

Efficiency in a Fully-Expansive Theorem Prover

Richard John Boulton
Churchill College

A dissertation submitted for the degree of
Doctor of Philosophy in the University of Cambridge

December 1993

Preface

This report is a slightly revised version of my PhD thesis, incorporating some suggestions from my examiners and a few changes of my own. I would like to take this opportunity to thank my examiners, Dr Rob Arthan and Dr Martin Richards.

Richard John Boulton
May 1994

Acknowledgements

I am grateful for the help and encouragement of many people over the past three years. First, my thanks to Mike Gordon for his supervision of my research and for encouraging me to develop skills that will, I'm certain, be of great use to me in the future. Tom Melham deserves to be called my second supervisor for all he has taught me and the help he has given. Other members of the Hardware Verification Group in Cambridge have helped me and all have contributed to a very friendly atmosphere in which to work. I thank all the members, past and present, for this.

A number of people have made specific contributions to my work: Mike Gordon, Jim Grundy and anonymous referees provided constructive comments on various papers; John Van Tassel helped me with Lisp and many other computer-related things; the work of Roger Fleming of Hewlett Packard Laboratories, Bristol, stimulated much of my research; Albert Camilleri, also of Hewlett Packard, generously provided access to the linear arithmetic proof procedure he wrote while a student at Cambridge University, and also gave me pointers into the literature; Paul Curzon made his symbolic compiler available to me for experiments; Sreeranga Rajan at UBC produced the ML version of it which was a vital element in those experiments; Malcolm Newey encouraged me to develop a linear arithmetic procedure for HOL, and directed me to the literature; and Robert S. Boyer, Paul Jackson, Matt Kaufmann and Tobias Nipkow also supplied useful references and suggestions.

I am indebted to everyone involved in the development of the HOL system and its predecessors; it is only because of their work that this research has been possible. I would also like to thank the departmental staff for keeping the computers and everything else running.

Financially, this research was supported by the Science and Engineering Research Council of Great Britain. The Computer Laboratory (University of Cambridge), Churchill College, the Commission of the European Communities, ACM SIGDA, the British Council (Canada), and the University of British Columbia provided funds for a number of trips to conferences, universities and research organisations. My gratitude to Andrew Ireland, Matt Kaufmann, Natarajan Shankar and Jeff Joyce for hosting my visits to their respective establishments.

Outside of the department, many friends have enriched my life: Claire Bennetto, Richard Black, Juanito and Astrid Camilleri, Roy Crole, Andy Dean, Chris Fewster, Rupert Ford, Graham French, and everyone in St Andrew's Street Baptist Church and the Robert Hall Society. Finally, I thank my parents for encouraging and supporting my studies over many years. I dedicate this thesis to them and to my grandparents.

*To my parents
and grandparents*

Abstract

The HOL system is a *fully-expansive* theorem prover: Proofs generated in the system are composed of applications of the primitive inference rules of the underlying logic. This has two main advantages. First, the soundness of the system depends only on the implementations of the primitive rules. Second, users can be given the freedom to write their own proof procedures without the risk of making the system unsound. A full functional programming language is provided for this purpose. The disadvantage with the approach is that performance is compromised. This is partly due to the inherent cost of fully expanding a proof but, as demonstrated in this thesis, much of the observed inefficiency is due to the way the derived proof procedures are written.

This thesis seeks to identify sources of non-inherent inefficiency in the HOL system and proposes some general-purpose and some specialised techniques for eliminating it. One area that seems to be particularly amenable to optimisation is equational reasoning. This is significant because equational reasoning constitutes large portions of many proofs. A number of techniques are proposed that transparently optimise equational reasoning. Existing programs in the HOL system require little or no modification to work faster.

The other major contribution of this thesis is a framework in which part of the computation involved in HOL proofs can be postponed. This enables users to make better use of their time. The technique exploits a form of lazy evaluation. The critical feature is the separation of the code that generates the structure of a theorem from the code that justifies it logically. Delaying the justification allows some non-local optimisations to be performed in equational reasoning. None of the techniques sacrifice the security of the fully-expansive approach.

A decision procedure for a subset of the theory of linear arithmetic is used to illustrate many of the techniques. Decision procedures for this theory are commonplace in theorem provers due to the importance of arithmetic reasoning. The techniques described in the thesis have been implemented and execution times are given. The implementation of the arithmetic procedure is a major contribution in itself. For the first time, users of the HOL system are able to prove many arithmetic lemmas automatically in a practical amount of time (typically a second or two).

The applicability of the techniques to other fully-expansive theorem provers and possible extensions of the ideas are considered.

Contents

1	Introduction	1
1.1	Formal Verification	1
1.2	Mechanical Theorem Proving	2
1.3	Fully-Expansive Theorem Provers	3
1.4	Objectives	5
1.5	Related Work	5
1.5.1	Verifying Proof Procedures	5
1.5.2	Unnecessary Inference and Proof Transformation	8
1.5.3	Laziness and Partial Evaluation	9
1.5.4	Equational Reasoning	10
1.6	The HOL System	11
1.6.1	HOL Terms and Types	12
1.6.2	The Metalanguage ML	13
1.6.3	Abstract Data Types	15
1.6.4	ML Data Types for Terms and Theorems	15
1.6.5	Backward Proof	16
1.6.6	Simulating Lazy Evaluation	16
1.6.7	Garbage Collection	16
1.7	Outline of Thesis	17
2	Unnecessary Inferences	19
2.1	Separating Search from Justification	20
2.2	Repeated Traversals	21
2.3	Duplication of Subterms	22
2.4	Discarding Changes	22
2.5	Pre-proving Lemmas	22
2.6	Memoisation	23
2.7	Exploiting the Form of Terms	24
2.8	Use of Extra-Logical Procedures	25
3	Delayed Computation	27
3.1	Using Delay to Improve Productivity	27
3.2	Postponing Primitive Inferences	28
3.3	Lazy Theorems	28
3.4	Lazy Primitive Inference Rules	30

3.5	Lazy Derived Inference Rules	32
3.6	Lazy Tactics	33
3.7	Three Modes of Operation	34
3.8	A Complete Lazy System	35
3.9	Laziness and Verified Rules	35
3.10	Results	36
4	Equational Reasoning	39
4.1	Overview of Equational Reasoning	39
4.1.1	Equivalence Relations	39
4.1.2	Congruences	40
4.1.3	Rewriting	40
4.1.4	Substitution	41
4.2	Conversions	42
4.3	An Example Using Depth Rewriting	43
4.4	Optimisation Using Exceptions	47
4.5	Optimisation Using an ML Data Type	50
4.6	Laziness Enables Further Optimisation	52
4.6.1	Lazy Structures	52
4.6.2	Lazy Conversions	53
4.6.3	Sequencing Lazy Structures	54
4.6.4	Evaluation Using Congruence Rules	57
4.6.5	Sequencing Depth	57
4.6.6	Evaluation Using Substitution	61
4.6.7	Duplicated Lazy Conversions	63
4.7	Destructive Rewrite Rules	64
4.7.1	Extended Lazy Structures	65
4.7.2	Dynamic Dependency	66
4.7.3	Optimising Performance	68
4.8	Abstract Type for Equational Reasoning	69
4.8.1	Signature for Equations	69
4.8.2	Defining New Conversions	72
4.8.3	Complex Primitives	73
4.9	Results	74
5	Decision Procedures	77
5.1	Choosing a Decision Procedure	77
5.2	A Decision Procedure for Linear Arithmetic	78
5.2.1	Linear Arithmetic	78
5.2.2	Selecting the Algorithm	79
5.2.3	The Algorithm	81
5.2.4	Implementation of Normalisation	83
5.2.5	Implementation of Variable Elimination	89
5.2.6	Existentially-Quantified Formulas	92
5.2.7	Results	92

5.3	A Symbolic Compiler	95
5.4	Polymorphic Lazy Theorems	97
5.5	Combining Decision Procedures	99
6	Other Theorem Provers	101
6.1	HOL in Standard ML	101
6.2	Isabelle	102
6.3	LAMBDA	103
6.4	Veritas	103
6.5	Nuprl	104
6.6	Nqthm	105
7	Conclusions	107
7.1	Summary of Research	108
7.1.1	Sources of Inefficiency and Some Solutions	108
7.1.2	Using Delay	109
7.1.3	Optimising Equational Reasoning	110
7.1.4	Decision Procedures	111
7.2	Conclusions for Implementors	112
7.3	Discussion of Laziness	114
7.4	Prospects for Further Work	115

Chapter 1

Introduction

1.1 Formal Verification

Digital computer systems are now widespread and pervasive in society. Their most prominent form is the personal computer to be found in offices and homes, but they are also used extensively to control and enhance the operation of electrical and mechanical appliances. The systems are usually composed of both hardware and software, though application-specific hardware is common in control applications.

The systems being used are becoming increasingly complex. The high degree of complexity has made it difficult to ensure that the systems behave as intended. The discovery of ‘bugs’ in commercial software is very common; indeed the existence of bugs is almost taken for granted. It can be argued that for most commercial software minor bugs are no more than an inconvenience to the user. Whether or not this is true, the same cannot be said for the general-purpose microprocessors around which personal computers and workstations are built, or for hardware/software systems used to monitor or control some aspect of the real world. In the former case, a design error will usually result in the repetition of the complete fabrication process, costing the microprocessor manufacturer huge sums in lost revenue because of a failure to get the product on the market ahead of a competitor. In the microelectronics industry, a few weeks can be critical. In the case of systems interacting with the real world, a design error may endanger life or undermine security, and again have commercial implications.

For these reasons considerable effort and expense is taken to ensure the correctness of designs as early as possible in the process. Typically this has taken the form of simulation, in which test data is given to a model of the design and the results analysed for conformity to the specification. Unfortunately, exhaustive simulation is impractical for all but the very simplest of designs. Systems containing even a moderate amount of state cannot be thoroughly tested because of the huge number of possible input combinations and sequences required. The verification process therefore relies on a very careful choice of input data, intended to exercise all the features of the design. As complexity increases, it becomes more difficult to select test data with a good coverage.

An alternative approach is to exploit the regularity in designs by representing and

analysing them mathematically (formally). These *formal methods* include the use of binary decision diagrams (BDDs [Bry92]), model checkers [McM93], and theorem provers. Their use is known as *formal verification*. Many systems are being *formally specified* but not verified. Formal specification is much easier than verification and often brings to light errors in the design simply by forcing ambiguities to be resolved.

Formal verification is not yet used extensively in the computer industry because the techniques and tools are not sufficiently well-developed to cope with the complexity of industrial-strength designs. In addition, many of the tools require expert human assistance. Theorem provers especially suffer from this drawback. However, they can potentially cope with much greater complexity by means of abstraction. Despite the general lack of interest in industry, for safety-critical and security-critical applications the need for high assurance in the correctness of the design has promoted the use of formal verification. Detailed discussions of formal methods can be found elsewhere [CS90, JAR89].

There has been much progress in the past few years in developing the BDD and model checking approaches for complex systems. However, since the research presented here deals with mechanical theorem proving, nothing further will be said about these approaches.

1.2 Mechanical Theorem Proving

Mechanical Theorem Proving has been an active area of research for more than thirty years. During that time a multitude of theorem proving systems have been developed. Many of these have been implementations of new ideas for automated deduction and are not intended to support formal verification or similar tasks. Other systems have been developed specifically to support formal verification and related activities. There has been considerable exchange of ideas between the two research communities. The proof automation techniques are welcomed by those practising formal methods since the techniques provide them with better tools. Conversely, the need for better tools has driven some of the research into automated deduction. For an introduction to mechanical theorem proving, see Bundy's book [Bun83].

Theorem proving systems currently used for formal verification include the HOL system [GM93], Nqthm [BM88a] (a development of the Boyer-Moore theorem prover [BM79]), Nuprl [C⁺86], Isabelle [Pau90], LAMBDA [AHL91], Eves [CKM⁺91], Larch [GG91], RRL [KZ89], OBJ3 [GWM⁺92], PVS [SOR93], VERITAS [HDH92], SDVS [Mar91], and Gypsy [Goo84]. Of these, the two most widely used systems are HOL and Nqthm. They are two very different systems.

Nqthm is a largely automatic theorem prover for an untyped quantifier-free first-order logic. In contrast, the HOL system is an environment that supports a more expressive logic (higher-order logic) and in which users can develop their own proof tools. It is easier to specify systems and their properties in HOL [Gor86], but proofs of correctness require far more user intervention than in Nqthm.

The logic implemented by Nuprl is based on Martin-Löf's type theory [Mar85], a highly-expressive constructive logic. Isabelle is a generic theorem prover, i.e., users

can define the logic they wish to use. LAMBDA is a commercial system with a proof infrastructure similar to that of Isabelle but dedicated to a particular logic. Early versions implemented a constructive higher-order logic, but more recently the developers have moved to a non-constructive (classical) logic like the one in HOL. The LAMBDA system also features a graphical interface for hardware design. Eves is a verification environment, the main component of which is a theorem prover called Never. Larch and RRL are theorem provers for first-order logic which concentrate on equational reasoning and induction. OBJ3 is based on order-sorted equational logic and is organised in parameterised modules. One feature lacking from HOL is dependent types (types that may involve expressions from the logic). Dependent types are useful for representing generically such things as machine words. VERITAS and PVS provide such types, as does Nuprl. SDVS is a verification system that incorporates simplifiers, decision procedures and symbolic execution. It is based on a temporal logic and has been mainly used for verification of computer descriptions written in languages such as Ada and VHDL. Gypsy is also a verification environment, in this case with its own language. Some of these systems are discussed at greater length in Chapter 6.

This thesis is concerned with a particular class of theorem proving systems of which HOL is an example. The properties of this class are discussed in the following section, and the advantages and disadvantages are presented.

1.3 Fully-Expansive Theorem Provers

There are many ways to classify mechanical theorem provers. As previously discussed, they support different logics and varying degrees of automation. However, this thesis is concerned with the distinction between theorem provers that generate proofs entirely in terms of primitive inferences of the logic, and those that do not. These will be referred to as *fully-expansive* and *partially-expansive* theorem provers, respectively.

Ideally, the primitive inferences of (some formulation of) a logic are the minimal set of rules required to provide the deductive power expected of the logic. In practice, some of the rules taken as primitive may be derivable from others, i.e., the set of rules is not minimal. This may simply be for convenience or because no-one has seen the derivation. However, to call high-level rules ‘primitive’ is probably undesirable because to do so would blur the distinction between primitive and derived rules. Wong [Won93] uses the term *basic inference rules* to refer collectively to the real primitive rules and the derived rules taken as primitive. As a general principle, the primitive rules should be simple and talk about only the very basic constructs of the logic.

An example of a primitive rule of the HOL logic is *modus ponens*:

$$\frac{\Gamma_1 \vdash t_1 \quad \Gamma_2 \vdash t_1 \Rightarrow t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

This states that if a term t_1 holds under assumptions Γ_1 and that under assumptions Γ_2 , t_1 implies t_2 , then t_2 holds under the set-theoretic union of Γ_1 and Γ_2 .

