)=>FF|TT)"

Head  "!(ABS:(*a)List2. Head ABS == P1(OUTR(repList2 ABS :(. )
      u + (*a,(*a)List2)X) :(*a,(*a)List2)X) :*a"

Tail  "!(ABS:(*a)List2. Tail ABS == P2(OUTR(repList2 ABS :(. )
      u + (*a,(*a)List2)X) :(*a,(*a)List2)X) :(*a)List2"

copyList2  "!(FUN:(*a)List2 -> (*a)List2. !ABS:(*a)List2. copyList2
      FUN ABS == (Null ABS=>Nil|Cons(Head ABS :*a)(FUN(Tail ABS)
      ) :(*a)List2)"

FIXList2  "!(ABS:(*a)List2. FIX (copyList2:((*a)List2 -> (*a)
      List2) -> ((*a)List2 -> (*a)List2)) ABS == ABS"

```

FACT 12

```

constructList2  "Nil == Nil:(*a)List2 & !(a:*a. !l:(*a)List2.
      Cons a l == Cons a l :(*a)List2)"

selectList2  "!(ABS:(*a)List2. Head ABS == Head ABS :*a) & (
      !ABS. Tail ABS == Tail ABS :(*a)List2)"

coverList2  "!(ABS:(*a)List2. (Null ABS=>Nil|Cons(Head ABS :*a)
      (Tail ABS :(*a)List2)) == ABS"

absList2UU  "absList2 (UU:(. )u + (*a,(*a)List2)X) == UU:(*a)
      List2"

repList2UU  "repList2 (UU:(*a)List2) == UU:(. )u + (*a,(*a)List2)
      X"

```



```

DEFabsList2  "!REP:(.)u + (*a,(*a)List2)X. DEF(absList2 REP
:(*a)List2) == DEF REP :tr"

DEPrepList2  "!ABS:(*a)List2. DEF(repList2 ABS :(.)u + (*a,(*a)
List2)X) == DEF ABS :tr"

DEFNil  "DEF (Nil:(*a)List2) == TT"

DEFCons  "!a:*a. !l:(*a)List2. DEF(Cons a l :(*a)List2) == (DEF
a=>DEF l|UU:tr)"

NullUU  "Null (UU:(*a)List2) == UU:tr"

isConsUU  "isCons (UU:(*a)List2) == UU:tr"

HeadUU  "Head (UU:(*a)List2) == UU:*a"

TailUU  "Tail (UU:(*a)List2) == UU:(*a)List2"

copyList2UU  "!FUN:(*a)List2 -> (*a)List2. copyList2 FUN (UU
:(*a)List2) == UU:(*a)List2"

NullNil  "Null (Nil:(*a)List2) == TT"

NullIMPNil  "!l:(*a)List2. Null l == TT IMP l == Nil:(*a)List2"

NullIMPCons  "!l:(*a)List2. Null l == FF IMP l == Cons(Head l
:*a)(Tail l :(*a)List2) :(*a)List2"

DEPaDEF1NullCons  "!a:*a. !l:(*a)List2. DEF a == TT & DEF l ==
TT IMP Null(Cons a l :(*a)List2) == FF"

HeadCons  "!a:*a. !l:(*a)List2. DEF l == TT IMP Head(Cons a l
:(*a)List2) == a"

TailCons  "!a:*a. !l:(*a)List2. DEF a == TT IMP Tail(Cons a l
:(*a)List2) == l"

UUCons  "!l:(*a)List2. Cons (UU:*a) l == UU:(*a)List2"

ConsUU  "!a:*a. Cons a (UU:(*a)List2) == UU:(*a)List2"

```

THEORY lfun2

```

newparent 'l2' ;;

newconstant ( 'Unit' , " :*a -> (*a)List2" ) ;;

newolinfix ( 'e' , " :(*a)List2 f (*a)List2 -> (*a)List2" ) ;;

NEWAXIOMS(;;)

Unit  " !a:*a. Unit a == Cons a (Nil:(*a)List2) :(*a)List2"

Append  " !l1:(*a)List2. !l2:(*a)List2. l1 @ l2 == (Null l1=>l2|Cons
  (Head l1 :*a)((Tail l1 :(*a)List2) @ l2):(*a)List2) :(*a)
  List2)"

```

FACT lfun2

```

UUApp  " !l:(*a)List2. (UU:(*a)List2) @ l == UU:(*a)List2"

NilApp  " !l:(*a)List2. (Nil:(*a)List2) @ l == l"

ConsApp  " !a:*a. !l1:(*a)List2. !l2:(*a)List2. (Cons a l1 :(*a)
  List2) @ l2 == Cons a((l1 @ l2):(*a)List2) :(*a)List2"

AppUU  " !l:(*a)List2. l @ (UU:(*a)List2) == UU:(*a)List2"

AppNil  " !l:(*a)List2. l @ (Nil:(*a)List2) == l"

Assoc  " !l1:(*a)List2. !l2:(*a)List2. !l3:(*a)List2. ((l1 @ l2)
  :(*a)List2) @ l3 == (l1 @ ((l2 @ l3):(*a)List2)):(*a)List2"

DEFApp  " !l1:(*a)List2. !l2:(*a)List2. DEF((l1 @ l2):(*a)List2)
  == (DEF l1=>DEF l2|UU:tr)"

UnitUU  "Unit (UU:*a) == UU:(*a)List2"

ConsUnit  " !a:*a. !l:(*a)List2. Cons a l == ((Unit a :(*a)List2)
  @ l):(*a)List2"

```

THEORY lfree2

```
newparent 'lfun2' ;;
```

```
newconstant ( 'FreeMonoid' , " :*b f (*b f *b -> *b) -> ((*a ->
  *b) -> ((*a)List2 -> *b))" ) ;;
```

```
NEWAXIOMS( );;
```

```
FreeMonoid " !Z:*b. !P:*b f *b -> *b. !f:*a -> *b. !l:(*a)List2.
  FreeMonoid(Z,P)f l == (Null l=>Z|P(f(Head l),FreeMonoid(Z,P)
  f(Tail l :(*a)List2)))"
```

FACT lfree2

```
FreeMonUU " !Z:*b. !P:*b f *b -> *b. !f:*a -> *b. FreeMonoid
  (Z,P)f (UU:(*a)List2) == UU:*b"
```

```
FreeMonNil " !Z:*b. !P:*b f *b -> *b. !f:*a -> *b. FreeMonoid
  (Z,P)f (Nil:(*a)List2) == Z"
```

```
FreeMonCons " !P:*b f *b -> *b. !f:*a -> *b. !Z:*b. (!x:*b. P
  (x,UU) == UU:*b) & f UU == UU:*b & (!x. P(UU,x) == UU:*b)
  IMP (!a:*a. !l:(*a)List2. FreeMonoid(Z,P)f(Cons a l :(*a)
  List2) == P(f a,FreeMonoid(Z,P)f l))"
```

```
FreeMonUnit " !P:*b f *b -> *b. !Z:*b. !f:*a -> *b. (!x:*b. P
  (x,Z) == x) & f UU == UU:*b IMP (!a:*a. FreeMonoid(Z,P)f(Unit
  a :(*a)List2) == f a)"
```

```
FreeMonAppend " !P:*b f *b -> *b. !Z:*b. !f:*a -> *b. (!x:*b.
  P(UU,x) == UU:*b) & (!x. P(Z,x) == x) & (!x. P(x,UU) == UU
  :*b) & (!x. !y:*b. !z:*b. P(P(x,y),z) == P(x,P(y,z))) IMP
  (!l1:(*a)List2. !l2:(*a)List2. FreeMonoid(Z,P)f((l1 @ l2)
  :(*a)List2) == P(FreeMonoid(Z,P)f l1,FreeMonoid(Z,P)f l2)
  )"

```

```
FreeMonUnique " !f:*a -> *b. !G:(*a)List2 -> *b. !Z:*b. !P:*b
  f *b -> *b. f == \a:*a.G(Unit a) & G UU == UU:*b & G Nil
  == Z & (!x:*b. P(x,UU) == UU:*b) & (!x. P(UU,x) == UU:*b)
  & (!x1:(*a)List2. !x2:(*a)List2. G(x1 @ x2) == P(G x1,G x2)
  ) IMP (!l:(*a)List2. FreeMonoid(Z,P)f l == G l)"
```

THEORY eqfun

newparent 'basic' ;;

FACT eqfun

GenTrans " $\lambda EQ: * \rightarrow (* \rightarrow tr). \lambda x: *. \lambda y: *. \lambda z: *. \lambda t: tr. (\lambda x1: *. \lambda x2: *. \lambda x3: *. EQ\ x1\ x2 == TT \ \& \ EQ\ x2\ x3 == TT \ IMP \ EQ\ x1\ x3 == TT) \ \& \ (\lambda x1. \lambda x2. EQ\ x1\ x2 == EQ\ x2\ x1) \ \& \ (\lambda x1. \lambda x2. DEF(EQ\ x1\ x2) == (DEF\ x1) \ and \ (DEF\ x2)) \ \& \ EQ\ x\ y == TT \ \& \ EQ\ y\ z == t \ IMP \ EQ\ x\ z == t$ "

EQxUU " $\lambda EQ: * \rightarrow (* \rightarrow tr). \lambda x: *. (\lambda x1: *. \lambda x2: *. DEF(EQ\ x1\ x2) == (DEF\ x1) \ and \ (DEF\ x2)) \ IMP \ EQ\ x\ UU == UU:tr$ "

EQUUX " $\lambda EQ: * \rightarrow (* \rightarrow tr). \lambda x: *. (\lambda x1: *. \lambda x2: *. DEF(EQ\ x1\ x2) == (DEF\ x1) \ and \ (DEF\ x2)) \ IMP \ EQ\ UU\ x == UU:tr$ "

SelfEQCong " $\lambda EQ: * \rightarrow (* \rightarrow tr). \lambda x: *. \lambda x': *. \lambda y: *. \lambda y': *. (\lambda x1: *. \lambda x2: *. \lambda x3: *. EQ\ x1\ x2 == TT \ \& \ EQ\ x2\ x3 == TT \ IMP \ EQ\ x1\ x3 == TT) \ \& \ (\lambda x1. \lambda x2. EQ\ x1\ x2 == EQ\ x2\ x1) \ \& \ (\lambda x1. \lambda x2. DEF(EQ\ x1\ x2) == (DEF\ x1) \ and \ (DEF\ x2)) \ \& \ EQ\ x\ x' == TT \ \& \ EQ\ y\ y' == TT \ IMP \ EQ\ x\ y == EQ\ x'\ y'$ "

THEORY atom

newparent 'eqfun' ;;

newolcinfix ('EqAt' , " : *At $\rightarrow$ (*At $\rightarrow$ tr)") ;;

FACT atom

```

GenTrans  " !x:*At. !y:*At. !z:*At. !t:tr. (!x1:*At. !x2:*At.
           !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1 EqAt x3
           == TT) & (!x1. !x2. x1 EqAt x2 == (x2 EqAt x1):tr) & (!x1.
           !x2. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & x EqAt
           y == TT & y EqAt z == t IMP x EqAt z == t"