A proof produced by a fully-expansive theorem prover can be viewed as a tree with axioms and pre-proved theorems as leaf nodes, and applications of primitive inference rules as internal nodes. A partially-expansive theorem prover may replace certain subtrees of the proof by applications of metatheorems. Metatheorems are properties that cannot be expressed directly in the logic but are believed to be valid, e.g., “Two conjunctions are equivalent if the sets of conjuncts are equal.” An instance of this metatheorem is:

$$(x \wedge y) \wedge z = y \wedge (x \wedge (z \wedge x))$$

Generating a proof entirely from primitive inferences provides security since, by use of suitable implementation techniques, the critical code of the theorem prover can be limited to the implementation of the primitive inference rules. This also allows the system to be more flexible since users can be given the facilities to write their own derived inference rules without the fear that they will make the theorem prover unsound. A further advantage is that proofs can be checked by an independent proof-checking program. Simple inference steps allow such a program to itself be simple and so be amenable to formal verification. (See Section 1.5.1 for a discussion of work to this end.) The major drawback of the approach is that fully-expansive theorem provers tend to be slow in comparison to systems that exploit metatheorems or implement derived rules of the logic as primitives. There are a number of reasons for this.

Tests on the form of terms and on the side-conditions of rules have to be made for each application of a primitive inference rule. A derived rule composed from several primitive inference rules may, in a fully-expansive theorem prover, repeat the same test several times. In a partially-expansive theorem prover the derived rule would make the test only once. A sophisticated derived rule may involve hundreds of primitive inferences, which corresponds to a lot of redundant computation. A certain amount of compromise is possible: For simple derived rules that are used extensively it may be worth adding them to the critical code, i.e., making them basic rules. This can improve performance significantly, but with a price to pay in terms of having more code to verify or trust.

Another possible reason for the superior performance of many theorem provers is their heavy use of metatheoretic properties. Many popular decision procedures do not produce (or even follow) a proof in the logic. One way around this problem is to prove the correctness of the decision procedure and then make it a new primitive inference rule of the system (see Section 1.5.1). The verification ensures that the extension is sound. The difficulty with this approach is that in order to verify the implementation of the procedure the data structures and algorithms used in the implementation must be represented and reasoned about in logic.

In many applications the security of a theorem prover is not of critical importance. However, one of the major applications of mechanical theorem provers is in the verification of computer systems. In this field the soundness of the theorem prover is of considerable importance, and is a crucial requirement when the systems being verified are safety or security critical, e.g., nuclear powerstation controllers, fly-by-wire aircraft systems, medical equipment, and defence systems.

An alternative to ensuring the correctness of the theorem prover is to construct the prover to generate a proof script which can be checked by a simple formally-verified program (see Section 32.2 of the UK Ministry of Defence’s guidance for the procurement of safety critical software in defence equipment [MoD91]). However, it is not yet clear whether this technique will be practical for large proofs since the proof scripts generated can be huge.

So, despite the poor performance of fully-expansive theorem provers, they provide security without the need for the entire implementation to be verified and they can avoid space problems by generating the proof over time. Enabling users to write their own procedures also allows specialised formal reasoning systems to be developed on top of the theorem proving system, with a sound theoretical foundation and existing theorem proving support.

1.4 Objectives

The aim of the work described in this thesis is to improve the performance of fully-expansive theorem provers by eliminating duplicated and unnecessary primitive inferences, and to improve the productivity of the user. The work has focussed on the HOL system. There has also been an auxiliary aim of providing more automatic proof tools for HOL and this is reflected in the use of decision procedures as a driving force for many of the optimisations.

1.5 Related Work

There are a number of areas of research loosely related to the work presented in this thesis. For example, much of the research in the field of automated reasoning is concerned with producing proofs more efficiently. (The remainder mostly aims to extend the deductive abilities of theorem proving systems.) However, there has been little previous work on efficiency in the context of fully-expansive theorem provers. One area that does merit discussion is that of verifying proof procedures since this can be seen as an alternative means of gaining efficiency without sacrificing security. Verification of proof procedures has already been mentioned briefly, but a more detailed discussion is given below.

The techniques described in Chapter 3 are related to lazy functional programming and partial evaluation. Chapter 4 describes the optimisation of equational reasoning in HOL. Related work on both of these topics is discussed below. Previous work on decision procedures for linear arithmetic (the main topic of Chapter 5) is described in Section 5.2.1.

1.5.1 Verifying Proof Procedures

A theorem prover that only allows the user to apply primitive inference rules is unlikely to be of much use. Only the most trivial proofs can be performed in such a

system. Recognising this, designers of theorem provers have provided various means of extending the power of the system.

One basic approach, exemplified by the LCF system (and also taken by the HOL system) is to provide users with the means to write more complex proof procedures in terms of the primitives. LCF uses the strong typing of the programming language it provides to prevent unsound procedures from being successfully applied.

Brown [Bro80] also argues for the use of complex proof procedures but ones which, for efficiency, do not ultimately apply primitive inferences. Instead he proposes that each procedural form of an inference rule be supplemented by a mathematical form that justifies it. This bears some similarity to the approach taken in Chapter 3 of this thesis where the structure of a theorem is separated from a function which justifies it. Brown's procedural form corresponds roughly to the insecure program used to generate the structure, and his mathematical form corresponds to the justification function. This is somewhat confusing because the rôles of the concrete representation and the representation as a procedure appear to be reversed.

Another approach to extending the power of a theorem prover is to exploit metatheoretic reasoning. There have been a number of attempts at this, with various objectives. One objective is to be able to introduce metatheoretically-provable formulas as theorems at the object level when they are not provable there. This is known as *reflection*. It increases the logical power of the system. A somewhat different objective for metatheoretic reasoning, and one which is more relevant to this thesis, is its use in efficiently introducing derived rules into a theorem prover. In this case, there may be no extension to the set of formulas that are theorems at the object level, but formulas that are already provable can be justified as theorems more quickly.

Reflection is the approach taken by Weyhrauch [Wey80] but he allows metatheorems to be asserted as axioms, so there is no guarantee that the extension is sound. However, the work is motivated more by an interest in formal theories of reasoning than in safely extending theorem provers. The logic considered is first-order.

Davis and Schwartz [DS79] introduce new inference rules by writing them in a decidable subtheory of their logic. A formula of the subtheory may be used as a new inference rule if it can be proved that adding it to the subtheory is a conservative extension. There is no explicit metatheory but adding the rules does change the underlying theory.

Boyer and Moore have used their theorem prover to verify the soundness of new functions represented in their Lisp-like logic [BM81]. The proof is done with respect to an axiomatisation of the syntax and meaning of terms. Representing the rules of inference in the logic is avoided by using a meaning function that maps objects denoting terms to the values of the terms under a given assignment. The representation of the function in the logic is automatically translated into actual Lisp code and used as an efficient derived rule of inference in their system. They claim that the result is comparable in efficiency to a hand-coded function.

Shankar [Sha85] has represented an entire proof checker for a first-order logic as a function in the Boyer-Moore logic. He then proves mechanically a significant metatheorem. By using Boyer and Moore's translator it is possible to use

the implementation of the metatheorem in place of the functions which generate a fully-expanded proof.

Knoblock and Constable [KC86] describe two approaches to formalising the metatheory of constructive type theory in Nuprl. One of these involves partial reflection. Their aim is to improve the reasoning capabilities of the system. They illustrate the construction of simple metatheoretic functions in Nuprl. The proof that such metafunctions produce the desired object-level proof can be done by users in Nuprl itself. It may then be unnecessary to explicitly produce the proof and so the system is made more efficient. In contrast to the earlier work, the proofs themselves are represented in the logic. Allen et al. [ACHA90] describe a continuation of the work in which there is a single type of proof that can mention itself rather than a hierarchy of types. They also describe a number of applications including ones in which efficiency is the main issue.

Howe [How88] describes the internalisation of a portion of Nuprl's metatheory and the use of a partial reflection mechanism as an alternative to using primitive inferences for all proof. Nuprl's logic is sufficiently expressive to be used as its own metalevel. Howe's technique is to lift a goal to the metalevel, apply some transformation at that level, then bring the result back to the original object level. It only deals with part of the logic (roughly the quantifier-free portion). Howe's approach allows user-defined extensions to the inference system in a more flexible way than the use of a reflection principle.

Slind has proposed a technique for his Standard ML version of the HOL system that will allow implementations of derived rules in terms of primitives to be replaced by equivalent but more efficient implementations [Sli92]. The equivalence of the two ML functions is verified by formal proof in the HOL system itself. This approach is becoming feasible with two recent implementations of formal semantics for Standard ML in HOL [VG93, Sym93].

The use of verified rules has the advantage that it is far more efficient than using primitive inferences, and indeed it will not be possible to make the use of primitive inferences as efficient. However, to date there have been few examples of formally-verified rules. The most practical approach may be to have a mixture of verified and unverified rules with the latter using primitive inferences, as suggested by Musser [Mus89].

Due to the difficulty of verifying implementations of theorem proving systems, there has been some interest recently in the use of simple proof checkers. The intention is to have an independent program checking a proof produced by a more sophisticated system. The proof checker should be as simple as possible so that it can be formally verified. Such a program does not need to generate the result of applying an inference rule; it simply needs to be capable of checking that the input formulas and output formulas form a valid instance of the rule.

If a proof checker is used, the correctness of the theorem prover becomes less critical. It may also provide customers with a means of checking for themselves the result of a proof they have contracted out. The big drawback is that representations of fully-expanded proofs tend to be huge and it is not yet clear that they are manageable for realistically-sized proofs. The size of the proof object can be reduced by

using higher-level inference rules but the proof checker then ceases to be a simple program. The work described in this thesis is relevant here because it reduces the number of primitive inferences performed and hence the size of the proof object.

Work is underway to generate proof scripts from the HOL system [Won93] and to formalise the primitive inference rules of the system within the system itself [vW94] (though another theorem prover could have been used). The ultimate aim is to verify an implementation of a proof checker for the logic.

1.5.2 Unnecessary Inference and Proof Transformation

Melham has paid considerable attention to writing efficient proof procedures in the HOL system. Perhaps the best example of this is his type definition package [Mel89] in which he uses a pre-proved general theorem to minimise the amount of inference that takes place at run-time. Others too have gone to trouble to make their code efficient in HOL. However, this thesis is almost certainly the first in-depth study of inefficiency in the HOL system or any other fully-expansive theorem prover.

In addition to ad hoc attempts to reduce the number of primitive inferences performed by proof procedures, there has been some effort to improve the speed of HOL by more general techniques for writing efficient (functional) programs, and by profiling followed by implementation-dependent optimisations. These activities have certainly yielded performance benefits but do not offer any new insight into the inefficiency of fully-expansive theorem provers in particular.

As previously stated, there has been a good deal of work on improving the efficiency of theorem provers, but with little connection to this thesis. Two fairly recent papers that deserve a mention are those by Benanav [Ben89] and Pierce [Pie90]. Benanav discusses unnecessary inference in the context of resolution-based systems. He considers the recognition and deletion of redundant and irrelevant clauses.

Pierce's work is more interesting in a fully-expansive context. It concerns the simplification of proofs by exploiting the correspondence between proofs and programs. Program transformation techniques are used to transform the computational content of a proof. The aim is to produce a shorter proof. One problem with the approach is that the result of the transformations may not correspond to a proof, so either the result must be checked for proofhood or the transformations must only be allowed if they preserve it. The former is less restrictive. The main transformation investigated is the deletion of adjacent inverse operations, an optimisation that is not considered in this thesis. However, Pierce suggests other transformations, as future work, that are considered here: detecting repeated subproofs due to common subexpressions, reordering proof steps, and eliminating superfluous proofs when a subexpression is thrown away by a rewrite rule.

Pierce's aim is not efficiency but more elegant proofs. Thus, he quite legitimately assumes that he has the original proof as an object prior to transformation. There is no efficiency to be gained in a fully-expansive theorem prover after the proof has been done so the transformations would have to be applied on-the-fly instead. However, since much of the computation involved in primitive inferences can be delayed (see Chapter 3) it may be possible to reap benefits from Pierce's approach. Whether

or not this is the case, the techniques might be used to guide the optimisation of derived inference rules, and shortening a proof after generation is advantageous if it is to be passed to a proof checker (see Section 1.5.1).

1.5.3 Laziness and Partial Evaluation

Many of the techniques described in this thesis have strong similarities with lazy functional programming. Functional programming languages perform computation by applying functions to values. Normally functions are themselves values. They are functions in the mathematical sense, i.e., they return the same result whenever they are applied to a particular value. The great advantage is that there are no side-effects, since there is no state. Thus, programmers can take a very localised view when writing code; they do not have to worry about how other parts of the program may affect the result.

There are two basic ways to evaluate an expression in a functional language: either by strict (eager) evaluation, in which all the arguments to a function are evaluated before the application of the function, or by lazy evaluation, in which the function is applied first. These strategies are often referred to as call-by-value and call-by-name, respectively. ML, the programming language of the HOL system, uses strict evaluation. It also has some imperative features, so some people do not consider it to be a true functional language.

The advantage of lazy evaluation is that it often avoids computation that does not need to be performed because only values actually needed for the final result are evaluated. The big drawback is that if an argument is used more than once in the body of the function it may be evaluated several times. This can be avoided by using acyclic graph structures rather than trees to represent expressions, so that once an argument has been evaluated any further references to it point to the evaluated version. There has been a lot of work on improving the efficiency of lazy functional programming and current implementations are good.

Many of the issues that arise in lazy functional programming also apply to this work. However, there is a fundamental difference between the two. The functional programming community is concerned with lazy evaluation at the level of the language semantics. In this thesis, laziness is applied at a higher level in order to postpone, and in certain circumstances avoid, justification of theorems. The critical feature here is the separation of the generation of the structure of a theorem from the justification of it by primitive inferences. Lazy evaluation at the programming language level (in this case, ML) is no replacement for this separation since the structure must be known in order to proceed with further proof and this forces the justification to take place as well. Nevertheless, lazy evaluation would probably avoid computation for theorems which are discarded altogether (i.e., even their structure is not used).

It is not clear that a lazy functional language could be used in place of ML in the HOL system because HOL makes use of a number of features that are not purely functional. However, a similar system, VERITAS, has been implemented in the lazy functional language Haskell, as well as in Standard ML, a strict language similar to

the one used in HOL. One of the aims of implementing VERITAS in Haskell is the desire to exploit parallel evaluation [HDH92]. The absence of side-effects in a pure functional language makes this easier.

Another technique related to lazy evaluation is partial evaluation [JGS93]. A partial evaluator takes a program and part of the input and generates a residual program. Computation that does not depend on the remaining input can be performed during this process. The residual program is, therefore, likely to be much faster than the original. An example of this in functional programming is the application of a function to one or more constant arguments. If the evaluation is left until run-time, the same computation may be performed many times. It would be interesting to see to what extent the HOL system can be optimised by partial evaluation.

1.5.4 Equational Reasoning

Much of the research in this thesis relates to equational reasoning in the HOL system. Paulson [Pau83] describes an implementation of rewriting for a system called LCF. The code for equational reasoning in HOL is based on this implementation. In his paper, Paulson shows how a rewriting engine can be written in a modular way using higher-order functions. Chapter 4 of this thesis demonstrates that the modularity not only provides clarity and flexibility but also allows transparent optimisation.

In 1990, Roger Fleming of Hewlett Packard Laboratories (Bristol, U.K.) developed a new implementation of rewriting for the HOL system. This had a different behaviour to the original and lost much of the modularity of Paulson's code. However, it avoided a considerable number of unnecessary inferences by exploiting the exception handling facilities of ML. This code was not installed in the main part of the HOL system. Due to the difference in behaviour, to have done so would have been very disruptive to users. However, Tom Melham also used exceptions in 1990 to optimise one of the rewriting functions in the main system. Then, in 1991, the current author combined the exception technique with the modular approach (see Section 4.4) allowing the whole of rewriting in HOL to be optimised without changing the behaviour.

There has been a lot of work on efficient rewriting strategies and many, if not all, of the benefits are applicable to fully-expansive theorem provers. However, the use of primitive inferences can change the relative costs of the processes involved. Rewriting involves the matching of the left-hand side of an equation to the term to be rewritten, and the construction of an instantiated version of the right-hand side. The cost of matching in partially-expansive and fully-expansive systems is comparable. However, term construction is much more expensive in a fully-expansive theorem prover because it has to be done by application of primitive inference rules. So, for example, algorithms that perform unnecessary term construction may be far less efficient in a fully-expansive system. It may, therefore, be worth accepting the overheads of a more sophisticated algorithm.

One piece of related work that should be mentioned is the investigation of proof techniques for the Eves verification environment [KP92]. The description of lazy

innermost rewriting in the report inspired the optimisation for destructive rewrite rules described in Section 4.7.

The Eves implementors optimise innermost rewriting, which is naturally an eager process, by arranging for the rewrite rules to be applied lazily. This is very similar to the technique described in Chapter 3 of this thesis but is not intended to be as general. The Eves implementors are looking to avoid only rewrites that will later be thrown away; the aim in Chapter 3 is to avoid all discarded inferences and, perhaps more importantly, to postpone all justification of theorems. The other difference between the approaches is that for Eves the entire evaluation of the discarded rewrites is avoided whereas here it is only the justification; the structures of the results are still computed. In Eves, computing the structure is all that is ever done. The soundness of the code is assumed.

Because the laziness used here operates at a different level to that used in the rewriting functions of Eves, the optimisation for destructive rewrite rules does not occur automatically — hence the special technique described in Section 4.7. The Eves report describes problems with variable bindings due to the change to lazy evaluation and states that the effectiveness of the technique depends on there being some rewrites that can be discarded. Similar observations apply to the work in this thesis. According to the report, lazy rewriting has not been implemented in the theorem prover of Eves because of the extensive changes that would be required.

1.6 The HOL System

The first fully-expansive theorem prover to provide users with a means of developing their own derived rules and proof procedures was Edinburgh LCF. This interactive system featured a higher-order functional programming language called ML [GMM⁺78, GM93] (for *Metalinguage*) in which logical terms, theorems and the primitive rules were available as data types and functions. LCF was developed first at Stanford [Mil72] and then at Edinburgh University [GMW79]. The Stanford version was not programmable. During the 1980's it was further developed at INRIA (France) and at Cambridge University [Pau87]. There are currently a number of systems derived from LCF or based on its methodology.

The HOL system is a direct descendant of LCF, used at many academic and industrial sites around the world for work in formal methods. Other systems currently in use that are in some way derived from LCF include Nuprl, Isabelle and LAMBDA. The main difference between HOL and LCF is in the logic supported. LCF was designed to mechanise a polymorphic version of Scott's Logic for Computable Functions, whereas HOL supports a version of classical higher-order logic.

The higher-order logic supported by the HOL system is a version of Church's simple theory of types [Chu40], adapted to allow type variables in the logic (polymorphism). Every term in the logic has a unique type, though the type may be polymorphic (contain variables). Higher-order functions are allowed. These are functions that take other functions as arguments or return a function as their result. In particular, it is possible to quantify over functions.

The HOL system is currently available under licence (but free of charge) in two versions. The first, HOL88 [GM93], is built on top of Lisp. Much of the low-level code comes directly from LCF. The other version, HOL90 [Sli91], is a reimplementa- tion in Standard ML [MTH90, Pau91]. The details of the user interfaces to the two versions differ somewhat. In particular, HOL90 uses Standard ML as its metalan- guage. However, the logic supported and the theorem proving tools are essentially the same. Most of the work described in this thesis has been involved with the earlier version, HOL88. The implications of the work for HOL90 are discussed in Section 6.1. The description of the HOL system in the following sections refers to HOL88.

There is also a commercial implementation of HOL (in Standard ML) called ‘ProofPower’ by ICL Secure Systems. ICL are using this system to support formal specification and verification through a mechanisation of the language Z [Spi89] in the HOL logic. The proof support in ProofPower differs somewhat from that in HOL88 and HOL90 but the logic is the same.

1.6.1 HOL Terms and Types

The terms of higher-order logic have types. A type in the HOL system is either a type variable (denoted by a name beginning with a ‘*’) or an application of a constructor to zero or more types. The number of arguments taken by a constructor is known as its *arity*. The basic types such as `bool` (Booleans) and `num` (Peano natural numbers) are constructors of arity zero. The basic types, cartesian-product types and function types are given special treatment by the pretty-printer, e.g.:

<code>()bool</code>	is printed as	<code>bool</code>
<code>(()bool, ()num)prod</code>	is printed as	<code>bool × num</code>
<code>(*, *)fun</code>	is printed as	<code>* → *</code>

The last example denotes a function from some type to itself. The variable ‘*’ can be instantiated to any type. In this thesis, HOL types are written in quotation marks with a leading colon, e.g., `":bool"`. HOL terms may also be quoted but without a colon.

A HOL term can be a constant, a variable, a function application, or a λ -abstraction:

$$t ::= c \mid v \mid t t' \mid \lambda v. t$$

Constants and variables are atomic. They consist of a name and a type. Function applications (also called *combinations*) consist of two terms, one a function and the other an argument. An abstraction consists of a variable (the bound variable) and another term (the body). It is assumed that the reader is familiar with these concepts from the λ -calculus. So, a term is a tree structure in which each of the internal nodes is either a combination or an abstraction. Each leaf node is either a constant or a variable. The structure of these trees plays a vital rôle in the optimisation of equational reasoning.

Notice that function application is written by juxtaposing the operator and the operand, e.g., $f x$. This could also be written as $f(x)$ but the parentheses are unnecessary. Functions can be *curried*, that is they may take more than one argument in sequence. Thus, $f x y$ means f applied to x then the result applied to y . This can be written more explicitly as $(f x) y$. So, the result of applying f to x is itself a function. The example application should not be confused with $f (x y)$ which denotes the result of applying f to the result of applying x to y . The same notation is used in the metalanguage, ML.

It is assumed that the reader has some basic familiarity with set theory and logic. The symbols \neg , \wedge , \vee , \Rightarrow , \Leftrightarrow , \forall , and \exists are used for negation ('not'), conjunction ('and'), disjunction ('or'), implication, if-and-only-if, universal quantification ('for all'), and existential quantification ('there exists'), respectively. In HOL, the notation " $\forall x. b$ " is actually an abbreviation for the application of a function ' \forall ' to a λ -abstraction: " $\forall (\lambda x. b)$ ", and similarly for ' \exists '. Some function constants are displayed as infixes, e.g., " $x \wedge y$ " is syntactic sugaring for the term " $(\wedge x) y$ ".

1.6.2 The Metalanguage ML

The techniques described in this thesis are presented using ML data type and function definitions. ML is a strongly-typed functional programming language which uses eager evaluation, i.e., all arguments to a function are evaluated before the function application itself is evaluated. It is an impure functional language: It has imperative features such as mutable data and a sequencing operation. The syntax of the version of ML used by the HOL system is similar, but not identical, to the syntax of the Standard ML language. The features of ML used in this thesis are given in Figure 1.1.

The case statement requires further explanation. The patterns are tested against the expression until one is found that matches. The identifiers in the matching pattern are then bound and are available within the environment when the corresponding result is being evaluated. An underscore can be used as a wildcard in place of a variable when the value is not needed. Figure 1.2 illustrates these features for the type `nat` given in Figure 1.1. Patterns may also be used in declarations.

Exceptions are an important feature of ML. In HOL88 these are simply strings (usually error messages) which can be raised and trapped. If an exception is never trapped it produces an error message to the user. The exceptions in Standard ML are more elaborate; they allow values other than strings to be propagated. Exceptions are normally used when a function cannot return a sensible value for the given arguments. However, as will be seen later, they can also be exploited as an implementation technique. In HOL88 there are three kinds of trapping construct. The first one listed in Figure 1.1 simply traps all exceptions and returns a default value (which must be of an appropriate type). The second allows selective trapping of only those exceptions consisting of a string in the specified list. Other exceptions are allowed to propagate. The third construct binds the exception (string) to the variable `s` so that it can be used in computing the value to return. In particular this allows all but one particular exception to be trapped by testing the value of the

Feature	Example
Function application	<code>f x</code>
λ -abstraction	<code>λx. x + 1</code>
Parentheses	<code>f (g x)</code>
Tuples	<code>(x,y)</code>
Lists	<code>[1;2;3]</code>
Conditionals	<code>if ... then ... else ...</code>
Case statements	<code>case exp of patt1 . res1 ... pattn . resn</code>
Local declarations	<code>let y = (f x) in (y + y)</code>
Function declarations	<code>let f x y = ... x ... y ...</code>
Recursive declarations	<code>letrec fact n = if (n=0) then 1 else n * fact(n-1)</code>
Type abbreviations	<code>lettype int_pair = int \times int</code>
New types	<code>type three = One Two Three</code>
Recursive types	<code>rectype nat = Zero Succ of nat</code>
Raising exceptions	<code>... failwith 'error'</code>
Trapping exceptions	<code>... ? default_value</code>
Selective trapping	<code>... ??['error1';'error2'] default_value</code>
Exception binding	<code>... ?λs (... s ...)</code>
Explicit typing	<code>... : * \rightarrow bool</code>

Figure 1.1: Features of ML

```

let is_zero n = case n of Zero . true | (Succ _) . false;;

letrec val_of n = case n of Zero . 0 | (Succ m) . 1 + val_of m;;

```

Figure 1.2: Examples of the use of the case statement

variable. If the value is the exception to be propagated then it is raised again using the `failwith` construct; otherwise, a result is returned.

The following functions on lists are also used in this thesis: `hd` (take the first element of a list), `tl` (take the remainder of the list), `null` (is the list empty?), `@` (infix function which concatenates two lists). A list can be formed from its head and tail using the ‘cons’ function (an infix ‘.’).

Finally, although ML is capable of inferring the most general type of an expression without being given type information, it is sometimes useful to be able to constrain the type. This can be done by augmenting an expression or subexpression with a colon followed by the type. The example in Figure 1.1 constrains the type of the elided expression to be a function from some variable type (denoted by `*`) to the type of Booleans.

1.6.3 Abstract Data Types

One other feature of ML required is abstract data types. These are required for securely representing theorems (see Section 1.6.4). The example in Figure 1.3 defines a new type for natural numbers using integers as the representation type. The functions `mk_nat` and `dest_nat` are added to the top-level environment. Their bodies have access to the functions `abs_nat` and `rep_nat` which map between the abstract and representation types, but `abs_nat` and `rep_nat` do not become available in the top-level environment. In the example, `dest_nat` has the same functionality as `rep_nat`. In contrast, `mk_nat` only introduces a new value of type `nat` if its argument is a non-negative integer.

```

abstype nat = int
  with mk_nat i = if (i < 0) then failwith 'mk_nat' else abs_nat i
  and dest_nat n = rep_nat n;;

```

Figure 1.3: ML abstract data type for natural numbers

1.6.4 ML Data Types for Terms and Theorems

Logical terms (`term`) and theorems (`thm`) are built-in types. Terms are parsed and printed as quotations, e.g., `"n + 1"`. Theorems in the HOL logic are *sequents*. A sequent is a pair with a set of formulas (the hypotheses) as the first component and a single formula (the conclusion) as the second component. In HOL, a formula is simply a Boolean-valued term. A theorem asserts that the conclusion of the sequent is a consequence of the hypotheses.

Theorems are an abstract type. The representation type consists of a list of terms for the hypotheses paired with a term for the conclusion. The identifiers exported from the abstract type are the axioms and the primitive inference rules. It

is this abstract type together with the strong typing of ML that ensures that only valid conjectures can become theorems. Theorems are printed as sequents: The hypotheses are printed separated by commas, followed by a turnstile, and then the conclusion, e.g., $m < n, n < p \vdash m < p$.

1.6.5 Backward Proof

The HOL system is probably better known for being a tactic-based theorem prover than for being fully-expansive. However, it is not necessary for a fully-expansive theorem prover to be tactic-based, nor is it necessary for a tactic-based theorem prover to be fully-expansive. HOL *tactics* are essentially a means of reversing the inference rules. The inference rules are used to perform *forward proof*; tactics are used for *backward proof*. Both inference rules and tactics are particular kinds of ML function.

In backward proof, instead of constructing the required theorem from simpler theorems, the conjecture is set up as a *goal* and tactics are applied to break it down into simpler goals. Backward proof is usually easier than forward proof. In HOL, each tactic has a rule associated with it so that if theorems can be generated for the subgoals, then application of the rule will yield a theorem for the original goal. When a tactic is applied that yields no subgoals, a theorem for the current goal can be generated immediately. This theorem is then passed back to the tactic which generated the current goal. When all subgoals have been dealt with, a theorem is obtained for the original conjecture. In HOL this is all managed by an interactive facility known as *the subgoal package*.

Users can write their own tactics in ML. This is made easier by the provision of functions for combining tactics, known as *tacticals*.

1.6.6 Simulating Lazy Evaluation

Since ML is an eager language it is necessary to simulate lazy evaluation in order to support the postponement of primitive inferences. This can be done by forming a function with a dummy argument. The argument is a value of the ML type `void` and the body is the expression whose evaluation is to be postponed. The type `void` has only one value, which is denoted by empty parentheses: `()`. ML allows this value to be used in place of the ‘bound variable’ of abstractions. Thus, the function has the form:

$$\lambda(). \textit{expression}$$

1.6.7 Garbage Collection

In functional programming languages the storage required for data does not have to be obtained explicitly by the programmer. Instead, the run-time system for the language automatically allocates memory for new data. This permits a higher level of programming but has the drawback that the programmer cannot relinquish

storage allocated to data that is no longer required. This problem is overcome by a process known as *garbage collection* which is also performed automatically by the system. When free storage is exhausted the system invokes the garbage collector to free-up memory being used for data that can no longer be accessed by the user. Typically this is intermediate data generated by functions in the course of their computation. Such data will be referred to as *garbage*.

1.7 Outline of Thesis

Chapter 1 Introduction, related work, and overview of the HOL system.

Chapter 2 The causes of inefficiency in fully-expansive theorem provers are discussed and techniques for eliminating unnecessary inferences when writing proof procedures are suggested.

Chapter 3 A framework is presented which allows the delay of part of the computation involved in proving theorems in the HOL system. By delaying part of the computation, the user is able to develop an interactive proof more quickly, since the waiting time between each instruction to the system is reduced. This approach seems particularly promising for proof steps that are very computationally-intensive (such as the application of decision procedures).

Chapter 4 This chapter describes equational reasoning in HOL and explains why the algorithms traditionally used can be very inefficient. Ideas for transparently optimising equational reasoning are developed, including optimisations that are only possible when the justification stage of theorem generation is delayed.

Chapter 5 Some of the optimisations and frameworks developed in the preceding chapters are illustrated by presenting a decision procedure for a subset of linear arithmetic and a symbolic compiler. The chapter also shows how performance of decision procedures can be further improved by using abstract (simpler) representations for certain logical terms, and suggests how this optimisation might be retained when decision procedures are combined.

Chapter 6 This chapter describes some other theorem provers and discusses the applicability to them of the ideas developed in the thesis.

Chapter 7 Conclusions and prospects for further work are presented.

Chapter 2

Unnecessary Inferences

Section 1.3 described the computationally-intensive nature of producing a proof in terms of primitive inferences. The extra computation is inherent in the detail of the proof and in the validity checks that are being performed for each primitive inference. However, much of the computation performed during automatic theorem proving does not contribute to the final theorem. Some of it is involved with searching for a proof and a lot is wasted. The application of primitive inference rules that do not contribute to the final theorem will be referred to as *unnecessary inference*.