EqAtUU    " !x:*At. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) ==
           (DEF x1) and (DEF x2)) IMP x EqAt (UU:*At) == UU:tr"

UUEqAt    " !x:*At. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) ==
           (DEF x1) and (DEF x2)) IMP (UU:*At) EqAt x == UU:tr"

EqAtSelfCongruence " !x:*At. !x':*At. !y:*At. !y':*At. (!x1:*At.
           !x2:*At. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP
           x1 EqAt x3 == TT) & (!x1. !x2. x1 EqAt x2 == (x2 EqAt x1)
           :tr) & (!x1. !x2. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF
           x2)) & x EqAt x' == TT & y EqAt y' == TT IMP x EqAt y ==
           (x' EqAt y'):tr"

```

THEORY la

```

newparent 'atom' ;;

newtype 1 'L' ;;

newconstant ( 'absL' , ":(.)u + (*At,(*At)L)X -> (*At)L" ) ;;

newconstant ( 'repL' , ":(*At)L -> (.)u + (*At,(*At)L)X" ) ;;

newconstant ( 'Nil' , ":(*At)L" ) ;;

newconstant ( 'Cons' , ":(*At -> ((*At)L -> (*At)L)" ) ;;

newconstant ( 'Null' , ":(*At)L -> tr" ) ;;

newconstant ( 'isCons' , ":(*At)L -> tr" ) ;;

newconstant ( 'H' , ":(*At)L -> *At" ) ;;

newconstant ( 'T' , ":(*At)L -> (*At)L" ) ;;

newconstant ( 'copyL' , ":((*At)L -> (*At)L) -> ((*At)L -> (*At)
           L)" ) ;;

NEWAXIOMS(;;)

```

```

absL  "!(ABS:(*At)L. absL(repL ABS :(. )u + (*At,(*At)L)X) == ABS"

repL  "!(REP:(. )u + (*At,(*At)L)X. repL(absL REP :(*At)L) == REP"

Nil   "Nil == absL(INL(UP ( ) :(. )u) :(. )u + (*At,(*At)L)X) :(*At)
      L"

Cons  "!(a:*At. !l:(*At)L. Cons a l == absL(INR((a £ l):(*At,
      (*At)L)X) :(. )u + (*At,(*At)L)X) :(*At)L"

Null  "!(ABS:(*At)L. Null ABS == ISL(repL ABS :(. )u + (*At,(*At)
      L)X) :tr"

isCons "!(ABS:(*At)L. isCons ABS == (ISL(repL ABS :(. )u + (*At,
      (*At)L)X)=>FF|TT)"

H     "!(ABS:(*At)L. H ABS == P1(OUTR(repL ABS :(. )u + (*At,(*At)
      L)X) :(*At,(*At)L)X) :*At"

T     "!(ABS:(*At)L. T ABS == P2(OUTR(repL ABS :(. )u + (*At,(*At)
      L)X) :(*At,(*At)L)X) :(*At)L"

copyL "!(FUN:(*At)L -> (*At)L. !ABS:(*At)L. copyL FUN ABS ==
      (Null ABS=>Nil|Cons(H ABS :*At)(FUN(T ABS))) :(*At)L)"

FIXL  "!(ABS:(*At)L. FIX (copyL:((*At)L -> (*At)L) -> ((*At)L
      -> (*At)L)) ABS == ABS"

```

FACT 1a

```

constructL "Nil == Nil:(*At)L & !(a:*At. !l:(*At)L. Cons a l
      == Cons a l :(*At)L)"

selectL  "!(ABS:(*At)L. H ABS == H ABS :*At) & !(ABS. T ABS ==
      T ABS :(*At)L)"

coverL  "!(ABS:(*At)L. (Null ABS=>Nil|Cons(H ABS :*At)(T ABS
      :(*At)L)) == ABS"

absLUU  "absL (UU:(. )u + (*At,(*At)L)X) == UU:(*At)L"

repLUU  "repL (UU:(*At)L) == UU:(. )u + (*At,(*At)L)X"

DEFabsL "!(REP:(. )u + (*At,(*At)L)X. DEF(absL REP :(*At)L) ==
      DEF REP :tr"

```

```

DEFrepL  "!(ABS:(*At)L. DEF(repL ABS :(. )u + (*At,(*At)L)X) ==
          DEF ABS :tr"

DEFNil   "DEF (Nil:(*At)L) == TT"

DEFCons  "!(a:*At. !l:(*At)L. DEF(Cons a l :(*At)L) == (DEF a=>DEF
          l|UU:tr)"

NullUU   "Null (UU:(*At)L) == UU:tr"

isConsUU "isCons (UU:(*At)L) == UU:tr"

HUU      "H (UU:(*At)L) == UU:*At"

TUU      "T (UU:(*At)L) == UU:(*At)L"

copyLUU  "!(FUN:(*At)L -> (*At)L. copyL FUN (UU:(*At)L) == UU
          :(*At)L"

HNil     "H (Nil:(*At)L) == UU:*At"

TNil     "T (Nil:(*At)L) == UU:(*At)L"

HCons    "!(a:*At. !l:(*At)L. DEF l == TT IMP H(Cons a l :(*At)
          L) == a"

TCons    "!(a:*At. !l:(*At)L. DEF a == TT IMP T(Cons a l :(*At)
          L) == l"

ConsUU   "!(a:*At. Cons a (UU:(*At)L) == UU:(*At)L"

UUCons   "!(l:(*At)L. Cons (UU:*At) l == UU:(*At)L"

NullIMPNil "!(l:(*At)L. Null l == TT IMP l == Nil:(*At)L"

NullIMPCons "!(l:(*At)L. Null l == FF IMP l == Cons(H l :(*At)
          (T l :(*At)L) :(*At)L)"

NullIMPUU "!(l:(*At)L. Null l == UU:tr IMP l == UU:(*At)L"

DEFNull  "!(l:(*At)L. DEF(Null l :tr) == DEF l :tr"

NullNil  "Null (Nil:(*At)L) == TT"

NullCons "!(l:(*At)L. !a:*At. Null(Cons a l :(*At)L) == not(
          (DEF a) and (DEF l))"

```

THEORY lafun

```

newparent 'la' ;;

newolinfix ( 'EqBa' , ":(*At)L @ (*At)L -> tr" ) ;;

newolinfix ( 'IsIn' , ":(*At)L @ (*At)L -> tr" ) ;;

newolinfix ( 'Minus' , ":(*At)L @ *At -> (*At)L" ) ;;

newolinfix ( '@' , ":(*At)L @ (*At)L -> (*At)L" ) ;;

newconstant ( 'Unit' , ":(*At)L -> (*At)L" ) ;;

NEWAXIOMS();;

Unit "la:*At. Unit a == Cons a (Nil:(*At)L) :(*At)L"

App "l1:(*At)L. l2:(*At)L. l1 @ l2 == (Null l1=>Nil|Cons(H
  l1 : *At)((T l1 :(*At)L) @ l2):(*At)L) :(*At)L)"

EqBa "l1:(*At)L. l2:(*At)L. l1 EqBa l2 == (Null l1=>Null l2|
  ((H l1 : *At) IsIn l2) and ((T l1 :(*At)L) EqBa ((l1 Minus
    (H l1 : *At)):(*At)L)))"

IsIn "la:*At. l1:(*At)L. a IsIn l == ((DEF a) and (Null l)=>FF|
  (a EqAt (H l : *At)) or (a IsIn (T l :(*At)L)))"

Minus "l1:(*At)L. la:*At. l Minus a == ((DEF a) and (Null l)
  =>Nil|((a EqAt (H l : *At)=>T l|Cons(H l : *At)((T l :(*At)
    L) Minus a):(*At)L) :(*At)L)))"

```

FACT lafun

```

NilApp "l2:(*At)L. (Nil:(*At)L) @ l2 == l2"

AppNil "l1:(*At)L. l @ (Nil:(*At)L) == l"

UUApp "l2:(*At)L. (UU:(*At)L) @ l2 == UU:(*At)L"

ConsApp "la:*At. l1':(*At)L. l1:(*At)L. (Cons a l :(*At)L) @
  l' == Cons a (l @ l'):(*At)L) :(*At)L"

AssocApp "l2:(*At)L. l3:(*At)L. l1:(*At)L. ((l1 @ l2):(*At)
  L) @ l3 == (l1 @ (l2 @ l3):(*At)L):(*At)L"

```



```

AppUU  "l1:(*At)L. l @ (UU:(*At)L) == UU:(*At)L"

ConsUnit  "la:*At. l1:(*At)L. Cons a l == ((Unit a :(*At)L) @
  l):(*At)L"

DEFUnit  "la:*At. DEF(Unit a :(*At)L) == DEF a :tr"

DEFApp  "l1l2:(*At)L. l1l2:(*At)L. DEF((l1 @ l2):(*At)L) == (DEF
  l1) and (DEF l2)"

DEFMinus  "(lx:*At. ly:*At. DEF((x EqAt y):tr) == (DEF x) and
  (DEF y)) IMP (l1:(*At)L. la:*At. DEF((l Minus a):(*At)L) ==
  (DEFa) and (DEF l))"

IsInCons  "(lx1:*At. lx2:*At. DEF((x1 EqAt x2):tr) == (DEFx1)
  and (DEF x2)) IMP (l1:(*At)L. la':*At. la:*At. a IsIn (Cons a'
  l :(*At)L) == (DEF l) and ((a EqAt a') or (a IsIn l)))"

IsInNil  "la:*At. a IsIn (Nil:(*At)L) == (DEF a=>FF|UU:tr)"

UUIsIn  "l1:(*At)L. (UU:*At) IsIn l == UU:tr"

IsInUU  "la:*At. a IsIn (UU:(*At)L) == UU:tr"

DEFIsIn  "(lx:*At. ly:*At. DEF((x EqAt y):tr) == (DEF x) and
  (DEF y)) IMP (la:*At. l1:(*At)L. DEF((a IsIn l):tr) == (DEF
  a) and (DEF l))"

UUEqBA  "l1l2:(*At)L. (UU:(*At)L) EqBA l2 == UU:tr"

NilEqBA  "l1l2:(*At)L. (Nil:(*At)L) EqBA l2 == Null l2 :tr"

ConsEqBA  "la:*At. l1:(*At)L. l1':(*At)L. (Cons a l :(*At)L)
  EqBA l' == ((DEF l) and (a IsIn l')) and (l EqBA ((l' Minus
  a):(*At)L))"

EqBAUU  "l1l2:(*At)L. l1 EqBA (UU:(*At)L) == UU:tr"

DEFEqBA  "(lx:*At. ly:*At. DEF((x EqAt y):tr) == (DEF x) and
  (DEF y)) IMP (l1l2:(*At)L. l1l2:(*At)L. DEF((l1 EqBA l2):tr)
  == (DEF l1) and (DEF l2))"

UUMinus  "la:*At. (UU:(*At)L) Minus a == UU:(*At)L"

MinusUU  "l1:(*At)L. l Minus (UU:*At) == UU:(*At)L"