Unnecessary inference may be due to primitive inferences not being used, or due to repetition of inferences. Often, the inefficiency is more subtle than this, one sequence of inferences being equivalent to another in terms of the result but with a difference in computation time.

Unused inferences can occur during search down a blind alley. The inferences do not contribute to the proof of the original conjecture. Unused inference is easily spotted when it is due to high-level aspects of the algorithms, but it can also occur at a low level due to the detailed implementation of the algorithms.

Repetition of inferences may be due to a lemma being proved every time a proof procedure is executed, when it could be proved just once and instantiated on each call. Melham exploits pre-proved lemmas to obtain good performance in his type definition package [Mel89] for HOL, and others have followed his lead. Repeated inference can also occur due to the duplication of subterms, in normalisation procedures for example. Such repetition is inherent and can be difficult to avoid. Repetition may also be due to badly-organised control, in which case a change to a more delicate algorithm is all that is required.

To avoid inherent repetition it is often necessary to keep track of the inferences that have already been performed. This tends to be very costly in terms of space, so there is a time-space trade-off to be made. Also, managing the information that keeps track of the previous inferences may be costly in time. There is a well-known technique for eliminating repeated computation which provides a general-purpose solution to the problem of repeated inference. The technique is known as *memoisation* (see Section 2.6). However, the overheads involved with implementing the technique may outweigh the benefits. When applicable, more specialised approaches are often better.

For specialised proof procedures there is usually more opportunity to minimise unnecessary inference. Domain specific information can be exploited to improve performance. It may be possible to guide a proof procedure using knowledge about the form of the logical terms, or it may even be possible to compute an answer outside of the theorem prover (or at least the theorem proving procedures) which allows a much shorter proof to be obtained. Such techniques are exploited by all kinds of theorem prover, but in a partially-expansive system they would be used directly. If a procedure is available to compute an answer outside of the logic it may be used to decide the truth of a formula. However, in a fully-expansive theorem prover it is not sufficient for the procedure to determine the validity of the formula; it must also assist in the application of primitive inference rules which rigorously establish the fact. Thus, not all procedures used by partially-expansive theorem provers are helpful in a fully-expansive system.

In the remainder of this chapter, some of the issues identified above are discussed in more detail.

2.1 Separating Search from Justification

Primitive inferences performed down unsuccessful branches of a search are a major source of wasted computation. Primitive inference rules are applied down such branches because the simplest way to code the procedure is to integrate the search for a proof with the formal justification of the proof. This applies for both high-level and low-level procedures. Rewriting (Section 4.1.3) is a good example of a low-level procedure in which integration of search and justification is inefficient.

A *depth rewrite* of a term is the application of one or more rewrite rules at every subterm to which they are applicable. There are a number of traversal strategies. Some rewrite subterms before rewriting the top-level term; others try to rewrite at the top level first [Pau83]. Some just try one pass over the term; others repeatedly pass over the term until no changes take place. The obvious algorithms for performing a depth rewrite of a term traverse its tree structure applying the rewrite rules wherever possible. Any branches of the tree in which the rewrite rules do not apply are unchanged. In fact, the obvious algorithms will rebuild such branches to produce a subtree identical to the original. This involves a large number of primitive inferences when one would suffice. The inefficiency of depth rewriting is considered further in Chapter 4.

It is (relatively) cheap, though not necessarily simple, to avoid performing unused inferences. Generally, all that is required is for the search to be separated from the actual inference, and only when the route to a proof has been found is the necessary inference performed. Chapter 3 presents a framework in which this can readily be achieved.

Separation of the search for a proof from the justification is critical for the efficiency of fully-expansive theorem provers. In partially-expansive systems such separation is far less important because fewer (if any) primitive inferences take place.

Efficiency is not the only reason to separate the search for a proof from the justification. Having the search separate clarifies the algorithms or heuristics that are being used and hence improves human understanding and the ability to automate the proof. The Mathematical Reasoning Group at the Department of Artificial Intelligence, Edinburgh have been working along these lines for several years. They have introduced a notion of *proof plans* which operate over specifications of the justification procedures rather than the procedures themselves [BvHHS91].

2.2 Repeated Traversals

The Edinburgh proof planner, CLAM, searches for a proof which can then be given to a theorem prover to check. Thus, using CLAM with a fully-expansive theorem prover should be efficient, assuming the planner finds a good (short) proof. However, the proof planner does not concern itself with low-level efficiency issues, and it would not be desirable for it to do so, as this would prevent it from being independent of the theorem prover. One consequence of this is that CLAM generates proof plans in which subterms are indicated by their position relative to the root (top) of terms. If the theorem prover follows these instructions directly it may make repeated traversals of the term when one would suffice. In a fully-expansive theorem prover, primitive inference rules are applied during these term traversals, so this is particularly costly. One way around this is to process the proof plans before passing them to the theorem prover. It is also possible to achieve the optimisation transparently using the approach described in Section 4.6.

Repeated traversals of a term may also arise in normalisation procedures. The natural way to think about many normalisation operations is as a sequence of steps such as elimination of operators, stratification (e.g., forming a disjunction of conjunctions from an arbitrary term involving disjunction and conjunction), reordering (using associativity and commutativity), etc. (See Bundy's paper on proof plans for normalisation [Bun91] for a description of these normalisation steps.) The obvious way to code the normalisation procedure is therefore as a composition of procedures for the constituent steps. Unfortunately, this may lead to unnecessary term traversal and associated unnecessary inferences. It is better to do all the normalisation that is going to occur at a subterm in one pass over the term. This is not always possible because part of the normalisation may have to wait until the consequences of earlier parts have been taken into account at higher levels in the term. In other words, not all normalisations can be performed in a single traversal of the term. However, it is worth considering what the minimum number of traversals required is, and implementing the procedure accordingly.

Note that although linear composition of normalisation procedures may not be consistent with efficiency, this does not mean that the constituent steps have to be merged into one procedure. Instead the procedures for the constituent steps can be parameterised over other procedures (provided the implementation language has higher-order functions). A procedure for the entire normalisation is then obtained by making some procedures arguments of others. This is illustrated in Section 5.2.4.

2.3 Duplication of Subterms

A common step in normalisation of a term is the elimination of operators. This often causes subterms in the original term to appear more than once in the new term. For example, logical equivalence (Boolean equality) gives rise to duplicated subterms when it is eliminated to obtain two implications:

$$(x \Leftrightarrow y) = (x \Rightarrow y) \wedge (y \Rightarrow x)$$

Apart from the performance issues associated with the increase in term size, this may also lead to the normalisation procedure being applied to the duplicated subterms (x and y) more than once. The repeated inference can often be avoided by changing the order of the normalisation operations so that the subterms are normalised before the operator is eliminated.

2.4 Discarding Changes

There are times when, due to the specific form of the term being processed, general algorithms for rewriting, normalisation, etc., perform inferences that are later discarded. One example of this is rewriting with an equation that has a variable on its left-hand side that does not appear on the right, e.g.:

$$0 * n = 0$$

When this rule is applied, any changes that have taken place in the subterm matched by the variable n are discarded. Other examples arise in normalisation procedures, where work can be done to normalise a subterm only to have it renormalised due to changes elsewhere in the term. It may seem that this can easily be solved by modifying the normalisation algorithm, but in practice such changes can lead to other inefficiencies and it can become very difficult to tell which approach is more efficient. In fact, this may depend on the form of the term being processed.

At present, careful thought on the part of the implementor is required to avoid discarding changes. It would be helpful if proof procedures could be written in a framework in which the system functions detect the waste and eliminate it. In Chapter 4, methods are described for achieving transparent optimisation for unchanged subterms, repeated traversals, and use of destructive rewrite rules such as the one above. The programmer does not have to use algorithms that explicitly implement the optimisations. The prospects for relieving the programmer of considerations about some of the other causes of unnecessary inference are discussed briefly in Section 7.4.

2.5 Pre-proving Lemmas

One of the most effective means of improving efficiency in a fully-expansive theorem prover is to replace a subtree of the proof tree with a pre-proved theorem. Instead of

calling a proof procedure to generate the subproof, the pre-proved theorem can just be inserted as a leaf node, possibly after some instantiation. As a simple example of this, consider the derivation of the theorem:

$$\vdash (x \vee y) \wedge (x \vee \neg y) \Rightarrow x$$

from:

$$\vdash (x \vee y) \Rightarrow ((x \vee \neg y) \Rightarrow x)$$

This can be achieved by the following proof:

- | | |
|---|--------------------|
| 1. $(x \vee y) \vdash (x \vee \neg y) \Rightarrow x$ | from the theorem |
| 2. $(x \vee y), (x \vee \neg y) \vdash x$ | from 1 |
| 3. $(x \vee y) \wedge (x \vee \neg y) \vdash (x \vee y) \wedge (x \vee \neg y)$ | trivial entailment |
| 4. $(x \vee y) \wedge (x \vee \neg y) \vdash (x \vee y)$ | from 3 |
| 5. $(x \vee y) \wedge (x \vee \neg y) \vdash (x \vee \neg y)$ | from 3 |
| 6. $(x \vee y) \wedge (x \vee \neg y), (x \vee \neg y) \vdash x$ | from 4 and 2 |
| 7. $(x \vee y) \wedge (x \vee \neg y) \vdash x$ | from 5 and 6 |
| 8. $\vdash (x \vee y) \wedge (x \vee \neg y) \Rightarrow x$ | from 7 |

However, it is shorter to instantiate the following pre-proved theorem:

$$\vdash \forall a b c. (a \Rightarrow b \Rightarrow c) = (a \wedge b \Rightarrow c)$$

The universally-quantified variables a , b and c are specialised to " $x \vee y$ ", " $x \vee \neg y$ " and " x " respectively. Use of a pre-proved theorem is not guaranteed to be faster than a direct derivation, but in many cases it is. The general idea is that the specialisation of a theorem takes a fixed number of inferences per variable no matter what the form of the theorem. Therefore, theorems that capture as much change as possible should be used thereby moving some of the inferences out of the proof procedure and into the one-off proof of the theorem. (The theorems used in the procedure are pre-proved.)

Higher-order logics such as the one supported by HOL are particularly amenable to the use of pre-proved lemmas since more interesting properties can be captured as theorems thanks to the presence of function-valued variables.

2.6 Memoisation

Avoiding repeated inferences may in general involve too many overheads in both space and time for it to be worthwhile. However, there is a simple and general technique that can be employed to eliminate repeated inferences, which under favourable circumstances will optimise performance. The technique is known as *memoisation* and was first suggested by Michie in 1968 [Mic68]. It is similar to the notion of a cache in hardware design.

The idea is to modify a function so that it has a memory. When a result has been computed a pair is added to the memory consisting of the argument and the result. When the function is applied it first checks the memory to see if the argument is present. If so, the result has already been computed and is in the memory ready to be returned immediately. If the argument is not found in the memory then the result is computed normally and added to it.

The function modifies itself, so it can be used just as any other function would be. The behaviour of the function is not changed by memoising it; only the computation time is altered. The technique is not of course applicable to ‘functions’ that are already state-dependent since they may not return the same result when called more than once on the same argument value. The costs of the technique are in the storage required to maintain the results and in the time to search the memory for the argument value. Both of these can be reduced by having a fixed-size memory together with a policy for discarding results when the memory overflows. The search can also be implemented efficiently by the usual techniques such as hashing.

As an example of a situation in which memoisation can be worthwhile, consider proving results about arithmetic operations on numeric constants. In particular, consider multiplication. In HOL, the multiplication operator for natural numbers is defined recursively by the following theorem:

$$\vdash (\forall n. 0 * n = 0) \wedge (\forall m n. (m + 1) * n = (m * n) + n)$$

Thus, to prove that $2 * 3 = 6$ the following steps are required:

$$\begin{aligned} 2 * 3 &= (1 + 1) * 3 = (1 * 3) + 3 \\ &= ((0 + 1) * 3) + 3 = ((0 * 3) + 3) + 3 \\ &= (0 + 3) + 3 = 3 + 3 = 6 \end{aligned}$$

This requires the use of both equations in the above definition of multiplication and a procedure for proving results of addition. There are m recursive calls to the multiplication procedure, where m is the value of the first argument of the product, and m additions of the second argument to a partial result. So, proofs of multiplication properties can be computationally expensive, especially if the arguments have large values.

A given arithmetic proof may involve the same multiplication in several places. Considering the high cost of proving the result of each multiplication, it may be worth avoiding the repetition by memoising the procedure.

2.7 Exploiting the Form of Terms

In certain circumstances it may be known that a term has a particular structure either because it was machine-generated or because of the current context. Examples of this are terms representing the semantics of a programming or hardware description language, and terms representing the implementation of some electronic circuit.

As an example, consider simplifying terms representing the semantics of some language. It is often possible to exploit the syntax of the language to restrict the search space. (The syntax restricts the possible form of the term.) Simplification procedures can be written that recurse in the same way as the productions of the grammar for the language. This is more efficient than using brute-force depth rewriting.

Suppose that as part of the semantics of a hardware description language there is a constant, `CONC`, defined which represents a construct for concatenating two buses (represented by time-varying lists). The following theorem defines `CONC` in terms of the function `APPEND` for concatenating two lists. Thus `CONC` is a version of `APPEND` lifted to the level of signals.

$$\vdash \forall bus_1 bus_2 t. (bus_1 \text{ CONC } bus_2) t = \text{APPEND } (bus_1 t) (bus_2 t)$$

Before doing any proofs about a term involving `CONC` or any other *semantic constant* [BGG⁺92], it may be desirable to expand the constants using their definitions. This can be achieved by a brute-force depth rewrite. However, for large systems which may have as many as a hundred semantic constants, this is highly inefficient. It would involve a depth rewrite on the term where at each subterm all the semantic constant definitions are tried until the correct one is found.

A more efficient approach is to determine the name of the function being applied at the top level of the subterm. The correct definition can then be chosen by indexing on the name. Unfortunately, for real languages, the situation may not be this simple. The applications of semantic constants may have such things as λ -abstractions constructed around them. A better approach is to make use of the syntax of the language, knowing that the term (having been generated by machine translation) conforms to the syntax. The idea is to have mutually-recursive simplification functions, one for each syntactic category. Each of these functions only has to handle the semantic constants associated with its syntactic category.

There is a problem with this approach. If the syntax is mutually recursive and the procedures for each syntactic category call the next one before considering a non-recursive case, the procedures may loop. Looping can happen in the simplification procedures even when it cannot happen in the translator. This is because in the syntax there may be parentheses that have to be consumed before recursing, but in the logical term there may be nothing equivalent to the parentheses.

Looping can be prevented by passing a flag around the mutually-recursive procedures. The flag is set to false on entering the procedures and at some point in the loop the flag is tested. If it is true, the non-recursive possibilities are investigated. Otherwise the flag is set to true and recursion continues. This prevents the procedures from going round the loop two or more times on the same term.

2.8 Use of Extra-Logical Procedures

Metalogical and extra-logical procedures cannot be used directly in a fully-expansive theorem prover since they do not apply primitive inference rules. However, there

are situations in which such procedures can provide information that enables a more efficient object-level proof. For example, proving the validity of an existentially-quantified formula may be simple once a witness for the formula is found. Any means can be used to find the witness; there is no need for primitive inferences to be performed.