NilMinus  "la:*At. (Nil:(*At)L) Minus a == (DEF a=>Nil|UU:(*At)
  L)"

```

ConsMinus "(!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEFx1) and (DEF x2)) IMP (!a:*At. !l:(*At)L. !a':*At. (Cons a' 1 :(*At)L) Minus a == (a' EqAt a=>1|Cons a'((1 Minus a):(*At)L):(*At)L))"

T4p17 "!a:*At. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) IMP (!l1:(*At)L. !l2:(*At)L. DEF l2 == TTIMP a IsIn ((l1 @ l2):(*At)L) == (a IsIn l1) or (a IsIn l2))"

T4p18 "!a:*At. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & (!x:*At. !y:*At. x EqAt y == (y EqAt x) :tr) IMP (!l1:(*At)L. !l2:(*At)L. a IsIn l1 == TT IMP ((l1 @ l2):(*At)L) Minus a == (((l1 Minus a):(*At)L) @ l2):(*At)L)"

T4p6 "(!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1)and (DEF x2)) & (!x:*At. x EqAt x == DEF x :tr) IMP (!l1:(*At)L. l1 EqBA l1 == DEF l1 :tr)"

T4p10 "!a:*At. !a':*At. (!x1:*At. !x2:*At. DEF((x1 EqAt x2) :tr) == (DEF x1) and (DEF x2)) & (!x1. !x2. !x3:*At. x1 EqAt x2== TT & x2 EqAt x3 == TT IMP x1 EqAt x3 == TT) & (!x1. !x2. x1EqAt x2 == (x2 EqAt x1):tr) & (!x:*At. x EqAt x == DEF x :tr)IMP (!l:(*At)L. a EqAt a' == TT IMP a IsIn l == (a' IsIn l):tr)"

fact15 "!a':*At. !a:*At. !l:(*At)L. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & a' EqAt a == TT IMP(Cons a' 1 :(*At)L) Minus a == 1"

fact16 "!a:*At. !l':(*At)L. !l:(*At)L. a IsIn l' == TT IMP(Cons a 1 :(*At)L) EqBA l' == (1 EqBA ((l' Minus a):(*At)L)):tr"

T4p19 "!a:*At. !l2:(*At)L. (!x1:*At. !x2:*At. DEF((x1 EqAtx2) :tr) == (DEF x1) and (DEF x2)) & (!x:*At. !y:*At. x EqAt y== (y EqAt x):tr) IMP (!l1:(*At)L. a IsIn l1 == FF IMP ((l1 @l2) :(*At)L) Minus a == (l1 @ ((l2 Minus a):(*At)L)):(*At)L)"

T423 "!a:*At. !a':*At. !l:(*At)L. !l':(*At)L. (!x1:*At. !x2 :*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & (!x :*At. !y:*At. x EqAt y == (y EqAt x):tr) & a EqAt a' == TT & l EqBA l' == TT IMP (Cons a 1 :(*At)L) EqBA (Cons a' l' :(*At)L) == TT"

T4p13 "!a:*At. !a':*At. (!x1:*At. !x2:*At. DEF((x1 EqAt x2) :tr) == (DEF x1) and (DEF x2)) & (!x1. !x2. !x3:*At. x1 EqAt x2== TT & x2 EqAt x3 == TT IMP x1 EqAt x3 == TT) & (!x1. !x2. x1EqAt x2 == (x2 EqAt x1):tr) & (!x:*At. x EqAt x == DEF x :tr)IMP (!l:(*At)L. a EqAt a' == TT IMP l Minus a == (1 Minus a'):(*At)L)"

T4p11 "(!x1:*At. !x2:*At. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1EqAt x3 == TT) & (!x1. !x2. x1 EqAt x2 == (x2 EqAt x1):tr) & (!x1. !x2. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) IMP (!a:*At. !a':*At. !l2:(*At)L. a EqAt a' == FF IMP a IsIn ((l2 Minus a'):(*At)L) == (a IsIn l2):tr)"

T4p12 "(!x1:*At. !x2:*At. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1 EqAt x3 == TT) & (!x:*At. x EqAt x == DEF x :tr) & (!x1. !x2. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & (!x1. !x2. x1 EqAt x2 == (x2 EqAt x1):tr) IMP (!a:*At. !l1:(*At)L. !l2:(*At)L. l1 EqBA l2 == TT IMP a IsIn l1 == (a IsIn l2):tr)"

T4p20 "(!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & (!x:*At. !y:*At. x EqAt y == (y EqAt x):tr) IMP (!l2':(*At)L. !l2:(*At)L. !l1':(*At)L. !l1:(*At)L. l1 EqBA l1' == TT & l2 EqBA l2' == TT IMP ((l1 @ l2):(*At)L) EqBA ((l1' @ l2'):(*At)L) == TT)"

fact28 "!a:*At. !a':*At. !l:(*At)L. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & a EqAt a' == FF IMP(Cons a l :(*At)L) Minus a' == Cons a((l Minus a'):(*At)L) :(*At)L)"

fact26 "!a:*At. !a':*At. !l:(*At)L. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & a EqAt a' == FF IMPa IsIn (Cons a' l :(*At)L) == (a IsIn l):tr"

T4p14 "!a:*At. !b:*At. (!x1:*At. !x2:*At. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1 EqAt x3 == TT) & (!x1. !x2. x1 EqAt x2 == (x2 EqAt x1):tr) & (!x:*At. x EqAt x == DEF x :tr) & (!x1. !x2. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) IMP(!l:(*At)L. ((l Minus a):(*At)L) Minus b == (((l Minus b):(*At)L) Minus a):(*At)L)"

fact23 "!a:*At. (!x:*At. x EqAt x == DEF x :tr) & (!x1:*At. !x2:*At. x1 EqAt x2 == (x2 EqAt x1):tr) & (!x1. !x2. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1 EqAt x3 == TT) & (!x1. !x2. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) IMP (!l1:(*At)L. !l2:(*At)L. a IsIn l1 == TT & a IsIn l2 == TT IMP l1 EqBA l2 == (((l1 Minus a):(*At)L) EqBA ((l2 Minus a):(*At)L)):tr)"

T4p8 "(!x:*At. !y:*At. x EqAt y == (y EqAt x):tr) & (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) IMP (!l':(*At)L. !a:*At. !l:(*At)L. l EqBA (Cons a l' :(*At)L) == ((DEF l') and (a IsIn l)) and (((l Minus a):(*At)L) EqBA l'))"

fact9 "(!a:*At. !a':*At. !l:(*At)L. (!x:*At. x EqAt x == DEF x :tr) & (!x. !y:*At. x EqAt y == (y EqAt x):tr) & (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) IMP (Consa(Cons a' 1 :(*At)L) :(*At)L) EqBA (Cons a'(Cons a 1 :(*At)L):(*At)L) == ((DEF a) and (DEF a')) and (DEF 1)"

T4p21 "(!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1)and (DEF x2)) & (!x:*At. !y:*At. x EqAt y == (y EqAt x):tr) & (!x1. !x2. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1EqAt x3 == TT) & (!x. x EqAt x == DEF x :tr) IMP (!l:(*At)L. !a:*At. !a':*At. a EqAt a' == TT & a IsIn 1 == TT IMP 1 EqBA (((1 Minus a):(*At)L) @ (Unit a' :(*At)L)):(*At)L) == TT)"

T4p15 "(!a:*At. !a':*At. (!x1:*At. !x2:*At. DEF((x1 EqAt x2) :tr) == (DEF x1) and (DEF x2)) & (!x1. !x2. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1 EqAt x3 == TT) & (!x1. !x2. x1EqAt x2 == (x2 EqAt x1):tr) & (!x:*At. x EqAt x == DEF x :tr)IMP (!l:(*At)L. !l':(*At)L. a EqAt a' == TT & 1 EqBA 1' == TTIMP ((1 Minus a):(*At)L) EqBA ((1' Minus a') :(*At)L) == TT)"

fact29 "(!x:*At. x EqAt x == DEF x :tr) & (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & (!x1. !x2. !x3:*At. x1 EqAt x2 == TT & x2 EqAt x3 == TT IMP x1 EqAt x3 ==TT) & (!x1. !x2. x1 EqAt x2 == (x2 EqAt x1):tr) IMP (!l :(*At)L. !a:*At. !l1:(*At)L. !l2:(*At)L. 1 EqBA ((11 Minus a):(*At)L) == TT & 11 EqBA 12 == TT IMP 1 EqBA ((12 Minus a) :(*At)L) == TT)"

T4p16 "(!x1:*At. !x2:*At. !x3:*At. x1 EqAt x2 == TT & x2 EqAtx3 == TT IMP x1 EqAt x3 == TT) & (!x:*At. x EqAt x == DEF x :tr) & (!x1. !x2. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & (!x1. !x2. x1 EqAt x2 == (x2 EqAt x1):tr) IMP (!l1: (*At)L. !l2:(*At)L. !l3:(*At)L. 11 EqBA 12 == TT & 12 EqBA 13 == TT IMP 11EqBA 13 == TT)"

T4p9 "(!x:*At. !y:*At. x EqAt y == (y EqAt x):tr) & (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) IMP (!l1:(*At)L. !l2:(*At)L. 11 EqBA 12 == (12 EqBA 11):tr)"

T4p7c "!l1:(*At)L. !l2:(*At)L. !l3:(*At)L. (!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) and (DEF x2)) & (!x:*At. x EqAtx == DEF x :tr) IMP (((11 @ 12):(*At)L) @ 13):(*At)L) EqBA ((11 @ ((12 @ 13):(*At)L)):(*At)L) == (DEF 11) and ((DEF 12) and(DEF 13))"

T4p22 "(!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1)and (DEF x2)) & (!x:*At. x EqAt x == DEF x :tr) & (!x. !y:*At. x EqAt y == (y EqAt x):tr) & (!x1. !x2. !x3:*At. x1 EqAt x2 ==TT & x2 EqAt x3 == TT IMP x1 EqAt x3 == TT) IMP (!l1:(*At)L. !l2:(*At)L. ((11 @ 12):(*At)L) EqBA ((12 @ 11):(*At)L) == (DEF11) and (DEF 12))"

T4p7a " $(!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) \text{ and } (DEF x2)) \ \& \ (!x:*At. x EqAt x == DEF x :tr) \text{ IMP } (!l1:(*At) L. (((Nil:(*At)L) @ l1):(*At)L) EqBA l1 == DEF l1 :tr))$ "

T4p7b " $(!x1:*At. !x2:*At. DEF((x1 EqAt x2):tr) == (DEF x1) \text{ and } (DEF x2)) \ \& \ (!x:*At. x EqAt x == DEF x :tr) \text{ IMP } (!l1:(*At) L. ((l1 @ (Nil:(*At)L)):(*At)L) EqBA l1 == DEF l1 :tr))$ "

Appendix 4.

Selected ML definitions.

A number of ML definitions are now included for completeness. They are grouped together into three ML text files whose functions are briefly described below:-

basic.ml contains various useful tactics, derived inference rules and some other general-purpose functions.

resolve.ml contains the basic resolution function RESOLVE.

meta.ml contains the ML functionals METARULE and METATAC.

```
% basic.ml %
```

```
%-----%
%
%          GENERAL ML FUNCTIONS.
%
%-----%
```

```
do( map mlcinfix ``Then mod orelse`` )
;;
```

```
let f Then g = g o f
and (f orelse g) x = f(x) ? g(x)
and (a mod b) = (b = 0) => failwith 'mod'
                        | (a > 0) =>  a - (a/b)*b
                        |      (a - (a/b)*b) + b
;;
```

```
let pair x y      = (x,y)
and swop f x y    = (f y x )
and curry f x y   = f (x,y)
and uncurry f (x,y) = f x y
and dual f (a,b)  = f(b,a)      % dual = curry o swop o uncurry %
and delta f (a,b) = (f a,f b)
and eq(x,y)       = (x = y)
;;
```

```
%-----%
%
% Recursive versions of While and Until given below:-
%
% let While pred body arg =
%       if pred(arg) then (While pred body (body arg)) else arg
% ;;
%
% let Until pred body arg =
%       let res = body(arg) in
%       if pred(res) then res else (Until pred body res)
% ;;
%
%-----%
```

```
let While pred body arg =
  letref res = 'arg in
    if pred( res ) loop (res := body(res)) else res
;;
```

```

let Until pred body arg =
    letref res = body( arg ) in
        if pred( res ) then res loop (res := body(res))
;;

letrec power f n x = (n < 1) => x | power f (n-1) (f x)
;;

let sel n l = (hd(power tl (n-1) l)) ? failwith 'sel: Not Found'
;;

let divide pred l = itlist arbitrator l ([], [])
    where arbitrator x (inlst, outlst) =
        pred(x) => (x.inlst, outlst) | (inlst, x.outlst)
;;

let divfilter (rejectpred, pred) l = itlist filtrator l ([], [])
    where filtrator x (inlst, outlst) =
        (rejectpred(x) ? true) => (inlst, outlst) |
        pred(x) => (x.inlst, outlst) | (inlst, x.outlst)
;;

let quicksort (eqpred, ordpred) l = QS(l)

whererec QS l =
    (null l) => [] |
    (let h.t = l
     in
     let (l1, l2) =
         divfilter ( (λx. eqpred(x,h)) , (λx. ordpred(x,h)) ) t
     in
     QS(l1) @ [h] @ QS(l2)
    )
;;

let join tk1 tk2 = implode((explode tk1) @ (explode tk2))
;;

let deltrivbind bind = filter ($not o eq) bind
;;

let genunion eqpred l1 l2 = itlist geninsert l1 l2
    where geninsert x l = x. (gendelete x l)
    where gendelete x l = filter (λy. not(eqpred(x,y))) l
;;

%-----%
%
%          CONJUNCTION FUNCTIONS.
%
%-----%

```



```
% The following definitions assume the standard Formula Identification%
% convention (page 72 [LCF]).
```

```
letrec destconj1 fm =
  (let (Lfm, Rfm) = destconj fm in Lfm. (destconj1 Rfm)) ? [fm]
;; % Decomposes given formula into it's component conjuncts (if any)%
```

```
letrec DESTCONJL th = ((SEL1 th) . (DESTCONJL (SEL2 th))) ? [th]
;; % Decomposes given theorem into it's component conjuncts (if any)%
```

```
let CONJL th1 = itlist (curry CONJ) th1 AXTRUTH
;; % Forms the conjunction of the given list of theorems. %
```

```
%-----%
%
%               QUANTIFIER and VARIABLE FUNCTIONS.
%
%-----%
```

```
let isclosed fm = null (formfrees fm)
;;
```

```
let isopen fm = not (isclosed fm)
and allvars (fml, fm) = formlvars (fm.fml)
and globalfv (fml, fm) = formlfrees (fm.fml)
and fixedfv (fml, fm) = formlfrees fml
and localfv (fml, fm) = formfrees fm
;;
```

```
let passivefv fmlfm = subtract (fixedfv fmlfm , localfv fmlfm)
and mutualfv fmlfm = intersect (fixedfv fmlfm , localfv fmlfm)
and activefv fmlfm = subtract (localfv fmlfm , fixedfv fmlfm)
;;
```

```
let AllVars = destthm Then allvars
and GlobalFV = destthm Then globalfv
and FixedFV = destthm Then fixedfv
and LocalFV = destthm Then localfv
and PassiveFV = destthm Then passivefv
and MutualFV = destthm Then mutualfv
and ActiveFV = destthm Then activefv
;;
```

```
let IsClosed th = null (GlobalFV th)
and IsOpen th = not(null (GlobalFV th))
;;
```

```

let openhyp th = filter isopen (hyp th)
and closedhyp th = filter isclosed (hyp th)
;;

let destquant1 fm = dql( [], fm )

whererec dql( v1 , fm ) =
  ( let v , fm' = destquant fm in dql( v1 @ [v] , fm' ) ) ? (v1 , fm)
;;

let diffvars (q1, v1) = fst(itlist DiffOneVar q1 ([], v1))
  where DiffOneVar q (q1, v1) =
    let q' = variant(q, v1) in (q'.q1, q'.v1)
;;

let gendestquant1 v1 fm = gendql ([], v1, fm)
whererec
  gendql(q1, v1, fm) =
    ( let (qv, fm1) = destquant fm
      in
      let qv' = variant(qv,v1)
      in
      gendql(q1 @ [qv'], qv'.v1, (substinform [(qv', qv)] fm1))
    )
    ? (q1, fm)
;;

let genDESTQUANTL v1 th =
  let c = concl th in
    ( let thv1 = formlfrees(c.(hyp th))
      and q1 = (fst o destquant1) c
      in
      let q1' = diffvars(q1, v1 @ thv1)
      in
      (q1', (revitlist SPEC q1' th))
    ) ? ([], th)
;; % Quantifier stripping avoids free variables in the hypotheses %
% (or in v1). %

let DESTQUANTL th = genDESTQUANTL [] th
;;

let mkquant1 (v1, fm) = itlist (curry mkquant) v1 fm
;;

let MKQUANTL v1 th = (itlist GEN v1 th) ? failwith 'MKQUANTL'
;;

```

```

let GENL vl th = itlist GEN vl' th
  where vl' = intersect( subtract( vl
                                , formlfrees(hyp th)
                              )
                        , formfrees(concl th)
                      )
;;

let openq      = (snd o destquantl)
and closeq fm = mkquantl(formfrees fm, fm)
and quantvars = (fst o destquantl)
and OPENQ     = (snd o DESTQUANTL)
and CLOSEQ th = GENL (formfrees(concl th)) th
and QUANTVARS = (fst o DESTQUANTL)
;;

let ARRANGEQVS varl th = GENL varl (OPENQ th)
;;

%-----%
%
%          IMPLICATION FUNCTIONS.
%
%-----%

let PM = swop MP                                % PM ath impth = MP impth ath %
;;

let DISCHL thm = itlist DISCH (hyp thm) thm
;;

let CHARGE thm =
  (let assmth1 = ((map ASSUME) o destconjl o fst o destimp o concl) thm
   in
   revitlist PM assmth1 thm
  ) ? thm
;;

let DRESSTHM th =
  (OPENQ Then CHARGE
   Then OPENQ
   Then (\th'. itlist DISCH (openhyp th') th')
   Then CLOSEQ
  ) th
;; % Makes all closed antecedents into hypotheses. Good for simprules%

let TRYMP impth ath =
  (let afm = concl ath
   in
   let assmfml = (destconjl o fst o destimp o concl) impth
   in
   let afml, nonafml = divide (\fm. aconvform(afm, fm)) assmfml

```

```

in
  if (null afml) then fail else
    (let assmth1 = map ASSUME assmfml
     in
       let concth = revitlist PM assmth1 impth
       in
         let impth' = itlist DISCH (afm. nonafml) concth
         in
           MP impth' ath
    )
) ? failwith 'TRYMP'
;; % A generalised form of Modus Ponens. It tries to find an %
% antecedent to eliminate. As for MP, the first argument must %
% be an implication. %

let MPANY qimpth ath =
  let ql, impth = DESTQUANTL qimpth
  in
    GENL ql (TRYMP impth ath)
;;

let ADDHYP f th = MP (DISCH f th) (ASSUME f)
;; % Adds redundant hypotheses to already-proven theorems. %

%-----%
%
% GENERAL INFERENCE AND PPLAMBDA RELATED FUNCTIONS.
%
%-----%

letrec termsize tm =
  ( (let (Ltm, Rtm) = destcomb tm
   in
     termsize(Ltm) + termsize(Rtm)
   )
  ? (let tm1 = (snd o destabs) tm
   in
     1 + termsize(tm1)
   )
  ? 1
  )
;;

letrec formsize fm =
  ( (let fm1 = (snd o destquant) fm
   in
     1 + formsize(fm1)
   )
  ? (let (Ltm, Rtm) = (destequiv orelse destinequiv) fm
   in
     termsize(Ltm) + termsize(Rtm)
   )
  )

```

```

? (let (Lfm, Rfm) = (destimp orelse destconj) fm
  in
    formsize(Lfm) + formsize(Rfm)
)
? 1
)
;;

let existsaconv fm = exists (\fm'.aconvform(fm, fm'))
;;

let isatomic fm = isequiv(fm) or isinequiv(fm)
;;

let iscontr fm =
  (let a,b = (destequiv fm) ? (destinequiv fm)
   in
    let atok, aty = destconst a
    and btok      = (fst o destconst) b
    in
      ( (aty = ":tr")
        & (not(atok = btok))
        & ( (isequiv fm) or (btok = 'UU') )
      )
    ) ? false
;;

let DEPCASES tm (thtt, thuu) =
  let thff =
    CONTR (concl thtt)
    (TRANS ((SYM (ASSUME "DEF itm == FF")), AXDEF tm ))
  in
    CASES "DEF itm" (thtt, thff, thuu)
;;

let CONDCASES x w t ( TTth, FFth, UUth ) =
  let c, t1, t2 = destcond t
  in
    let case tv = SYM( SUBSOCCS [ [1], SYM( ASSUME "ic == itv" ) ]
                        ( CONDCONV "itv =) it1 | it2" )
    )
  in
    CASES c ( SUBST [case "TT", x] w TTth,
                SUBST [case "FF", x] w FFth,
                SUBST [case "UU:tr", x] w UUth
            )
;; % As given on page 98 of [LCF] %

```

```

let DERIVE th1 th =
  let hyp1 = hyp th
  in
  let disch1 = filter ( $\lambda$ th1. exists( $\lambda$ fm'. aconvform(fm1, fm')) hyp1
    where fm1 = concl th1
    )
    th1
  in
  itlist DERIVESTEP disch1 th

```

```

where DERIVESTEP alteration th =
  MP (DISCH (concl alteration) th) alteration
;;

```

```

%-----%
%
%          SIMPLIFICATION ORIENTED TACTICS.
%
%-----%

```

```

let CALCTAC ss1 (w, ss, asml) =
  let (gl, prf) = SIMPTAC (w, ss1, asml)
  in
  ((if null gl then [] else [ (fst(hd(gl))), ss, asml ]), prf)
;;

```