In a similar way, an expression simplifier may be slower (and more difficult to write) than a procedure that proves the equivalence of an expression to a simplified version of the same expression. In such a case, an external simplifier can be used to compute the new expression. This is then passed to the equivalence procedure and the overall effect is the same as if a fully-expansive simplifier had been used. Harrison and Théry [HT93] have taken this approach for reasoning about the real numbers in HOL by interfacing HOL to Maple, a computer algebra system.

A program or system that provides an answer for a fully-expansive theorem prover will be referred to as an *oracle*. An oracle does not provide a proof, but merely some information that enables a proof to be performed more quickly or easily. In contrast, it is also possible to interface a fully-expansive theorem prover to another theorem prover or a proof planner. In this case the external program or system provides some sort of proof for the fully-expansive prover to follow, either in terms of applications of primitive inference rules or at least in sufficient detail for the fully-expansive prover to fill in the gaps.

Chapter 3

Delayed Computation

3.1 Using Delay to Improve Productivity

The HOL system is an interactive theorem prover. The user executes commands in the metalanguage, ML, to prove a theorem, either by applying inference rules to axioms and existing theorems, or by setting up the conjecture as a goal and applying tactics (Section 1.6.5) to generate new goals until there are no goals remaining to be proved. Each function application takes time. When that time exceeds a second or two, the user will be spending some time waiting. This is both frustrating and unproductive. Often the user will already know what the next step of the proof is and be eager to get on with it. In other cases, the user may have to wait to see whether the application has succeeded and achieved the required result.

It is the waiting time during each function application that creates the perception of the system being slow. Certainly, the actual speed of the functions is important, but when a proof is running automatically, the user will be doing something else and so will be less concerned about the actual speed of the proof functions. Users could be more productive if the waiting times were concentrated into larger chunks. One way to achieve this is to increase the automation of the theorem prover. However, that is not always possible and the user often wants to have more delicate control over the proof process. Another approach is to delay part of the computation so that control is returned to the user more quickly. The delayed computation can be built up into a large chunk to be executed while the user does something else. This is possible in a fully-expansive theorem prover because part of the computation is involved with checking the result, not simply with generating it.

In 1989, Musser [Mus89] suggested postponing part of the computation performed by HOL tactics. This chapter presents details of an approach that meets Musser's objectives. The approach also allows optimisations to be made that would not otherwise be possible, and it provides a framework in which search can readily be separated from justification. The parts of the computation that it is reasonable to postpone are described, problems are discussed, and results are presented that give an indication of the extent to which the approach improves the productivity of the user.

3.2 Postponing Primitive Inferences

This section outlines the approach to postponing justification. The basic idea is to determine the result of the application of a derived rule or other proof procedure in an efficient (and possibly metatheoretic) way so that the interactive proof development can continue, while retaining the ability to perform all the validity checks at the primitive-inference level at some later time. Security is not breached because a conjecture is not considered to have been proved until all the validity checks have been made. This opens up the possibility of proceeding with a proof believing a previous step to be valid when it is not. However, this can only arise when there is an error in the implementation. In partially-expansive theorem provers such an error may lead to an invalid conjecture being ‘proved’. With the approach presented here, the error will be detected before the conjecture is considered proved, though the detection may not happen at the invalid proof step.

The technique makes use of *lazy theorems*. These consist of the underlying term structure of a theorem together with a *justification function* capable of generating the theorem for that structure. The generation of the term structure for a theorem together with a justification function will be referred to as the *initial stage* of computation. The evaluation of the justification function to obtain a real theorem for the structure will be referred to as the *justification stage*.

3.3 Lazy Theorems

An ordinary theorem in HOL is an object of ML type `goal` wrapped up as an abstract type `thm`. The type `goal` captures the structure of a theorem. It consists of a list of terms representing the hypotheses of the theorem and a single term for the conclusion. The simplest representation for a lazy theorem in the HOL system is given by the ML type abbreviation in Figure 3.1.

```

lettype goal = (term)list × term

lettype lazy_thm = goal × (void → thm)

```

Figure 3.1: ML type for simple lazy theorems

Elements of this type are pairs. The first component of such a pair captures the structure of the lazy theorem. The second component is a justification function which takes a dummy argument and returns a theorem. The dummy argument prevents the theorem from being computed until the justification function is applied. In this way the justification stage of a proof can be delayed. The function is essentially the ‘lazy theorem’, but in order to do anything useful with a lazy theorem, programs need to know its structure.

Unfortunately this simple implementation of lazy theorems has a serious drawback. Suppose the following lazy theorem has been generated:

$$((["x \wedge y"], "y \wedge x"), f)$$

where f is the justification function. The inference rule `CONJ_PAIR` takes a theorem whose conclusion is a conjunction and generates two theorems, one for each conjunct. A lazy version of this rule would, when applied to the lazy theorem above, produce the following:

$$((["x \wedge y"], "y"), f_1) \quad ((["x \wedge y"], "x"), f_2)$$

Both of the functions f_1 and f_2 will contain evaluations of f . So when f_1 is evaluated all the computation involved in evaluating f will be performed. This is also the case when f_2 is evaluated. So, f will be evaluated twice. As other rules are applied to the two new lazy theorems the number of times f can be evaluated may increase further.

A similar problem arises in the implementation of lazy functional programming languages. The solution adopted there (called *sharing* [Pey87]) can be exploited to solve the problem with lazy theorems. This is done by making a lazy theorem an assignable entity, so that once it has been fully proved by evaluating the justification function, the lazy structure is replaced by a real theorem. This can be achieved by having a concrete representation type for lazy theorems that consists of two possible structures, as given in Figure 3.2. The representation is either a lazy theorem as described above or it is a fully-proved theorem.

```
type lazy_thm_rep = Lazy_thm of goal × (void → thm)
                  | Proved_thm of thm
```

Figure 3.2: Representation type for lazy theorems

Reference (assignable) cells incorporating this representation type can be implemented in ML using the imperative features of the language. These reference cells can be used to implement lazy theorems that will only ever be proved once. In the example with the rule `CONJ_PAIR`, when one of the resulting lazy theorems is proved, the function f will be evaluated and the result used to replace the original lazy theorem. This result will be waiting in the reference cell ready for use when the other lazy theorem is proved.

The reference cells are encapsulated in an abstract data type. There are two reasons for this:

- Lazy theorems may be mutually recursive with the type of real theorems, so to retain security lazy theorems must be defined within the abstraction for theorems.

- By encapsulating the reference cells, a functional view is retained. The programmer does not see the imperative features.

Retaining a functional view is not essential. However, if lazy theorems are placed within the abstraction for theorems, it is vital for security that the user cannot update the reference cells. This is illustrated in Section 3.4 when the definition of primitive inference rules is discussed. That section discusses the advantages and disadvantages of having lazy theorems within the abstraction for theorems.

```

mk_lazy_thm      : (goal × (void → thm)) → lazy_thm
mk_proved_thm   : thm → lazy_thm
dest_lazy_thm    : lazy_thm → goal
prove_lazy_thm  : lazy_thm → thm

```

Figure 3.3: Functions for manipulating lazy theorems

The primitive functions for manipulating lazy theorems are given in Figure 3.3. The function `mk_lazy_thm` generates a lazy theorem from a goal and a justification function; `mk_proved_thm` generates a lazy theorem from a theorem that has already been proved; `dest_lazy_thm` obtains the structure of a lazy theorem (the goal); and `prove_lazy_thm` obtains a proved theorem from a lazy theorem. It is in a call to the last function that the justification is performed. The function checks that the theorem generated has the same structure as the goal. (The user could have made a lazy theorem from one structure together with a function that generates a theorem with a different structure.) If not, an ML exception is raised. It is also possible for the application of the justification function to fail rather than generate a theorem. Of course, if the lazy theorem has already been proved then there is no justification or checking to do.

3.4 Lazy Primitive Inference Rules

As mentioned in Section 1.6.4, the primitive inference rules of HOL are part of the abstract data type for theorems. Within the abstract data type, the function `abs_thm` is used to generate a theorem from a goal¹.

In order to avoid having to make major changes throughout the source code (and user code) for the HOL system, it is necessary to change all the inference rules to operate over lazy theorems instead of theorems. For consistency, this must include the primitive inference rules. Lazy versions of the primitive rules could be

¹In versions of the HOL system built on top of Lisp, the data type of theorems is not defined using the ML `abstype` construct, but directly in Lisp. However, the idealised view will be taken here. Furthermore, the `abs_thm` function in those versions (actually called `mk_thm`) verifies that the hypotheses and conclusion are Boolean-valued. Implementors of lazy rules should bear in mind that these tests may have to be introduced explicitly into the initial stage of computation.

implemented around the existing primitives. However, all the validity checks for the rule would then be performed twice, once to ensure that the lazy theorem will give rise to a real theorem, and once within the justification function that generates the real theorem. Since all proofs in a fully-expansive theorem prover are ultimately in terms of the primitive rules, this could make the full computation of theorems take twice as long.

Take as an example the primitive inference rule **ABS**, which has the following property:

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x. t_1) = (\lambda x. t_2)} \quad x \text{ is not free in } \Gamma$$

Assuming that the original implementation (which returns a real theorem) is called **PRIM_ABS**, the implementation of **ABS** in terms of it can be seen in Figure 3.4. The ML identifier **lth** is bound to the lazy theorem on the top of the rule above, and **x** is bound to the object language variable to be used in the λ -abstraction. The function checks that **x** does not occur in the free variables of the hypotheses (**gamma**). Provided this check succeeds, the underlying structure for the new lazy theorem is built and the lazy theorem is generated. The validity check and the term construction will be repeated when the application of **PRIM_ABS** is evaluated.

```

let ABS x lth =
  let (gamma,eqn) = dest_lazy_thm lth
  in if mem x (freesl gamma)
     then failwith 'ABS'
     else let (t1,t2) = dest_eq eqn
          in let gl = (gamma,mk_eq(mk_abs(x,t1),mk_abs(x,t2)))
             in mk_lazy_thm
                (gl,(λ(). PRIM_ABS x (prove_lazy_thm lth)));;

```

Figure 3.4: A lazy primitive rule derived from a real primitive

So, in order for a lazy system to be practical, the lazy primitive inference rules must be implemented directly using **abs_thm**. In other words, they must be part of the abstract type for theorems. So, it is necessary to modify the abstract data type for theorems, the heart of the security of the system. On the face of it, this is not a serious concern because the code for the lazy primitive inference rules can be based directly on the original primitives. The only real change is that the call to **abs_thm** is placed inside a justification function instead of being applied immediately. However, this has a significant effect on the semantics. The validity checks are performed on term structures obtained from the arguments to the rule. These arguments may now be lazy theorems, so the terms can only be trusted if the lazy theorem gives rise to a real theorem. It is therefore essential to check that all the argument lazy theorems can be justified before generating a real theorem for the result of the rule.

```

let ABS x lth =
  let (gamma,eqn) = dest_lazy_thm lth
  in if mem x (freesl gamma)
     then failwith 'ABS'
     else let (t1,t2) = dest_eq eqn
           in let gl = (gamma,mk_eq(mk_abs(x,t1),mk_abs(x,t2)))
              in mk_lazy_thm
                 (gl,(λ(). prove_lazy_thm lth; abs_thm gl));;

```

Figure 3.5: Implementation of a lazy primitive rule

This is done by applying `prove_lazy_thm` to the arguments *before* the `abs_thm` is applied (see Figure 3.5). The semicolon is a sequencing operation. If the attempt to justify the lazy theorem `lth` fails, the application of `abs_thm` will not be evaluated.

When the lazy primitive rules are defined within the abstract data type for theorems it is important that the abstraction hides the reference cells used to implement lazy theorems. Otherwise, the user could arrange for the application of `prove_lazy_thm` in Figure 3.5 to succeed even when `lth` has a justification function that does not prove the structure. To do this, the user would simply have to replace the original contents of `lth` with a justifiable lazy theorem before the result of the call to `ABS` is justified.

It is necessary to check the argument lazy theorems only when `abs_thm` is being used. Derived lazy inference rules are defined in terms of the primitive rules, so all necessary validity checks are made, provided the primitive rules make them.

3.5 Lazy Derived Inference Rules

At the risk of generating lazy theorems that will not be justifiable, and at the cost of increasing the overall computation time, derived rules can be written to minimise the initial computation time. If a derived rule is defined directly in terms of primitive rules then all the checks of the primitive rules will be performed during the initial stage of computation. None of the computation will be delayed, but there will also be no repetition of checks. Figure 3.6 gives an example of such an implementation for the derived rule that adds a term to the assumptions. Figure 3.7 gives an alternative implementation in which some of the computation is delayed at the cost of duplicating the essential validity checks.

The behaviours of `MP`, `DISCH` and `ASSUME` are not important to the discussion. It is sufficient to know that they are three of the primitive inference rules. Each will make checks on the form of the terms and will construct a number of intermediate terms that do not contribute to the final theorem. This is inevitable if all proof is to be done using primitive inferences, but it is possible to bypass this security temporarily in order to get a result quickly and hence proceed with the proof.

```

let ADD_ASSUM t lth =
  MP (DISCH t lth) (ASSUME t) ? failwith 'ADD_ASSUM';;

```

Figure 3.6: A standard lazy derived rule

```

let ADD_ASSUM t lth =
  if (type_of t = bool_ty)
  then let (gamma,conc) = dest_lazy_thm lth
        in let gl = (union [t] gamma,conc)
            in mk_lazy_thm
                (gl,(λ(). prove_lazy_thm (MP (DISCH t lth) (ASSUME t))))
  else failwith 'ADD_ASSUM';;

```

Figure 3.7: A lazy derived rule with increased delay

During the *initial* stage of computation the implementation in Figure 3.7 does the essential checks and term construction only once. (The only check required is that the term to be added to the assumptions is Boolean-valued.) For the purposes of the initial computation this makes the implementation into a primitive rule because a lazy theorem can be generated without all the corresponding primitive inferences being performed. However, the implementation does not use `abs_thm`, so the rule is *derived* with respect to generating real theorems, and so cannot breach the security of the system. For a simple derived rule like `ADD_ASSUM` which only uses three primitive rules, the reduction in the time to perform the initial computation may not be significant. However, for derived rules that use tens or hundreds of primitive rules it will be. The price to pay is that the computation done in the initial stage is an addition to the total computation.

Note that all the tests necessary to ensure validity are performed in the initial computation, so it is reasonable to expect that a real theorem will be generated when the justification function is applied. It is also important that lazy rules fail in the initial stage of computation whenever their non-lazy counterparts would fail because some programming strategies exploit failure, e.g., backtracking.

3.6 Lazy Tactics

For the HOL system, the definitions of lazy versions of tactics and tacticals (see Section 1.6.5) follow closely the implementations of the non-lazy versions. The interactive subgoal package has to be modified to use lazy theorems instead of theorems, but this requires little more than changing the ML data types. There is an advantage in having tactics operate over lazy theorems. In the conventional (non-lazy) HOL

system, theorems are often proved during the initial application of a tactic. A good example of this is the rewriting tactics. The computation involved in generating these theorems is wasted if the branch of the proof is aborted. However, in a lazy system the theorem generation does not have to take place until later, so when the branch is aborted the computation involved is never performed.

3.7 Three Modes of Operation

It is simple to modify the abstract data type for lazy theorems so that the theorem prover can operate in different modes. Three modes that have been implemented are *lazy*, *eager* and *draft*. Lazy mode has been the main topic of this chapter so far. In eager mode theorems are generated immediately and stored as proved theorems within the representation for lazy theorems. In draft mode the justification functions are not maintained within the lazy theorems. A dummy function is used instead. This allows a proof to be tested without theorem generation taking place, but the lazy theorem produced at the end will not give rise to a real theorem. Lazy mode delays the computation and indeed the computation need never be performed. However, draft mode has the advantage that because it does not retain the justification functions, it is less likely that memory will be exhausted. In addition, garbage collection overheads are reduced.