```

let CALCTAC' ss1 (w, ss, asml) =
  CALCTAC (ssunion ss1 ss) (w, ss, asml)
;;

```

```

let NEWCALCTAC ss1 (w, ss, asml) = SIMPTAC (w, ss1, asml)
;;

```

```

let NEWCALCTAC' ss1 (w, ss, asml) =
  SIMPTAC (w, (ssunion ss1 ss), asml)
;;

```

```

let FINDSSTAC check (w, ss, asml) =
  let tryssadd fm1 ss1 = (ssadd (check fm1) ss1) ? ss1
  in
  ([ (w, (itlist tryssadd asml ss), asml) ], hd)
;;

```

```

let GndSSRuleCheck th =
  (let (qv', th') = DESTQUANTL th
   in
   let (Ltm, Rtm) = (destequiv o concl) th
   in
   (aconvterm(Ltm, Rtm)) => fail |
   isconst(Rtm) => th |
   isconst(Ltm) => GENL qv' (SYM th') |
   fail
  )
;;

```

```
let FINDGNDSSSTAC g = FINDSSTAC (GndSSRuleCheck o ASSUME) g
;;
```

```
%-----%
%
%               TACTICS and TACTICALS.
%
%-----%
```

```
let CONJTAC (w, ss, asml) =
  (let (Lfm, Rfm) = destconj(w)
   in
    ( [(Lfm, ss, asml); (Rfm, ss, asml)]
      , (λ[Lth; Rth]. CONJ(Lth,Rth))
    )
  ) ? failwith 'CONJTAC'
;;
```

```
let CONJLTAC = REPEAT CONJTAC
;;
```

```
let IMPTAC (w, ss, asml) =
  (let asm,con = destimp w
   in
    let fm, conjfm = (destconj asm) ? (asm, "TRUTH")
    in
      ["!conjfm IMP !con", ss, fm.asml] , (DISCH fm) o hd
    ) ? failwith 'IMPTAC'
  )
;;
```

```
let IMPTAC' (w, ss, asml) =
  (let asm,con = destimp w
   in
    let fm, conjfm = (destconj asm) ? (asm, "TRUTH")
    in
      ( ["!conjfm IMP !con", (ssadd (ASSUME fm) ss), fm.asml]
        , (DISCH fm) o hd
      )
    )
  ) ? failwith 'IMPTAC''
;;
```

```
let TRUETAC (w, ss, asml) =
  (istruth w) => ([], λ(). AXTRUTH) | failwith 'TRUETAC'
;;
```

```

let CONTRTAC (w, ss, asml) =
  (let contr = find iscontr asml
   in
    [], λ(). CONTR w (ASSUME contr)
  ) ? failwith 'CONTRTAC'
;;

let ASSUMETAC (w, ss, asml) =
  (tryfind instasmpfn asml) ? failwith 'ASSUMETAC'

where instasmpfn fm =
  if (aconvform (w, fm)) then ([], λ(). ASSUME w) else
  if (isquant fm) then
    (let th = ((INST (formmatch (openq fm) w)) o OPENQ o ASSUME) fm
     in
      if aconvform(w, concl th) then ([], λ(). th) else fail
    )
  else fail
;;

let SYMTAC (w, ss, asml) =
  (let (ltm, rtm) = destequiv w
   in
    (["↑rtm = ↑ltm", ss, asml], SYM o hd)
  ) ? failwith 'SYMTAC'
;;

let TRIVTAC =
  TRUETAC ORELSE CONTRTAC
  ORELSE ASSUMETAC
  ORELSE (SYMTAC THEN ASSUMETAC)
;;

let ATOMTAC = REPEAT( GENTAC ORELSE IMPTAC ORELSE CONJTAC )
and ATOMTAC' = REPEAT( GENTAC ORELSE IMPTAC' ORELSE CONJTAC )
                THEN SIMPTAC
;;

let TAUTAC =
  (CALCTAC BASICSS) THEN ATOMTAC THEN (TRIVTAC ORELSE IDTAC)
and TAUTAC' =
  (CALCTAC BASICSS) THEN ATOMTAC' THEN (TRIVTAC ORELSE IDTAC)
;;

let EUREKATAC fm (w, ss, asml) =
  ( ["↑fm IMP ↑w", ss, asml] ; (fm, ss, asml) ]
  , λ[Lth; Rth]. MP Lth Rth
  )
;;

```



```

let (DISCHTAC fm :tactic) (w, ss, asml) =
  (let (isfml, isntfml) = divide (\fm'. aconvform(fm, fm')) asml
   in
    if (null isfml) then fail else
      ["↑fm IMP ↑w", ss, isntfml], λ[th]. MP th (ASSUME fm)
  ) ? failwith 'DISCHTAC'
;;

let (USELEMMASTAC lemmalst :tactic) (w, ss, asml) =
  if (forall (swop existsaconv asml) ((hyp o CONJL) lemmalst))
  then ( [w, ss, (map concl lemmalst) @ asml] , (DERIVE lemmalst) o hd )
  else failwith 'USELEMMASTAC'
;;

let (ABSURDTAC thm :tactic) =
  (USELEMMASTAC [thm]) THEN (DISCHTAC (concl thm)) THEN SIMPTAC
;;

let CANONTAC (w, ss, asml) =
  let (eqnfml, noteqnfml) = divide (isequiv o snd o destquant1) asml
  in
    (null noteqnfml) => ([ (w, ss, (eqnfml @ noteqnfml)) ], hd) |
    (let th1 = flat(map (HNF o ASSUME) noteqnfml)
     % See Section 5.4.6 for definition of HNF %
     in
      ([ (w, ss, eqnfml @ (map concl th1)) ], (DERIVE th1) o hd)
    )
  )
;;

let (DEFCASESTAC tm :tactic) (w, ss, asml) =
  let deftt = "DEF ↑tm == TT" and defuu = "DEF ↑tm == UU"
  in
    ( [ (w, (ssadd (ASSUME deftt) ss), deftt.asml)
      ; (w, (ssadd (DEFUU (ASSUME defuu)) ss), defuu.asml)
      ]
      , λ[caseTT; caseUU]. DEFCASES tm (caseTT, caseUU)
    )
  )
;;

let GENDEFCASESTAC (w, ss, asml) =
  (let (v, W) = destquant w
   in
    let v1 = variant(v, formlfrees(w.asml))
    in
    let W1 = substinform [v1,v] W
    in
    (I THEN (DEFCASESTAC v1)) ([ (W,ss,asml) ], (GEN v1) o hd)
  ) ? failwith 'GENDEFCASESTAC'
;;

```

```

let GENCASESTAC (w, ss, asml) =
  (let (v, W) = destquant w
   in
    (typeof v = ":tr") =)
    (let v1 = variant(v, formlfrees(w.asml))
     in
      let W1 = substinform [v1,v] W
      in
        (I THEN (CASESTAC v1)) ([ (W,ss,asml) ], (GEN v1) o hd)
    )
  | fail
  ) ? failwith 'GENCASESTAC'
;;

```

```

%-----%
%
%          AUXILIARY (proof related) ML FUNCTIONS.
%
%-----%

```

```

letref FailList = [] :goal list
;;

```

```

let Prove T g =
  let gl, p = T g
  in
    if (null gl) then p []
    else (FailList := gl ; failwith 'Tactical proof incomplete')
;;

```

```

let TAUTOLOGY ss fm = Prove SIMPTAC (fm, ss, [])
;;

```

```

let CheckTh1 th1 (w,ss,asml) =
  forall (swop existsaconv asml) ((hyp o CONJL) th1)
;;

```

```

let declvars ty tk1 = map (\tk. mkvar(tk,ty)) tk1
;;

```

```

let ssadd1 thmlst ss = itlist ssadd thmlst ss
;;

```

```

let record (tk1, thml) =
  map (can newfact) (combine(tk1, thml))
  ? failwith 'record'
;;

```

% 1st order Formula Unification function.

```
(unifyform bndvars instvars fmbind Lfm Rfm) :(term $ term) list
```

returns a most general unifier of Lfm with Rfm, if it exists. This is computed with respect to a context specified by the first three parameters. The parameters have the following meanings:-

```
bndvars    =    association list specifying bound variable info.
instvars   =    A list of all variables that can be matched.
fmbind     =    A "partial" unifier with which the result has
                to be consistent.
```

During the computation of unifier's, monic substitutions are used to accumulate and represent partial results. However, for compatibility with other LCF functions, the would-be result is converted to a simultaneous substitution.

%

% resolve.ml %

```
let RESOLVE Lth Rth =
```

```
((($not o isimp o openq o concl) Rth => fail |
```

```
  let (Rq1, Rth') = DESTQUANTL Rth
```

```
  in
```

```
  let Rvars = AllVars Rth
```

```
  in
```

```
  let (Lq1, Lth') = genDESTQUANTL Rvars Lth
```

```
  in
```

```
  let Ivars = Lq1 @ Rq1
```

```
  and Lvars' = AllVars Lth'
```

```
  and Lfm' = concl Lth'
```

```
  in
```

```
  let tryresolve (lth, lfm) ql fm =
```

```
    (let tmbind = unifyform [] (subtract(Ivars, ql)) [] lfm fm
```

```
    in
```

```
    let (Ltmbind, Rtmbind) = divide ((λv. mem v Lq1) o snd) tmbind
```

```
    in
```

```
      [ (CLOSEQ (TRYMP (INST Rtmbind Rth')
```

```
                (GENL ql (INST Ltmbind lth))
```

```
          )
```

```
        ]
```

```
      ]
```

```
    ) ? []
```

```
  in
```

```
  let Resolver =
```

```
    if (isequiv Lfm')
```

```

then (let SLth' = SYM Lth'      in
      let SLfm' = concl SLth'    in

       $\lambda$ (ql, fm).
        if (isequiv fm)
          then ( (tryresolve (Lth', Lfm') ql fm)
                @ (tryresolve (SLth', SLfm') ql fm)
                )
          else []

      )

else  $\lambda$ (ql, fm).
      if (isequiv fm)
        then []
        else (tryresolve (Lth', Lfm') ql fm)
in
  flat( map
    (Resolver o (gendestquantl Lvars'))
    ((destconjl o fst o destimp o concl) Rth')
  )
) ? []
;;

```

```
% meta.ml %
```

```
% Assumes basic.ml %
```

```
% Polymorphic matching functions (as used in METARULE and METATAC).
```

Suppose that we have a pattern term, *ptm*, and a match term, *mtm*. We say that the term *mtm* "polymorphically matches" the pattern, *ptm*, if and only if there exists a (unique, upto renaming) pair of type and term instantiations, (*tyinst*, *tminst*) such that:-

```
mtm = substinterm tminst (instinterm tyinst ptm)
```

Similar definitions can be made for formulae.

The functions below generally require more than just the argument pattern and the match. They may also take into account the "context" of the match as well by importing type and term bindings with which the results must be consistent, as well as information about variables already bound in the context (to allow for alpha-conversion).

Summary of matching functions.

Term matching (dependant upon context) is performed by the function:-

```
(PolyTermMatch bndvars tyin tmin ptm mtm)
:((type  $\mathcal{E}$  type) list  $\mathcal{E}$  (term  $\mathcal{E}$  term) list)
```

Formula matching (dependant upon context) is performed by the function:-

```
(PolyFormMatch bndvars tyin tmin pfm mfm)
:((type  $\mathcal{E}$  type) list  $\mathcal{E}$  (term  $\mathcal{E}$  term) list)
```

where the parameters mean:-

```
bndvars = association list for (lambda or quant.) bound
          variables
tyin     = imported type instantiation
tmin     = imported term instantiation
ptm      = pattern term
mtm      = match term
pfm      = pattern form
mfm      = match form
```

Matching functions which are not dependant upon context can be easily defined as follows:-

```
let polytermmatch pfm mfm = PolyTermMatch [] [] [] pfm mfm
and polyformmatch pfm mfm = PolyFormMatch [] [] [] pfm mfm
;;
```

Each of these functions were written in ML.

&

let METARULE th =

```

  (if (null o hyp) th) then
    (let th' = OPENQ th in
      let fm = concl th' in
      let (afm, cfm) = (destimp fm) ? ("TRUTH" , fm)
      in
        if (istruth afm) then (λ[.th']) else
          (let localtmfv = subtract( formfrees cfm, formfrees afm)
            and localtyfv = subtract( formtyvars cfm, formtyvars afm)
            in
              if (null localtmfv) & (null localtyfv) then
                (let AnteSize = (length o destconj) afm
                  in
                    let PRODUCETH th1 =
                      if (length th1 = AnteSize)
                      then
                        (let conjth = CONJL th1
                          in
                            let (tyinst, tminst) =
                              polyformmatch afm (concl conjth)
                              in
                                MP (INST tminst (INSTTYPE tyinst th')) conjth)
                        )
                      ? failwith FAILTOKEN1
                      else failwith FAILTOKEN2
                    in
                      PRODUCETH
                  )
                else failwith FAILTOKEN3
              )
            )
          else failwith FAILTOKEN4
    )

```

```

where FAILTOKEN1 = 'METARULE: Derived inference rule fails.'
and    FAILTOKEN2 = 'METARULE: Wrong number of parameters'
and    FAILTOKEN3 = 'METARULE: Free Variables unbound within assumptions'
and    FAILTOKEN4 = 'METARULE: Not a Sentence'
;;

```

```

let METATAC th =
  (if ((null o hyp) th) then
    (let th' = OPENQ th      in
     let fm = concl th'      in
     let (afm, cfm) = (destimp fm) ? ("TRUTH" , fm)
     in
     let (METAC (tyinst, tminst) :tactic) (w, ss, asml) =
       (let (tyinst', tminst') =
          (PolyFormMatch [] tyinst tminst cfm w)
          ? (failwith FAILTOKEN1)

          in
          let newth' = INST tminst' (INSTTYPE tyinst' th')
          in

          if istruth(afm)
          then (if aconvform(concl newth', w)
                 then ([], λ(). newth')
                 else failwith FAILTOKEN2
                )
          else (let afm', cfm' = (destimp o concl) newth'
                 in
                 if aconvform(cfm', w)
                 then ( (map (λw'. w', ss, asml) (destconjl afm'))
                        , (MP newth') o CONJL
                      )
                 else failwith FAILTOKEN2
                )
                )
          )
    in
    METAC
  )
  else failwith FAILTOKEN3
)
where FAILTOKEN1 = 'METATAC: Inappropriate Goal'
and   FAILTOKEN2 = 'METATAC: Bad Match'
and   FAILTOKEN3 = 'METATAC: input theorem must be a sentence.'
;;

```

```

%-----%
% Some simple tactics obtained as a simple inversion of inference
% rules. This assumes LCF theory 'kernel'.
%-----%

```

```

let MINTAC          = METATAC    (FACT 'kernel' 'MIN')      ([],[])
and SYNHTAC         = METATAC    (FACT 'kernel' 'SYNTH')    ([],[])
and SYMTAC          = METATAC    (FACT 'kernel' 'SYM')      ([],[])
and MONOEQTAC       = METATAC    (FACT 'kernel' 'MONOEQ')   ([],[])
and MONOINEQTAC     = METATAC    (FACT 'kernel' 'MONOINEQ') ([],[])
and EXTEQTAC        = METATAC    (FACT 'kernel' 'EXTEQ')    ([],[])
and EXTINEQTAC      = METATAC    (FACT 'kernel' 'EXTINEQ')  ([],[])
and MINFIXTAC       = METATAC    (FACT 'kernel' 'MINFIX')   ([],[])
and INEQTAC         = METATAC    (FACT 'kernel' 'INEQ')     ([],[])
and TRANSEQTAC tm   = METATAC    (FACT 'kernel' 'TRANSEQ')
                                ([], [tm, mkvar('y', ":*")])
and 'TRANSINEQTAC tm = METATAC    (FACT 'kernel' 'TRANSINEQ')
                                ([], [tm, mkvar('y', ":*")])

;;

let TRANSTAC tm = (TRANSEQTAC tm) ORELSE (TRANSINEQTAC tm)
and MONOTAC     = MONOEQTAC ORELSE MONOINEQTAC
and EXTTAC      = EXTEQTAC ORELSE EXTINEQTAC
;;

```


Appendix 5

The Axiomatisation and Structural Induction Packages.

The ML text of the axiomatisation and structural induction packages described in Sections 5.1 and 5.2 are given below. The axiomatisation package consists of a master ML text file called `axiom.ml` which successively invokes another nine ML text files, named `ax1.ml` to `ax9.ml`. The structural induction package consists of a single ML text file called `induct.ml`.

```
% axiom.ml %

%
  Assumes the possible use of smash product. Hence, includes
  smash product theory, if not already in theory hierarchy
%

newparent 'smash'
//

sty           , 'sty is bound.'           //
shape         , 'shape is bound.'         //
discshape     , 'discshape is bound.'     //
selshape      , 'selshape is bound.'      //

mlin('ax1',false)
// % Performs some general-purpose bindings %

mlin('ax2',false)
// % Makes appropriate checks to shape etc.%

mlin('ax3',false)
// % Makes the representing type and the isomorphism pair %

mlin('ax4',false)
// % Makes the constructors %

mlin('ax5',false)
// % Makes the discriminators %

mlin('ax6',false)
// % Makes the selectors %

mlin('ax7',false)
// % Makes the copy-functional, and fix point axiom %

mlin('ax8',false)
// % Proves 2 tautologies to tell ind. pack. names %

mlin('ax9',false)
// % Proves some standard thms. %
```

```
% ax1.m1 %

% Some general purpose functions %

mlcinflix 'Then'
''

let f Then g = g o f
''

let curry f x y = f(x, y)
''

let swop f x y = f y x
''

let join tk1 tk2 = implode( (explode tk1) @ (explode tk2) )
''

let ap tm1 tm2 = mkcomb(tm1, tm2)
''

letrec isset l = (null l) => true |
    (let h.t = l in not(mem h t) & (isset t))
''

let mkaxiom (names, lhtms, rhtms) =
    map newaxiom (combine( names
        , (map mkequiv (combine(lhtms, rhtms)))
    )
    )
''

let genconst tkty = newconstant tkty ; mkconst tkty
''

letrec splice (tkl1, tk12) =
    (null tkl1) => tkl2 |
    (null tkl2) => [] |
    (let tkl1.tkl1' = tkl1
        and tkl2.tkl2' = tkl2
        in
        ((tk1 = '_' ) => tkl2 | tkl1) . splice(tkl1', tkl2')
    )
''

%
This function is used to process user-given operator names, assuming
standard names for operators when the user either indicates a
default, or omits the remaining names to be specified. The user
indicates the inclusion of defaults by mentioning the character
_ (UNDERSCORE) instead.
```

```
This is used when naming selectors, and discriminators.

%
''

let isDot tm = (tm = "UP ( )")
''

let isUP tm = ((fst o destconst o fst o destcomb) tm = 'UP') ? false
''

let ispair tm =
    ((tk = 'prod') or (tk = 'X') where (tk, ( )) = desttype(typeof tm))
    ? false
''
```

```
% ax2.ml %
```

```
let unprime = implode o rev o stripp o rev o explode
```

```
whererec stripp tk1 =
  (null tk1) => fail |
  (let tk.tk1' = tk1
   in
    (tk = '') => stripp tk1 | tk1
  )
;;
```

```
letrec clense tm =
  (isvar tm) => mkvar(unprime(fst(destvar tm)), typeof tm) |
  (isconst tm) => tm |
  (iscomb tm) =>
    (let (tmL, tmR) = destcomb tm
     in
      mkcomb((clense tmL), (clense tmR))
    ) |
  (let (tmL, tmR) = destabs tm
   in
      mkabs((clense tmL), (clense tmR))
    )
;;
```

```
let (connames, conterms) =
  let (tk1, tml) = split shape
  in
    (null shape) => failwith 'Empty Domain Equation' |

    (let discselnames =
       filter (λtk. not(tk = '_')) (discshape @ selshape)
     in
       not(isset(tk1 @ discselnames)) => failwith 'Non Unique Names' |

       (tk1, (map (λtm. (tm = "()") => "UP()") | clense tm) tml))
    )
;;
```

```
% Checking definee in Domain Equation %
```

```
let (styt, styvars) =
  (let (tk, tyl) = desttype sty
   in
    if ( (forall isvartype tyl)
      & ((length tyl) = (length (typetyvars sty)))
    )
    then (tk, tyl)
    else fail
  ) ? failwith 'Inappropriate Definee Type'
;;
```

```
let goodvartype tm =
  (forall (λty1. mem ty1 styvars) (termtyvars tm) ) &
  (let ty = (snd o destvar) tm
   in
    (ty = sty) or (isvartype ty) => true |
    (let (tytk, tytyl) = desttype ty
     in
      (
        (tytk = 'prod')
        or (tytk = 'X') => false | (forall IsStyFree tytyl)
      )
    ) ? false
  )
```

```
whererec IsStyFree ty =
  (isvartype ty) => true |
  (let (tytk, tytyl) = desttype ty
   in
    not(tytk = styt) & (forall IsStyFree tytyl)
  )
;;
```

```
letrec desttuple tm =
  (isUP tm) => desttuple (snd(destcomb tm)) |
  (isvar tm) =>
    ( (goodvartype tm) => [tm]
      | failwith 'Bad Var Type'
    ) |
  (ispair tm) =>
    (let (tmL, tmR) = (destpair tm) ? (destsmashpair tm)
     in
      (desttuple tmL) @ (desttuple tmR)
    )
  | failwith 'Argument is not a tuple of variables'
;;
```

```

let convars =
  not(forall isset cvs) => failwith 'Not distinct variables' |
  (exists (exists (\tk. mem tk ('ABS FUN REP' @ connames)))
    (map (map (fst o destvar)) cvs)
  ) => failwith 'Bad Variable Token in Example Term'
  | cvs

where cvs = map getvars conterms

whererec getvars tm = (isDot tm) => [] | desttuple tm
//

```

```

% ax3.ml %

% Manufacturing the representation type %

let rty = mkrtty (map typeof conterms)

whererec mkrtty (ty.tyl) =
  (null tyl) => ty | mktype('sum',[ty; (mkrtty tyl)])
//