Eager mode allows the lazy version of the theorem prover to be used as an ordinary system with full justification of theorems taking place immediately (though with the theorems wrapped up as lazy theorems). More interestingly, draft mode can be combined with one of the other modes in two passes through a proof. The proof can be developed in draft mode to obtain a lazy theorem. Then if the user is concerned about security the proof can be repeated non-interactively in lazy or eager mode. The proof script developed in draft mode will work without modification in the other modes. The only changes required are the setting of the mode at the beginning of the session, and possibly the generation of a real theorem from the lazy theorem at the end of the session.

Another possibility is to arrange for proof procedures which benefit from laziness to select lazy mode for themselves, then restore the mode to the original state at the end of their execution. Such procedures can operate lazily in both lazy and eager modes without intervention from the user. Moreover, in eager mode there is little chance of memory being exhausted because the use of laziness is restricted.

Hanna and Daeche [HD92] describe three development orderings for formal synthesis of hardware that resemble the three modes of operation. A design tree is developed by applying *techniques* (which are similar to tactics). The process involves both refinement of behavioural specifications and the proof of propositions. Users are free to choose the order in which they tackle these tasks. A *prudent* order is to perform the proofs first. In an *exploratory* order, the design is developed first. Never doing the proofs is referred to as a *reckless* order. Compare these to the eager, lazy, and draft modes, respectively. The Nuprl system also has modes of operation analogous to eager mode and draft mode. This is discussed in Section 6.5.

3.8 A Complete Lazy System

Lazy theorems would not be used everywhere in the theorem prover. In HOL, for example, only fully-justified theorems can be stored in theory files. It would probably be best for this to continue to be the case in a lazy system. Allowing lazy theorems to be stored might encourage users to neglect justification. It is important for security that they realise that a real theorem, not a lazy theorem, must be generated before they can be sure of the formula's validity. In principle a lazy theorem could be stored in a file with a marker to label it as unjustified, but this would involve the storing of functions (the justification functions). At least one implementation of Standard ML has this capability in the form of persistent storage.

It is, of course, possible to do away with the justification functions entirely and simply mark a theorem as justified or unjustified. Then, just as with the draft mode described above, proof scripts could be reprocessed off-line when full justification is required. However, this does not allow laziness to be used locally. Some procedures benefit from the ability to postpone justification until it is known whether or not the theorem is to be used. (See, for example, Section 5.2.5.) In a lazy framework this is possible without forcing the user to reprocess the proof script.

Insisting that theorems be fully justified before storing them in theory files raises some issues concerning the interaction between real and lazy theorems. Most of the functions in HOL would be modified to work with lazy theorems. However, those that access the theory files would continue to deal with real theorems, as would the functions for defining new logical types and constants because these access the theory as a side-effect. Functions that duplicate the theory-file operations but also convert between lazy and real theorems might also be provided. Of course, any such function that converts from a lazy to a real theorem might take a long time to execute because it would be performing the justification stage of the proof.

3.9 Laziness and Verified Rules

In his report on HOL [Mus89], Musser proposes the use of tactics that do not use rules to justify their operation but have been proved to be valid. He suggests that a mixture of proved and unproved tactics should be allowed, with only the unproved tactics having a justification step.

For most derived rules (or tactics) it will not be worth the effort of verifying an optimised version of the code (see Section 1.5.1). However, in a lazy HOL system the speed of the optimised version can still be exploited by using it to perform the initial computation, and so obtain a structure for the theorem more quickly. For derived rules that are used a lot, the verification may be worthwhile. Verified rules can also be used in a lazy system. Thus, the lazy approach is complementary to the use of verified derived rules as new primitive rules.

	Run	GC	Total	PInfs
HOL88 Version 2.01	83.2	8.7	91.9	16054
Lazy HOL (in eager mode)	84.0	8.7	92.7	16084
Lazy HOL (total computation)	73.6	11.0	84.6	12296
Lazy HOL (initial computation)	61.7	8.8	70.5	0
Lazy HOL (total) / HOL88	88%	126%	92%	
Lazy HOL (initial) / HOL88	74%	101%	77%	

Table 3.1: Results for the multiplier benchmark

3.10 Results

Version 2.01 of the HOL88 system has been modified to incorporate the lazy features described in this chapter. Having introduced lazy versions of the primitive inference rules, the modification was straightforward. The lazy implementation will be referred to as Lazy HOL.

Table 3.1 gives the results of running the HOL benchmark in HOL88 and Lazy HOL. Figures are given for both the total and the initial computation in the lazy mode of Lazy HOL. The latter is the time to obtain a result without generating the theorem for it. The benchmark is a verification of a multiplier circuit. Part of the proof involves the use of rewriting and some specialised equational reasoning, but much of it is general tactic-based reasoning. The times given are for proof. The times taken to make definitions and for input and output are not included.

The results in Table 3.1 were obtained on a SparcStation ELC with 40 Mbytes of real memory. Both versions of HOL were built using Austin Kyoto Common Lisp. Similar percentage figures have been obtained for a SparcStation 2 with 96 Mbytes of memory and a SparcStation SLC with 16 Mbytes, though the relative performance of Lazy HOL begins to deteriorate in the latter case.

The table should be interpreted as follows. ‘Run’ is the run time in seconds. This does not include garbage collection time. The garbage collection time in seconds is given separately and is labelled ‘GC’. The total time is also given (labelled ‘Total’). ‘PInfs’ is the number of applications of primitive inference rules used to prove the theorems. The inferences are considered to have taken place only when the real theorems have been generated.

As can be seen from the table, only about one quarter of the computation is delayed for the multiplier benchmark. This is somewhat disappointing. However, so far little attempt has been made to exploit laziness in the derived inference rules of HOL. In addition, it is difficult to measure the increase in productivity brought about by delaying the justification in tactics since this only manifests itself in interactive sessions when the user may be exploring unsuccessful avenues of proof.

The benefits of delaying justification can be much greater for specialised proof procedures. This is illustrated by results for a linear arithmetic decision procedure in Section 5.2.7 and a symbolic compiler (Section 5.3). As described in the next

chapter, delaying justification also enables certain kinds of optimisation to be made in equational reasoning. The figures for the benchmark in Lazy HOL improve under these optimisations but the change is not dramatic since only part of the proof is equational reasoning.

Despite the disappointing figures for delay, laziness is an overall optimisation for the benchmark. There is a significant decrease in the number of primitive inferences and the run time is reduced. The latter is partly counteracted by an increase in garbage collection. It is also worth noting that the performance of Lazy HOL in eager mode is not significantly worse than that of HOL88. Thus, there is little to be lost from using the Lazy HOL system whether or not laziness is exploited. The slight discrepancy between the number of primitive inferences performed in HOL88 and in eager mode can be put down to small changes in the inference rules introduced when they were made lazy.

Chapter 4

Equational Reasoning

Equational reasoning is a major feature of theorem proving. It involves proving theorems that state an equivalence between two terms. This manifests itself most commonly as rewriting, but many other theorem proving operations also involve equational reasoning. Due to the special form of the theorems generated by equational reasoning, there is more scope for optimising the proof generation than for arbitrary theorems. The prevalence of equational reasoning combined with the scope for optimisation has led to fruitful research into efficient equational reasoning in fully-expansive theorem provers. That research is presented in this chapter.

Before embarking on a discussion of techniques for efficiency, the basic principles of equational reasoning and the causes of inefficiency are presented.

4.1 Overview of Equational Reasoning

4.1.1 Equivalence Relations

A *binary relation* R on sets X and Y is a subset of the cartesian product $X \times Y$. For $x \in X$ and $y \in Y$, the statement xRy is taken to be true if the pair (x, y) is an element of R , and is false otherwise. Relations on three or more sets can be defined in a similar way, but the discussion here will be restricted to binary relations. The above definition of a binary relation is the set-theoretic view. In HOL it is usual to consider a binary relation to be a function of two arguments which returns a Boolean value. Instead of sets, the relation is defined for HOL types.

An *equivalence relation* R is a binary relation for which both arguments are taken from the same set (or type) and which satisfies the following properties:

Reflexivity	$\forall x. xRx$
Symmetry	$\forall x y. xRy \Rightarrow yRx$
Transitivity	$\forall x y z. xRy \wedge yRz \Rightarrow xRz$

Informally these properties state that: Any element is related to itself; if x is related to y then y is related to x ; and, if x is related to y and y is related to z then x is related to z . These are the properties one would normally expect of an ‘equality’.

The higher-order logic implemented by the HOL system has a built-in equivalence relation, denoted by '='. Actually, it is a function that represents the relation by mapping pairs of values to true if they are in the relation or to false otherwise. It is polymorphic and takes its arguments separately rather than as a pair. Thus, its most general type is " $* \rightarrow (* \rightarrow \text{bool})$ ".

4.1.2 Congruences

One normally expects an 'equality' to be more than just an equivalence relation. If two expressions are 'equal', one expects to be able to substitute one for the other when it appears as a subterm of a term. The equality in HOL has just such a property, expressed by the following *congruence* rules:

$$\frac{\Gamma \vdash x = y}{\Gamma \vdash f x = f y} \text{ (AP_TERM)} \quad \frac{\Gamma \vdash f = g}{\Gamma \vdash f x = g x} \text{ (AP_THM)}$$

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x. t_1) = (\lambda x. t_2)} \text{ } x \text{ is not free in } \Gamma \text{ (ABS)}$$

It may also be possible to derive congruence rules for user-defined equivalence relations. An equivalence relation equipped with congruence rules is known as a *congruence relation*.

4.1.3 Rewriting

Informally, an *equation* is a term consisting of the application of an equality to two terms of the same type. In the context of the HOL system it is useful to consider an equation to be a theorem of the form:

$$\vdash l E r$$

where E is a congruence relation. Such a theorem can be used as a *rewrite rule*:

$$l \longrightarrow r$$

by matching the left-hand side of the equation against a term and instantiating the right-hand side appropriately. For example, the theorem:

$$\vdash x + y = y + x$$

can be matched against the term " $(a * b) + (c + d)$ " so that x is instantiated with the term " $a * b$ " and y is instantiated with " $c + d$ ". The free variables of the theorem are implicitly universally quantified, so the theorem is equivalent to:

$$\vdash \forall x y. x + y = y + x$$

Specialising the bound variables as determined by the matching, the following theorem is obtained:

$$\vdash (a * b) + (c + d) = (c + d) + (a * b)$$

This process is known as *rewriting*. When congruence rules are available the rewriting can be done *at depth*. For example, specialising the free variables of the rewrite rule with "c" and "d" yields the theorem:

$$\vdash c + d = d + c$$

Using the congruence rule for the operand of a function application (`AP_TERM`) it is possible to obtain:

$$\vdash (a * b) + (c + d) = (a * b) + (d + c)$$

Thus, in effect, the term " $(a * b) + (c + d)$ " has been rewritten by applying the rewrite rule to the second argument of the top-level sum.

A rewrite rule may also be *conditional*; that is, the application of the rule is dependent on some formula, c , being satisfied. As a theorem, a conditional rule may appear in the form of an implication:

$$\vdash c \Rightarrow (l E r)$$

or with the condition in the hypotheses:

$$c \vdash l E r$$

For the rule to be applicable to a term t , the left-hand side, l , must match t and c must be provable when instantiated using the binding generated by the match. For example, the following rule is not applicable to the term "0/0" because when instantiated with 0 the condition is false:

$$\vdash (x \neq 0) \Rightarrow (x/x = 1)$$

For an in-depth discussion of equations and rewrite rules, see the survey by Huet and Oppen [HO80].

4.1.4 Substitution

The built-in equality in HOL gives rise to a notion of substitution. Conceptually, *substitution* is the replacement of free variables by terms of the same type. In higher-order logic it is possible to generalise this rule to the replacement of arbitrary subterms with other terms. For efficiency reasons, the HOL system takes this derived notion of substitution as its primitive. It is very much like the rewriting described in Section 4.1.3.

Restrictions have to be placed on what substitutions can be made because of problems with the capture of free variables when substituting inside λ -abstractions. In practice, more restrictions than really necessary may be imposed for implementation reasons. Slind discusses the issues in detail in his Master's thesis [Sli91].

The substitution primitive in HOL implements simultaneous parallel substitution using multiple equations. The terms to be replaced must not contain any bound variables, i.e., substitution cannot in general occur inside abstractions. Bound variables are automatically renamed to prevent free variables in the right-hand sides becoming bound when substituted inside an abstraction. It is consistent with the λ -calculus notion of substitution to not allow bound variables to be replaced since such variables should not be visible.

The generalised substitution in HOL is one of its primitive rules. It is therefore reasonable to suppose that it might be more efficient to implement depth rewriting using this substitution primitive than using the congruence rules. This would only work for abstraction-free terms, of course, because of the interplay of substitution with bound variables. One could, however, conceive of a rewriting function which used substitution for abstraction-free subterms combined with the congruence rule for abstractions as 'glue'. Nor would it be unreasonable to imagine a version of HOL in which the substitution primitive does allow substitution inside abstractions, provided the bound variable does not appear free in the hypotheses of the (equational) theorem being used. Such a rule would, in effect, be a combination of the congruence rules listed in Section 4.1.2.

Substitution (in whatever form) could only be used efficiently for the built-in equality, since for user-defined equivalence relations there is no primitive substitution rule. The use of substitution for equational reasoning is considered further in Section 4.6.6.

4.2 Conversions

Equational reasoning in HOL is implemented using ML functions called *conversions* as introduced by Paulson [Pau83]. Paulson's conversions have the ML type:

$$\text{term} \rightarrow \text{thm}$$

which is abbreviated as `conv`. So, conversions are functions from terms to theorems, but they also have the property that for an argument t , they return a theorem of the form:

$$\vdash t = t'$$

That is, the theorem is an equation, and the left-hand side is the argument term. Paulson describes some basic conversions and some functions for combining these to form new conversions, thus allowing sophisticated rewriting strategies to be developed.

Figure 4.1 gives some of the code for the HOL implementation of conversions. The basic functions given can be used to compose basic conversions into more sophisticated ones. Examples of basic conversions are application of a rewrite rule to a term, and beta-reduction. `ALL_CONV` is the identity conversion which leaves a term unchanged; it is defined to be the reflexivity rule, `REFL`. `NO_CONV` is a conversion which always fails. `THENC` and `ORELSEC` are both infix functions and are used for sequencing and alternating conversions respectively. `THENC` takes two conversions and produces a conversion that applies them in succession. The results are combined using the transitivity rule, `TRANS`. `ORELSEC` applies the first conversion, but if this fails the second is applied.

`RATOR_CONV`, `RAND_CONV` and `ABS_CONV` are the ‘congruence’ conversions, that is they implement the congruence rules `AP_THM`, `AP_TERM` and `ABS` (Section 4.1.2) as conversions. The function `RATOR_CONV` applies a conversion to the operator of a function application (combination), and `RAND_CONV` applies a conversion to the operand. `ABS_CONV` applies a conversion to the body of an abstraction, but fails if the bound variable is free in the hypotheses of the theorem returned by the conversion. This failure is rare because it is unusual for conversions to return theorems with hypotheses.