% Manufacturing the isomorphism pair and their axioms %

let abs = "ABS:†sty"
and rep = "REP:†rty"
//

let abstk = join 'abs' stytk
and reptk = join 'rep' stytk
//

let abstm = genconst (abstk, "†rty -) †sty")
and reptom = genconst (reptk, "†sty -) †rty")
//

let absaxm = newaxiom (abstk, "†abstm(†reptom †abs) == †abs")
and repaxm = newaxiom (reptk, "†reptom(†abstm †rep) == †rep")
//

```

```
% ax4.ml %

% Manufacturing Constructors %

let contypes = map mkcontype convars

whererec mkcontype tml =
  (null tml) => sty |
  (let tm.tml' = tml
   in
    mktype ('fun', [(typeof tm), (mkcontype tml')]))
  )
  ''

let contms = map genconst (combine (connames, contypes))
''

let conlhs =
  map (λ(tm,tml).revitlist (swop ap) tml tm)
    (combine(contms, convars))
  ''

let conrhs =
  map (ap abstm) (mkconrhs conterms rty)

whererec mkconrhs (tm.tml) ty =
  (null tml) => [tm] |
  (let [(), tyR] = (snd o desttype) ty
   in
    "(INL itm) :ity" . (map (λtm'."(INR itm') :ity")
                           (mkconrhs tml tyR))
  )
  ''

let conaxms = mkaxiom(connames, conlhs, conrhs)
''
```

```
% ax5.ml %

% Manufacturing discriminator axioms %

let discnames = splice(discshape, (map (join 'is') connames))
''

let disctms = map (λtk.genconst (tk,disctype)) discnames

where disctype = ":†sty -> tr"
''

let disclhs = map (λtm.ap tm abs) disctms
''

let outjtml = map (λtm.outj(tm, abs)) conrhs

whererec outj(tml, tm2) =
  (iscomb tml) =>
    (let (ftm, atm) = destcomb tml
     in
      (isconst ftm) =>
        (let ftk = (fst o destconst) ftm
         in
          (ftk = abstk) => outj(atm, ap reptm tm2) |
          (ftk = 'INR') => outj(atm, "OUTR itm2 :†(typeof atm)")
            | tm2
          )
        )
      | tm2
    )
  ''

letrec package l =
  (null l) => [] |
  (let h.t = l
   in
    [h] . (map (λl'.h . l') (package t))
  )
  ''

let discrhs =
  (length outjtml = 1) => ["DEF †(hd outjtml)"]
  | map mkrhstm (package outjtml)

whererec mkrhstm (tm.tml) =
  let tm' = "ISL itm" ? "TT"
  in
    (null tml) => tm' | mkcond(tm', "FP", (mkrhstm tml))
```

```

;;
let discaxms = mkaxiom (discnames, disclhs, discrhs)
;;

```

```

% ax6.ml %

% Manufacturing selector axioms %

let conselterms, conselargs, conselnames =
  let needssselectors =
    filter ($not o isDot o fst)
      (combine(conterms, (combine(outjtml, connames))))
  in
    (fst l, split(snd l)) where l = split needssselectors
;;

let selterms = map mkselterm (combine(conselterms, conselargs))

whererec mkselterm (ctm, stm) =
  let stm' = "(OUTL !stm)" ? stm
  in
    (iscomb ctm) =>
      ((isUP ctm) =>
        (let atm = (snd o destcomb) ctm
         in
           mkselterm(atm, "DOWN !stm'")
        )
      | (let (a,b) = (destpair ctm) ? (destsmashpair ctm)
        and astm = "FST !stm'" ? "P1 !stm'"
        and bstm = "SND !stm'" ? "P2 !stm'"
        in
          (mkselterm (a,astm)) @ (mkselterm (b,bstm))
        )
      )
    | [stm']
  ;;

let (selnames, seltypes, selrhs) =
  letrec mkselname (tk, n, tml) =
    (null tml) => [] |
    (let tm.tml' = tml
     in
       tktytm . mkselname (tk,(n+1),tml')
    )
  where tktytm = ( (join (join 'SEL' tk) (tokofint n))
    , (typeof tm)
    , tm
    )
  in
  let selnametytml =
    split(revitlist
      (λ(tk,tml).λreslst. reslst @ (mkselname (tk,1,tml)))
    )

```

```

        (combine(conselnames, selterms))
        []
    )
    in
    let (selnames', seltypes', selrhs') =
        fst selnametytml, split(snd selnametytml)
    in
    let selnames'' = splice(selshape, selnames')
    in
    (selnames'', seltypes', selrhs')
    ,,

let seltns = map (λ(tk,ty). genconst (tk, "†sty -> †ty"))
    (combine (selnames, seltypes))
    ,,

let sellhs = map (λtm. ap tm abs) seltns
    ,,

let selaxms = mkaxiom (selnames, sellhs, selrhs)
    ,,

```

```

% ax7.ml %

% Manufacturing the 'copy' function. %

let copytk = join 'copy' stytk
    ,,

let copyty = "†(†sty -> †sty) -> (†sty -> †sty)"
    ,,

let copytm = genconst (copytk, copyty)
    ,,

let fun = variant("FUN†sty -> †sty", flat convars)
    ,,

let apfun tm = (ap fun tm) ? tm
    ,,

let copycasetml =
    mkcopycases (conlhs, sellhs)

whererec mkcopycases ( tml1, tml2 ) =
    (null tml1) ⇒ [] |
    (let (stm, tml2') = applyfunsel (hd tml1, tml2)
     in stm . mkcopycases (tl tml1, tml2'))
    )

where applyfunsel (tm, tml) =
    (letrec deflatetm tm' =
        (let (ftm, atm) = destcomb tm' in atm.(deflatetm ftm)) ? [tm']
     in
     letrec inflatetm (tm', al, sell) =
        (null al) ⇒ (tm', sell) |
        (null sell) ⇒ fail |
        inflatetm ( ap tm' (apfun (hd sell)), tl al , tl sell )
     in
     let funtm.arg1 = rev(deflatetm tm)
     in
     inflatetm (funtm, arg1, tml)
    )
    ,,

let copyrhs =
    if (length copycasetml) = 1
    then mkcond((hd disclhs), (hd copycasetml), "UU†sty")
    else mkcopyrhs (combine(disclhs, copycasetml))

whererec mkcopyrhs ((tml1, tml2).tml) =

```

```

(null tml) => tm2 | mkcond(tml,tm2,(mkcopyrhs tml))
''

let copylhs = "(!copytm) (!fun) (!abs)"
''

let copyaxm = newaxiom(copytk, mkequiv(copylhs, copyrhs))
''

let fixaxm = newaxiom(join 'FIX' stytk, "FIX !copytm !abs == !abs")
''

```

/

```
% ax8.ml %
```

```
%
```

We now prove the only facts needed for the induction package. These essentially allow it to discover the appropriate names of the constructors and selectors easily. Of course, these could have been figured out purely from the recursive structure of the Copy Functional, but we chose, as Milner did in his previous package, to record a guaranteed tautology which encodes each constructor and selector along with standard names for their arguments. Note that it is unnecessary to know in an equal fashion, the names of discriminators, since this information will be available to the induction package as and when appropriate.

Note that all proofs are performed once the axiomatisation has been completed, so as to lessen the risk of erroneously using an 'incorrectly' specified theory.

```
%
```

```

let CONJL thml = itlist (curry CONJ) thml AXTRUTH
and CLOSEQ eqn =
  itlist GEN (subtract(formfrees(concl eqn), formlfrees(hyp eqn))) eqn
''

```

```

let confct = newfact ( join 'construct' stytk
                      , CONJL (map (CLOSEQ o REFL) conlhs)
                      )

```

```

and selfct = newfact ( join 'select' stytk
                      , CONJL (map (CLOSEQ o REFL) sellhs)
                      )

```

```

and coverfct =
  (newfact( join 'cover' stytk, cthm )

```

```

    where cthm = (      (SPEC abs)
                    Then (ABS abs)
                    Then SYM
                    Then FIX
                    Then (λth. APTHM th abs)
                    Then (SIMP (ssadd copyaxm BASICSS))
                    Then SYM
                    ) fixaxm

```

```
''
```


% ax9.ml %

% The remaining theorems are an easy bonus; Each one may easily be necessary in even simple proofs, and pretty well all of them are proveable via simplification.

let DEPCASES tm (thmtt, thmuu) =

%

$$\frac{\text{DEF tm} = \text{TT} \text{]- w , DEF tm} = \text{UU} \text{]- w}}{\text{]- w}} \text{ DEPCASES}$$

%
 CASES "DEF itm"
 (thmtt
 , CONTR (concl thmtt) (TRANS(SYM(ASSUME "DEF itm = FF")
 , AXDEF tm
)
)
 , thmuu
)

;;

let MINIMA gftm =

%

$$\frac{\text{]- !x. g(f x) = x}}{\text{]- g UU = UU}} \text{ MINIMA}$$

%
let (g, fx) = destcomb(fst(destequiv(snd(destquant(concl gftm))))))
in
let f = fst(destcomb fx)
and [tya, tyb] = snd(desttype(typeof g))
in
 SYNTH(TRANS(APTERM g (MIN "if (UU:ityb)", SPEC "UU:ityb" gftm)
 , MIN "ig (UU:itya)"
)

;;

let DEFISOPRF (Gftm,Fuu,Guu,F,G,X,tyA,tyB) =

%

$$\frac{\text{]- ! X. G(F X) = X , F UU = UU , G UU = UU , etc}}{\text{]- ! X. DEF(F X) = DEF X}} \text{ DEFISOPRF}$$

%
let FX = "if !X"
in
let dxuu = ASSUME "DEF !X == UU"
and dxtt = ASSUME "DEF !X == TT"
in

let caseFXtt = TRANS(ASSUME "DEF !FX == TT", SYM dxtt)
and GFXuu = TRANS(APTERM G (DEPUU(ASSUME "DEF !FX == UU")), Guu)
in
let Xuu = TRANS(SYM(SPEC X Gftm), GFXuu)
in
let dxuu' = TRANS(APTERM "DEF :ityB -) tr" Xuu
 , DEPCONV "DEF (UU:ityB)"
)

in
let ttuu = TRANS(SYM dxtt, dxuu')
in
let caseFXuu = CONTR "DEF !FX == DEF !X" ttuu
in
let caseXtt = DEPCASES FX (caseFXtt, caseFXuu)
and FXuu = TRANS(APTERM F (DEPUU dxuu), Fuu)
in
let dFXuu' = TRANS(APTERM "DEF :ityA -) tr" FXuu
 , DEPCONV "DEF (UU:ityA)"
)

in
let caseXuu = TRANS(dFXuu', SYM dxuu)
in
 DEPCASES X (caseXtt, caseXuu)

;;

% Record starts here ! %

let absuu = newfact(join abstk 'UU', MINIMA absaxm)
and repuu = newfact(join reptk 'UU', MINIMA repaxm)
 ;;

let defabs = newfact(join 'DEF' abstk
 , DEFISOPRF
 (repaxm,absuu,repuu,abstm,reptm,rep,sty,rty)
)
and defrep = newfact(join 'DEF' reptk
 , DEFISOPRF
 (absaxm,repuu,absuu,reptm,abstm,abs,rty,sty)
)

;;

let thyss = itlist ssadd
 ([absaxm, repaxm, absuu, repuu, defabs, defrep]
 @ conaxms
 @ discaxms
 @ selaxms
)
 BASICSS % assumes that SMASHSS is included here %

;;

```

let DefConthms =
  map newfact (combine( map (join 'DEF') connames
                           , map proveDefCon conlhs
                           )
               )

where proveDefCon tm = snd(simp term thyss "DEF itm")
''

let DiscSelUthms =
  map newfact (combine( map (\tk.join tk 'UU') (discnames @ selnames)
                           , map proveDiscSelUU (disclhs @ sellhs)
                           )
               )

where proveDiscSelUU tm =
  snd(simp term thyss (substinterm ["UU:fsty", abs] tm))
''

let copyfunuu = newfact( join copytk 'UU'
                          , SIMP thyss
                          (SPEC "UU:fsty" (SPEC fun copyaxm))
                          )
''

```

```

% induct.ml %

% Assumes basic.ml %

let ap = curry mkcomb
''

letrec distribute (l, l1) =
  (null l1) => [] |
  (null l) => failwith 'distribute' |
  (let (l1, r) = pairlst (l, hd l1, []) in r . distribute(l1, tl l1))

whererec pairlst (l1, l2, r1) =
  (null l2) => (l1, r1) |
  (null l1) => failwith 'pairlst' |
  pairlst(tl l1, tl l2, r1 @ [hd l1, hd l2])
''

let unprime = implode o rev o strip1 o rev o explode

whererec strip1 tk1 =
  (null tk1) => [] |
  (let tk . tk11 = tk1
   in
    (tk = '') => strip1 tk11 | tk1
  )
''

let varytm (tm, v1) =
  let tmv1 = termfrees tm
  in
    substinterm (combine(newfrees(tmv1, v1), tmv1)) tm

whererec newfrees (v11, v12) =
  (null v11) => [] |
  (let v = variant(hd v11, v12) in v . newfrees(tl v11, v . v12))
''

letrec getcurryarg tm =
  (let (f,a) = destcomb tm in (getcurryarg f) @ [a]) ? []
''

letrec mkcondcases tm =
  (let (c,tm1,tm2) = destcond tm in tm.(mkcondcases tm2)) ? []
''

```

```

let INDSCH thytk tytk =
  % --- Get, once and for all, the data for Struc. Ind. --- %
  let abstk = join 'abs' tytk
  and reptk = join 'rep' tytk
  and copytk = join 'copy' tytk
  in
  let sty = (typeof o fst o destquant o concl) (AXIOM thytk abstk)
  in
  let (stytk, styvars) = desttype sty
  in

  not(stytk = tytk) => failwith 'Inappropriate Induction Scheme' |

  let rty = (typeof o fst o destquant o concl) (AXIOM thytk reptk)
  in
  let copyaxm = AXIOM thytk copytk
  and fixaxm = AXIOM thytk (join 'FIX' stytk)
  and confct = FACT thytk (join 'construct' tytk)
  and selfct = FACT thytk (join 'select' tytk)
  in

let ITAC:tactic (w, ss, fml) =
  (let (xorigin, Worigin) = destquant w
   in
   let (x, W) =
     (let x1 = variant(xorigin, formlfrees (Worigin . fml))
      in
      (x1, substinform [x1,xorigin] Worigin)
     )
   in

   not(admitsinduction(W, x)) => failwith 'Induction Fails' |

  let xty = typeof x
  in
  let (xtytk, xtyvars) = desttype xty
  in

  not(xtytk = stytk) => failwith 'Bad Type' |

  let INSTTY th =
    INSTTYPE (combine(xtyvars, styvars)) th ? (failwith '???')
  and goalfvs = formlfrees (W.fml)
  in
  letrec VARYQ th =
    (let v = (fst o destquant o concl) th
     in
     VARYQ(SPEC (varytm(v, goalfvs)) th)
    ) ? th

```

```

in
let ConfCts = map VARYQ (DESTCONJL(INSTTY confct))
and SelfCts = map (SPEC x) (DESTCONJL(INSTTY selfct))
in
let contms = map (fst o destequiv o concl) ConfCts
and seltms = map (fst o destequiv o concl) SelfCts
in
let convars = map getcurryarg contms
in
let selconvl = distribute(seltms, convars)
in
let (selrecvars, selnonrecvars) =
  split(map (divide (λ(sel,v). typeof v = xty)) selconvl)
in
let (selrectms, recvars) = split(map split selrecvars)
and (selnonrectms, nonrecvars) = split(map split selnonrecvars)
in
let subtmxw tm = substinform [tm,x] W
in
let conclusions =
  map (λ(vl, tm). itlist (curry mkquant) vl (subtmxw tm))
  (combine(nonrecvars, contms))
in
let fmlst =
  (subtmxw "UU:ixty")
  .(map (λ(vl,fm).
    itlist (curry mkquant)
      vl
      (itlist (λv.λfml. mkimp(subtmxw v, fml))
        vl
        fm
      )
    )
  )
  (combine(recvars, conclusions))
in
  (map (λfm. fm, ss, fml) fmlst, INDRUL)

where INDRUL thml % Proof Part of ITAC % =
  (exists (λ(fm,th). ($not o aconvform) (fm, (concl th)))
   (combine(fmlst, thml))
  ) => failwith 'Incorrectly Acheived Goal' |

let Pty = ":ixty -> ixty"
in
let fun = mkvar('FUN', Pty)
and abs = mkvar('ABS', xty)

```

```

in
let uucase . cases = thml
in
let Basis =
  GEN x (SUBST [SYM(MINAP "(UU:↑Fty) ↑x"), x] W uucase)
in
let F = genvar Pty
in
let CopyAxm = INST [F, fun, x, abs] (OPENQ(INSITY copyaxm))
in
let copytm = mkconst(copytk, "↑Fty -> ↑Fty")
and copyrhstm = (snd o destequiv o concl) CopyAxm
in
let FW = "! ↑x . ↑(subtmxW "↑F ↑x")"
in
let FWth = ASSUME FW
in
let Precvars = map (map (ap F)) selrectms
and Fhypotheses = map (map (λv. SPEC v FWth)) selrectms
in
let specrecvars = map (λ(vl,th). revitlist SPEC vl th)
  (combine (Precvars, cases))
in
let MPhyps = map (λ(thl,th). revitlist (swop MP) thl th)
  (combine(Fhypotheses, specrecvars))
in
let IndCases = map (λ(vl, th). revitlist SPEC vl th)
  (combine(selnonrectms, MPhyps))
in
let (lastcase, restofcases) =
  ((h, rev t) where h.t = rev IndCases)
in
let Copybodythm =
  (null restofcases) =>
    CONDCASES x W copyrhstm (lastcase, uucase, uucase) |
    itlist (λ(thmtt,condtm). λthmff.
      CONDCASES x W condtm (thmtt, thmff, uucase)
    )
    (combine(restofcases, (mkcondcases copyrhstm)))
    lastcase
in
let Step = GEN x (SUBST [SYM CopyAxm, x] W Copybodythm)
in
let Induct = INDUCT [copytm, F] FW (Basis, Step)
in
let (xout,Wout) =
  (mem xorigin (formlfrees(hyp Induct))) => (x,W)
  | (xorigin,Worigin)
in
let FixAxm = SPEC xout (INSTTY fixaxm)

```

```

in
  GEN xout (SUBST [FixAxm, xout] Wout (SPEC xout Induct))
)
in
  ITAC
,,

let CASESCH thytk tytk =
  let confct = FACT thytk (join 'construct' tytk)
  and selfct = FACT thytk (join 'select' tytk)
  and coverfct = FACT thytk (join 'cover' tytk)
in
  let (stytk, styvars) =
    (desttype o typeof o fst o destquant o concl) coverfct
in

let CTAC:tactic (w, ss, fml) =
  (let (x, W) =
    (let (x1, W1) = destquant w
      in
      let x2 = variant(x1, formlfrees fml)
      in
      (x2, substinform [x2,x1] W1)
    )
  in
  let xty = typeof x
  in
  let (xtytk, xtyvars) = desttype xty
  in

    not(xtytk = stytk) => failwith 'Bad Type' |

  let INSTTY = INSTTYPE (combine(xtyvars, styvars))
  and goalfvs = formlfrees( W.fml )
  in
  letrec VARYQ th =
    (let v = (fst o destquant o concl) th
      in
      VARYQ(SPEC (varytm(v, goalfvs)) th)
    ) ? th
  in
  let ConFcts = map VARYQ (DESTCONJL(INSTTY confct))
  and SelFcts = (map (SPEC x) (DESTCONJL(INSTTY selfct))) ? []

    % The failure can arise when defining a finite
    % flat domain
    %

  and CoverFct = SPEC x (INSTTY coverfct)

```

```

in
let contms = map (fst o destequiv o concl) ConFcts
and seltms = map (fst o destequiv o concl) SelFcts
in
let convars = map getcurryarg contms
in
let contms1 = "UU:†xty".contms
in
let subtmxw tm = substinform [tm,x] W
in
let fmlst =
  map (λ(vl, tm). itlist (curry mkquant) vl (subtmxw tm))
    (combine([].convars, contms1))
in
let tmfmlst = combine(contms1, fmlst)
in

  (map (λ(tm, fm). (fm, ss, "†x == †tm".fml)) tmfmlst, CASERUL)

where CASERUL thml    % Proof Part of CTAC % =

  (exists (λ(fm,th). ($not o aconvform) (fm, (concl th)))
    (combine(fmlst, thml))
  ) ⇒ failwith 'Incorrectly Acheived Goal' |

let thml1 = map (λ(tm,thm). ( ((swop MP) (REFL tm))
  o (INST [tm,x])
  o (DISCH "†x == †tm")
  ) thm
  )
  (combine(contms1, thml))

in
let uucase . cases = thml1
and selspec11 = map (map fst) (distribute(seltms, convars))
in
let Cases = map (λ(vl, th). revitlist SPEC vl th)
  (combine(selspec11, cases))
and coverlhstm = (fst o destequiv o concl) CoverFct
in
let (lastcase, restofcases) = (h, rev t where h.t = rev Cases)
in
let CoverLhsThm =
  (null restofcases) ⇒
    CONDCASES x W coverlhstm (lastcase, uucase, uucase)
  | itlist (λ(thmtt,tm). λthmff.
    CONDCASES x W tm (thmtt, thmff, uucase)
  )
  (combine(restofcases, (mkcondcases coverlhstm)))
  lastcase

```

```

in
  GEN x (SUBST [CoverFct, x] W CoverLhsThm)
)
in
  CTAC
,,

let ADDSSCASESTAC:tactic (w, ss, asm.fmlst) =
  [(w, (ssadd (ASSUME asm) ss), asm.fmlst)], (λ[th]. th)
,,

```