In addition to `RATOR_CONV` and `RAND_CONV` it is useful to have a function `COMB_CONV`, which applies a conversion to both the operator and operand of a function application. This can be defined in terms of `RATOR_CONV`, `RAND_CONV` and `THENC`, but it is more efficient to make it a basic conversion function. The definition is given in Figure 4.2. The `MK_COMB` rule is a combination of the two congruence rules for function applications. It can be more efficient than separate calls to `AP_TERM` and `AP_THM`.

These basic conversions can be used to define term traversal strategies for rewriting (depth rewrites; see Section 4.1.3). First it is convenient to define a function for applying a conversion to all the subterms of a term (Figure 4.3). If the term is a function application, `COMB_CONV` is used. If it is an abstraction, the conversion is applied to the body. Otherwise, the term is left as it is.

The function `ONCE_DEPTH_CONV` (Figure 4.4) implements a top-down traversal in which the conversion is applied at most once to any subterm. The function is defined recursively. It attempts to apply the conversion to the entire term. If this succeeds, nothing more is done. Otherwise, the function is applied recursively to the subterms. In the event that the recursive call fails, the term is left unchanged.

4.3 An Example Using Depth Rewriting

Suppose a user wants to rewrite the term:

$$\lambda n. (n * 0) + (n * 4)$$

using the following equations:

$$x * y = y * x \tag{4.1}$$

$$0 * x = 0 \tag{4.2}$$

$$0 + x = x \tag{4.3}$$

```

let ALL_CONV = REFL;;

let NO_CONV:conv = λtm. failwith 'NO_CONV';;

let (conv1 THENC conv2):conv =
  λtm. let th1 = conv1 tm
        in let th2 = conv2 (rhs (concl th1))
        in th1 TRANS th2;;

let (conv1 ORELSEC conv2):conv = λtm. conv1 tm ? conv2 tm;;

let RATOR_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RATOR_CONV'
  in AP_THM (conv rator) rand;;

let RAND_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RAND_CONV'
  in AP_TERM rator (conv rand);;

let ABS_CONV conv tm =
  let (bv,body) = dest_abs tm ? failwith 'ABS_CONV'
  in let bodyth = conv body
  in (ABS bv bodyth ? failwith 'ABS_CONV');;

```

Figure 4.1: Original conversions for HOL

```

let COMB_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'COMB_CONV'
  in MK_COMB (conv rator, conv rand);;

```

Figure 4.2: Function for applying a conversion to both operator and operand

```

let SUB_CONV conv tm =
  if (is_comb tm) then COMB_CONV conv tm
  if (is_abs tm) then ABS_CONV conv tm
  else ALL_CONV tm;;

```

Figure 4.3: Function for applying a conversion to subterms

```

letrec ONCE_DEPTH_CONV conv tm =
  (conv ORELSEC (SUB_CONV (ONCE_DEPTH_CONV conv)) ORELSEC ALL_CONV)
  tm;;

```

Figure 4.4: A depth conversion

The user could indicate explicitly where, within the term, rewrites are to take place, but even for a small term that is very tedious (unless a graphical user interface is available). It is therefore usual in HOL to use a function such as `ONCE_DEPTH_CONV` that applies the specified rewrites everywhere it can within the term. So, for the example, the first thing to do is to apply the commutativity of multiplication (Equation 4.1). The $(0 * n)$ can then be reduced using Equation 4.2, which in turn allows Equation 4.3 to be applied. The end result is $(\lambda n. 4 * n)$.

The structure of the example term is given in Figure 4.5. Figure 4.6 lists the theorems generated for the three rewriting operations using `ONCE_DEPTH_CONV`. The figure is in four parts, the first three of which are subdivided into theorems generated by the rewrites and theorems to rebuild the term structure around them. The first three parts correspond to the three rewriting steps. The fourth consists of the two applications of the transitivity rule that are required to obtain the overall result from the three steps. These are introduced by using `THENC` (Section 4.2) to sequence the depth conversions.

When a subterm is unchanged by the rewriting function a theorem is not generated directly using a reflexivity rule, but is instead constructed from theorems for the leaf nodes. This behaviour is a consequence of using the most straightforward algorithm:

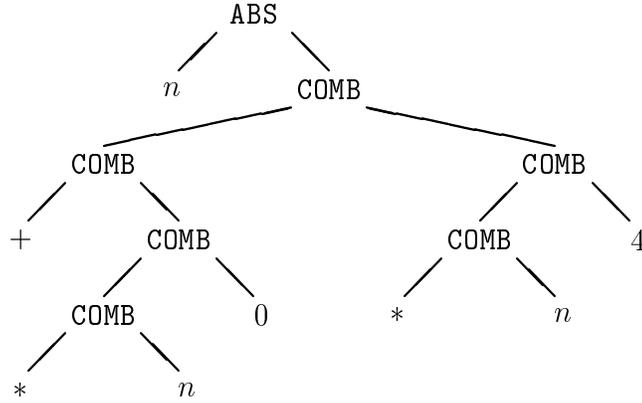
1. If a rewrite is applicable to the term, apply it; otherwise,
2. if the term is a variable or a constant apply the reflexivity rule; otherwise,
3. recursively apply the algorithm to the body of the abstraction or the operator and operand of the combination (as appropriate), then combine the results.

This algorithm is a consequence of the definition of `ONCE_DEPTH_CONV` and the other conversions used.

The example illustrates two of the inefficiencies identified in Chapter 2:

- application of inference rules when a subterm has not been changed;
- repeated traversals of a term when one would be sufficient.

The first of these inefficiencies can be avoided by means of a more delicate rewriting algorithm (Section 4.4). The second really requires proof steps to be combined and reordered, for which it is necessary to delay some of the computation. Section 4.6 describes an extension to the techniques of Chapter 3 that allows this.

Figure 4.5: The structure of the term " $\lambda n. (n * 0) + (n * 4)$ "

1	$\vdash n * 0 = 0 * n$	Rewriting with Equation 4.1
2	$\vdash n * 4 = 4 * n$	Rewriting with Equation 4.1
3	$\vdash + = +$	Reflexivity
4	$\vdash (+ (n * 0)) = (+ (0 * n))$	MK_COMB on 3 and 1
5	$\vdash (n * 0) + (n * 4) = (0 * n) + (4 * n)$	MK_COMB on 4 and 2
6	$\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. (0 * n) + (4 * n)$	ABS on 5
7	$\vdash 0 * n = 0$	Rewriting with Equation 4.2
8	$\vdash + = +$	Reflexivity
9	$\vdash (+ (0 * n)) = (+ 0)$	MK_COMB on 8 and 7
10	$\vdash * = *$	Reflexivity
11	$\vdash 4 = 4$	Reflexivity
12	$\vdash (* 4) = (* 4)$	MK_COMB on 10 and 11
13	$\vdash n = n$	Reflexivity
14	$\vdash 4 * n = 4 * n$	MK_COMB on 12 and 13
15	$\vdash (0 * n) + (4 * n) = 0 + (4 * n)$	MK_COMB on 9 and 14
16	$\vdash \lambda n. (0 * n) + (4 * n) = \lambda n. 0 + (4 * n)$	ABS on 15
17	$\vdash 0 + (4 * n) = 4 * n$	Rewriting with Equation 4.3
18	$\vdash \lambda n. 0 + (4 * n) = \lambda n. 4 * n$	ABS on 17
19	$\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. 0 + (4 * n)$	Transitivity on 6 and 16
20	$\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. 4 * n$	Transitivity on 19 and 18

Figure 4.6: Theorems generated for conventional rewriting

4.4 Optimisation Using Exceptions

The application of inference rules when a subterm has not been changed can be avoided by propagating a value representing ‘unchanged’ back up the term structure. This value is generated at any leaf node (a variable or a constant) that is unchanged. At an abstraction the value is allowed to propagate. At a function application the value is only allowed to propagate when ‘unchanged’ is obtained for both the operator and the operand. This modification to the algorithm is due to Roger Fleming of Hewlett Packard Labs, Bristol, England. He developed a rewriting package for HOL which exploited ML exceptions to simply and efficiently encode the ‘unchanged’ value. An exception is raised at any unchanged leaf node and caught at application nodes.

Fleming’s idea can be extended to apply to equational reasoning in general by reimplementing the basic conversions so that they raise and trap an ‘unchanged’ exception (Figure 4.7). The new conversions have the property that they may generate an ‘unchanged’ exception rather than return a theorem. This simply means that the argument term has not been changed by the conversion. A ‘wrapper’ function `U CONV` is provided to trap the exception and return a theorem of the form $\vdash t = t$ instead by applying a *single* primitive inference rule (`REFL`). `U CONV` should only be used when it is absolutely necessary to have a theorem, i.e., when there is no more equational reasoning to be done, since it breaks the optimisation.

The ‘unchanged’ exception is raised by the identity conversion `ALL_CONV` and by other basic conversions which can leave terms unchanged.

In the implementation of `THENC`, if the application of the first conversion does not change the term, an exception will be raised. This is caught by the trap in the last line, and the second conversion is then applied to the *original* term. Since the first conversion left the original term unchanged, the result will be correct, but it is obtained without having to generate a theorem for the result of the first conversion. If the first conversion does change the term, but the second one does not, another exception trap causes the result of applying the first conversion to be returned as the full result.

The alternation operator on conversions, `ORELSEC`, is the most delicate of the new definitions. It is meant to trap the exceptions that are generated because of errors. The syntax of ML makes it necessary to trap *any* exception raised from the application of the first conversion. The notation ‘`?λs`’ traps all exceptions and binds the exception string to the variable `s`. If the exception is ‘unchanged’, it is regenerated. Otherwise the result of the `ORELSEC` becomes the result of applying the second conversion to the original term. So, `ORELSEC` applies the first conversion and if an exception is raised because of an error the second conversion is applied instead. If the first conversion simply leaves the term unchanged then the ‘unchanged’ exception used to indicate this is allowed to propagate.

The definitions of `RATOR_CONV`, `RAND_CONV` and `ABS_CONV` are unchanged. Note, however, that the application of `conv` in each case must not be enclosed within an exception handler, else the ‘unchanged’ will not be able to propagate. In fact, it would cause an exception intended as an error message to be raised instead. The ex-

```

let ALL_CONV:conv = λtm. failwith 'unchanged';;

let NO_CONV:conv = λtm. failwith 'NO_CONV';;

let (conv1 THENC conv2):conv =
  λtm. (let th1 = conv1 tm
        in ((th1 TRANS (conv2 (rhs (concl th1))))
           ??['unchanged'] th1))
       ??['unchanged'] (conv2 tm));;

let (conv1 ORELSEC conv2):conv =
  λtm. (conv1 tm)
       ?λs if (s = 'unchanged')
            then (failwith 'unchanged')
            else (conv2 tm));;

let RATOR_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RATOR_CONV'
  in AP_THM (conv rator) rand;;

let RAND_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RAND_CONV'
  in AP_TERM rator (conv rand);;

let ABS_CONV conv tm =
  let (bv,body) = dest_abs tm ? failwith 'ABS_CONV'
  in let bodyth = conv body
    in (ABS bv bodyth ? failwith 'ABS_CONV');;

let UCONV conv tm = (conv tm) ??['unchanged'] (REFL tm);;

```

Figure 4.7: Conversions using exceptions

```

let th1 = conv1 tm
in let tm1 = rhs (concl th1)
in if (is_neg tm1)
    then th1 TRANS (RAND_CONV conv2 tm1)
    else th1 TRANS (conv2 tm1)

```

Figure 4.8: Selective application of a conversion

ception can propagate for `RATOR_CONV` and `RAND_CONV` because they only affect either the operator or the operand of an application, never both. So, if an ‘unchanged’ exception is raised for the part that can be affected, the whole application must be unchanged.

In theory, there is nothing to stop a programmer using ‘unchanged’ as the name of an exception, and hence interfere with the workings of the optimisation. In practice, however, this is very unlikely. Standard ML has a much cleaner exception mechanism than the ML used in HOL88, so is better suited to coding this optimisation.

So, the new implementations of the basic conversion functions behave in the same way as the originals (provided `UONV` is used at the top level), but are more efficient. The code for depth conversions remains unchanged since these conversions are implemented in terms of the basic functions. This also applies to the majority of users’ code. Thus, the optimisation is largely transparent to the programmer and user.

One way in which the technique is not a transparent change is if the programmer wishes to perform some special operation within a conversion as opposed to building it entirely from other conversions. In such a case, the programmer may wish to bind an intermediate result using a `let`-expression. The code fragment in Figure 4.8 is an example of this. The right-hand side of the result of applying `conv1` is obtained explicitly so that its form can be examined before applying `conv2`. If the term is a negation then `conv2` is applied only to the argument of the negation, rather than to the whole term.

The problem with the code is that the application of `conv1` may raise an ‘unchanged’ exception instead of returning a theorem. As the code stands, this exception will escape from the `let`-expression without `conv2` being applied. The obvious solution is to ensure that a theorem is generated by applying `UONV`. However, this destroys the optimisation. Another solution is to introduce an explicit exception trap but this is messy and the optimisation is no longer transparent. The code for applying `conv2` has to be duplicated (though not exactly) as shown in Figure 4.9. In the trap code it is known that the original term is the same as the term that would have been bound to the variable `tm1`.

Close inspection reveals that the exception traps introduced are similar in form to those appearing in the definition of `THENC`. This is not surprising since two conversions are being sequenced. In fact, the code can be rewritten using `THENC` in such a way that explicit handling of exceptions is not required (Figure 4.10). However,

```

(let th1 = conv1 tm
 in let tm1 = rhs (concl th1)
 in if (is_neg tm1)
      then (th1 TRANS (RAND_CONV conv2 tm1) ??['unchanged'] th1)
      else (th1 TRANS (conv2 tm1) ??['unchanged'] th1)
) ??['unchanged']
  (if (is_neg tm)
   then RAND_CONV conv2 tm
   else conv2 tm)

```

Figure 4.9: Selective application with exceptions

```

(conv1 THENC ( $\lambda$ tm1. if (is_neg tm1)
                      then RAND_CONV conv2 tm1
                      else conv2 tm1))
tm

```

Figure 4.10: Selective application using basic conversions

this is not always practical.

In the next section an alternative implementation of conversions is presented that allows an ‘unchanged’ value to be propagated without using exceptions.

4.5 Optimisation Using an ML Data Type

An alternative to using exceptions for optimisation of equational reasoning is to use an ML data type such as the one in Figure 4.11. A value of this type is either a theorem or an indicator that the original term has not been changed by the conversion. The definitions of the basic conversion functions for this new type are given in Figure 4.12. A slight variation is for the `Unchanged` constructor to take the unchanged term as an argument. Without this information an `equation` is incomplete as a stand-alone object; the structure it represents may not be known outside of the context of the conversion that created it.

```

type equation = Equation of thm | Unchanged

```

Figure 4.11: ML data type for optimisation

```

let ALL_CONV:conv = λtm. Unchanged;;

let NO_CONV:conv = λtm. failwith 'NO_CONV';;

let (conv1 THENC conv2):conv =
  λtm. let eq1 = conv1 tm
        in case eq1
          of (Equation th1) .
              (let eq2 = conv2 (rhs (concl th1))
                in case eq2
                  of (Equation th2) . Equation (th1 TRANS th2)
                     | Unchanged . eq1)
              | Unchanged . (conv2 tm);;

let (conv1 ORELSEC conv2):conv = λtm. conv1 tm ? conv2 tm;;

let RATOR_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RATOR_CONV'
  in case (conv rator)
    of (Equation th) . Equation (AP_THM th rand)
       | Unchanged . Unchanged;;

let RAND_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RAND_CONV'
  in case (conv rand)
    of (Equation th) . Equation (AP_TERM rator th)
       | Unchanged . Unchanged;;

let ABS_CONV conv tm =
  let (bv,body) = dest_abs tm ? failwith 'ABS_CONV'
  in case (conv body)
    of (Equation th) . Equation (ABS bv th ? failwith 'ABS_CONV')
       | Unchanged . Unchanged;;

let UCONV conv tm =
  case (conv tm) of (Equation th) . th | Unchanged . REFL tm;;

```

Figure 4.12: Conversions using an ML data type

The disadvantage with this approach to avoiding unnecessary inferences for unchanged subterms is that conversions no longer return theorems. Their new type is:

```
term → equation
```

In respect of their types, conversions are no longer simply a special form of inference rule. Hence, conversions do not interface directly with general rules such as **TRANS**. The function **UCONV** acts as an interface by converting values of type **equation** to values of type **thm**. The type constructor **Equation** performs a conversion in the other direction. The conversion between the two types is required wherever conversions are used as rules. Adding calls to the interface functions is not difficult but the changes required to users' code are tedious. This issue is discussed at greater depth in Section 4.8.

4.6 Laziness Enables Further Optimisation

Once the move has been made to using a new ML data type to represent the result of applying a conversion, it is straightforward to extend the data type to provide optimisation of repeated traversals over the term. The optimisation requires the application of inference rules to be delayed. This is similar to the approach described in Chapter 3 but is specialised for equational reasoning.

4.6.1 Lazy Structures

Delaying the justification stage of a proof (Chapter 3) opens up possibilities for non-local optimisations. For general logical reasoning the individual proof steps are too ad hoc for this to be exploited without incurring unreasonable overheads. However, in certain restricted areas non-local optimisation is practical. In order to support non-local optimisations, 'laziness' must be represented by a concrete structure rather than a function. The lazy structure for equational reasoning in HOL is given in Figure 4.13. A **lazy_eq** is essentially the result of a rewrite. A **(lazy_eq)list** is a sequence of rewrites. Sequencing may occur at any level in the term structure.

```

rectype lazy_eq = Lazy_eq_thm of void → thm
                | Lazy_comb of (lazy_eq)list × (lazy_eq)list
                | Lazy_abs of (lazy_eq)list

```

Figure 4.13: ML type of lazy structures for equational reasoning

A **lazy_eq** structure mimics the structure of HOL terms (see Section 1.6.1). There are internal nodes for function applications (combinations) and λ -abstractions

1	$\vdash n * 0 = 0 * n$	Rewriting with Equation 4.1
2	$\vdash n * 4 = 4 * n$	Rewriting with Equation 4.1
3	$\vdash 0 * n = 0$	Rewriting with Equation 4.2
4	$\vdash 0 + (4 * n) = 4 * n$	Rewriting with Equation 4.3
5	$\vdash n * 0 = 0$	Transitivity on 1 and 3
6	$\vdash (+ (n * 0)) = (+ 0)$	AP_TERM on 5
7	$\vdash (n * 0) + (n * 4) = 0 + (4 * n)$	MK_COMB on 6 and 2
8	$\vdash (n * 0) + (n * 4) = 4 * n$	Transitivity on 7 and 4
9	$\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. 4 * n$	ABS on 8

Figure 4.14: Theorems generated using lazy structure

(`Lazy_comb` and `Lazy_abs` respectively), and a leaf node for subterms that have had a basic rewrite applied to them (`Lazy_eq_thm`). At each internal node a sequence of rewrites can occur.

When used as the result of conversions instead of a theorem, the lazy structure allows the same optimisations as the techniques in Sections 4.4 and 4.5. An empty `lazy_eq` list indicates that there has been no change. The lazy structure provides additional optimisations, as can be seen in Figure 4.14, which is based on the example in Section 4.3. Only five theorems are generated to rebuild the structure instead of sixteen for the original implementation of rewriting. To achieve this optimisation it is necessary to delay the generation of theorems. The figure lists the theorems generated using an algorithm acting over the lazy structure. The repeated traversals of the term and subterms are eliminated. The algorithm is described in Section 4.6.3. Before that the implementations of the basic conversion functions for the lazy structure are presented (Section 4.6.2).

Only the term traversal need be delayed to achieve the optimisation, so it is not necessary to use a function with a dummy argument in `Lazy_eq_thm`. A fully-evaluated theorem could be used instead. However, a function is used so as to obtain the general delay provided by lazy theorems in addition to the optimisation of equational reasoning. So, there are two levels of delay in the lazy structure.

4.6.2 Lazy Conversions

In order to provide the performance benefits of the lazy structure and the ease of programming offered by conversions, the latter have been modified to return a lazy structure together with a representation of the theorem as terms. The type of these *lazy conversions* is:

$$\text{term} \rightarrow (((\text{term})\text{list} \times \text{term} \times \text{term}) \times (\text{lazy_eq})\text{list})$$

The first component of the result is a triple consisting of a list of hypotheses generated by the rewrites, the left-hand side of the equation, and the right-hand side.

The two sides of the equation are maintained as separate terms for ease of programming. The term structure of the equation has to be present in addition to the lazy structure so as to allow functions to test the result. The lazy structure can not provide this information unless it is evaluated, and to do that would neutralise the optimisation.

The implementations of the basic conversion functions are given in Figure 4.15. The identity lazy conversion, `ALL_CONV`, simply takes a term and returns the same term as the result with an empty hypothesis list and an empty list of changes (`lazy_eq` list).

`THENC` uses a set-theoretic union operation to combine the hypotheses generated by the two conversions it sequences. The function `sequence_lazy_eqs` is at the heart of the optimisation. It is described in detail in Section 4.6.3. The code for `THENC` illustrates the fact that it is not necessary to return the argument term as part of the result of the conversion. In `THENC` the values returned are thrown away by pattern matching with the wildcard character (an underscore). However, in practice it is better to keep the argument term (the left-hand side of the equation) explicitly in the structure. It is required, for example, in order to print the structure as an equation.

`RATOR_CONV` and `RAND_CONV` test to see if any changes have been brought about by the conversion. If not, an empty sequence is returned as opposed to a `Lazy_comb` with empty sequences for both its arguments.

`ABS_CONV` has to test explicitly for the presence of the bound variable of the abstraction in the hypotheses (as a free occurrence) since the `ABS` rule is not applied immediately (see Section 4.1.2 for the side-condition on `ABS`). The conversion must fail immediately or its behaviour with respect to `ORELSEC` will change. For example, the conversion:

```
(ABS_CONV conv1) ORELSEC conv2
```

would produce a structure based on the application of `conv1` when it should produce one based on `conv2`. An exception would be raised when an attempt is made to generate a theorem from the structure based on `conv1`. One of the real difficulties with delaying computation is this kind of interaction with exceptions.

The conversion functions assume that the hypothesis list must be empty if no changes have occurred. Note also that no inference rules are applied. The application of inference rules takes place later, when a theorem is generated from the structure by the function `prove_lazy_eqs` (Section 4.6.4). `UCONV` calls this function to generate a theorem. It could simply return this theorem. However, it first verifies that the theorem generated corresponds to the term structure. This is a consistency check for user-defined conversions.

4.6.3 Sequencing Lazy Structures

Lazy conversions are applied in sequence using the infix function `THENC` (Figure 4.15). The lazy conversion generated by sequencing `conv1` and `conv2` takes a term and applies `conv1` to it. `conv2` is then applied to the resulting term. The function

```

let ALL_CONV:conv = λtm. (([],tm,tm), []);;

let NO_CONV:conv = λtm. failwith 'NO_CONV';;

let (conv1 THENC conv2):conv =
  λtm. let ((hyps1,_,tm1),leqs1) = conv1 tm
        in let ((hyps2,_,tm2),leqs2) = conv2 tm1
        in if (null leqs1) then ((hyps2,tm,tm2), leqs2)
           if (null leqs2) then ((hyps1,tm,tm1), leqs1)
           else ((union hyps1 hyps2,tm,tm2),
                sequence_lazy_eqs leqs1 leqs2);;

let (conv1 ORELSEC conv2):conv = λtm. conv1 tm ? conv2 tm;;

let RATOR_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RATOR_CONV'
  in let ((hyps,_,tmf),leqsf) = conv rator
  in if (null leqsf) then (([],tm,tm), [])
     else ((hyps,tm,mk_comb(tmf,rand)), [Lazy_comb(leqsf,[])]);;

let RAND_CONV conv tm =
  let (rator,rand) = dest_comb tm ? failwith 'RAND_CONV'
  in let ((hyps,_,tmx),leqsx) = conv rand
  in if (null leqsx) then (([],tm,tm), [])
     else ((hyps,tm,mk_comb(rator,tmx)), [Lazy_comb([],leqsx)]);;

let ABS_CONV conv tm =
  let (bv,body) = dest_abs tm ? failwith 'ABS_CONV'
  in let ((hyps,_,tmb),leqsb) = conv body
  in if (null leqsb) then (([],tm,tm), [])
     else if (mem bv (frees1 hypsb))
        then failwith 'ABS_CONV'
        else ((hyps,tm,mk_abs(bv,tmb)), [Lazy_abs leqsb]);;

let UCONV conv tm =
  let ((hyps,tm,tm'),leqs) = conv tm
  in let th = prove_lazy_eqs leqs tm
  in let (h,c) = dest_thm th
  in let (l,r) = dest_eq c
  in if (h = hyps) & (l = tm) & (r = tm') then th
     else failwith 'UCONV';;

```

Figure 4.15: Conversions using a lazy structure

```

letrec sequence_lazy_eq leq1 leq2 =
  case (leq1,leq2)
  of (Lazy_eq_thm f1,Lazy_eq_thm f2) .
     [Lazy_eq_thm (λ(). (f1 ()) TRANS (f2 ()))]
  | (Lazy_comb (leqs1f,leqs1x),Lazy_comb (leqs2f,leqs2x)) .
     [Lazy_comb (sequence_lazy_eqs leqs1f leqs2f,
                 sequence_lazy_eqs leqs1x leqs2x)]
  | (Lazy_abs leqs1b,Lazy_abs leqs2b) .
     [Lazy_abs (sequence_lazy_eqs leqs1b leqs2b)]
  | (_) . [leq1;leq2]

and sequence_lazy_eqs leqs1 leqs2 =
  if (null leqs1) then leqs2
  if (null leqs2) then leqs1
  if (null (tl leqs1)) then
    (sequence_lazy_eq (hd leqs1) (hd leqs2))@(tl leqs2)
  else (hd leqs1).(sequence_lazy_eqs (tl leqs1) leqs2);;

```

Figure 4.16: Sequencing lazy structures

tests to see if either of the `lazy_eq` lists are empty (indicating that the term was not changed). If this is so, the computation can be optimised. In particular, if both `lazy_eq` lists are empty then the list returned will be empty. If neither list is empty the set-theoretic union of the hypotheses is formed and the `lazy_eq` lists are combined using the function `sequence_lazy_eqs` (Figure 4.16).

Concatenating two `lazy_eq` sequences requires two mutually-recursive functions: one to traverse the first sequence until it encounters the last `lazy_eq`; the other to combine this `lazy_eq` with the first one in the second sequence. If either sequence is empty the other is returned. Two `lazy_eqs` can be merged if they are of the same kind, i.e., both abstraction nodes, both combination nodes, or both leaf nodes. In the first two cases, merging is done simply by making recursive calls to concatenate the sequences for the subterms. In the latter case, a new proof function is formed by applying the transitivity rule, `TRANS`, to the results of the original proof functions.

Thus, `sequence_lazy_eqs` constructs a new sequence of `lazy_eqs` in such a way that any repeated traversals of subterms are avoided. The mechanism is capable of merging two traversals into one provided they are not separated by a structure-destroying operation. So, two rewrites within the operator of a function application can be merged into one even when separated by a rewrite within the operand. This is reordering of rewrites.

There is a price to pay for the elimination of repeated traversals: The lazy structures have to be manipulated in addition to the theorems. So, there is extra computation and extra garbage. In fact, lazy conversions would not be an optimisation in many theorem provers because the manipulation of the lazy structures

would be all that was required. It is only because theorem generation is so costly in a fully-expansive theorem prover that the technique is worthwhile. In principle, use of lazy structures may be a deoptimisation even in HOL. However, in most cases, sufficient primitive inferences are avoided to outweigh the overheads.

4.6.4 Evaluation Using Congruence Rules

When the result of some equational reasoning is required as a theorem so that more general reasoning can be performed, the final lazy structure must be processed. This can be done by applying the congruence rules. An implementation using mutually-recursive functions is given in Figure 4.17.

The functions take a term as an argument in addition to the lazy structures. At each internal node of the structure, the term must be of a corresponding form, else an exception is raised. In practice there should be no error, but if the programmer has given the wrong term as an argument (for example) then there could be a mismatch. The exceptions are present to highlight such errors.

At each leaf node (`Lazy_eq_thm`) the proof function is evaluated and the resulting theorem is tested to make sure that its left-hand side is equal to the argument term. These theorems are combined at each of the internal nodes (`Lazy_comb` and `Lazy_abs`) using the appropriate congruence rule. Different rules are used for `Lazy_comb` nodes depending on whether one or other or both or neither of the arguments is an empty list.

Sequences are processed by `prove_lazy_eqs` using the transitivity rule, `TRANS`. An optimisation is made when the sequence has only one element. Instead of making a recursive call on an empty sequence, causing `REFL` to be applied to the term, and then using transitivity, the theorem for the single `lazy_eq` in the sequence is returned directly.

Once the move has been made to the implementation of conversions using lazy structures, freedom is acquired to generate a theorem from the structures by any sound means. Using the congruence rules is not the only option. Another possibility is to use substitution. This is discussed in Section 4.6.6, but first the notion of *sequencing depth* is introduced. This is a measure of the number of substitutions required to generate a theorem from a lazy structure.

4.6.5 Sequencing Depth

Lazy structures can be viewed as three-dimensional trees. In two of the dimensions, their structure is similar to that of logical terms. The third dimension arises from the sequencing of equational manipulations. For example, the rewriting of the term:

$$\lambda n. (n * 0) + (n * 4)$$

in Section 4.3 gives rise to the lazy structure in Figure 4.18 when lazy conversions are used. The third dimension is indicated by broken lines. A short edge in the tree represents an empty sequence.

```

letrec prove_lazy_eq leq tm =
  case leq
  of (Lazy_eq_thm proof) .
      (let th = proof ()
       in if (lhs (concl th) = tm)
          then th
          else failwith 'prove_lazy_eq')
  | (Lazy_comb (leqsf,leqsx)) .
      (let (f,x) = dest_comb tm ? failwith 'prove_lazy_eq'
       in if (null leqsf)
          then if (null leqsx)
              then REFL tm
              else AP_TERM f (prove_lazy_eqs leqsx x)
          else if (null leqsx)
              then AP_THM (prove_lazy_eqs leqsf f) x
              else MK_COMB (prove_lazy_eqs leqsf f,
                           prove_lazy_eqs leqsx x))
  | (Lazy_abs leqsb) .
      (let (var,b) = dest_abs tm ? failwith 'prove_lazy_eq'
       in if (null leqsb)
          then REFL tm
          else ABS var (prove_lazy_eqs leqsb b))

and prove_lazy_eqs leqs tm =
  if (null leqs)
  then REFL tm
  else let th = prove_lazy_eq (hd leqs) tm
       in if (null (tl leqs))
          then th
          else th TRANS
          (prove_lazy_eqs (tl leqs) (rhs (concl th)));;

```

Figure 4.17: Evaluating lazy structures

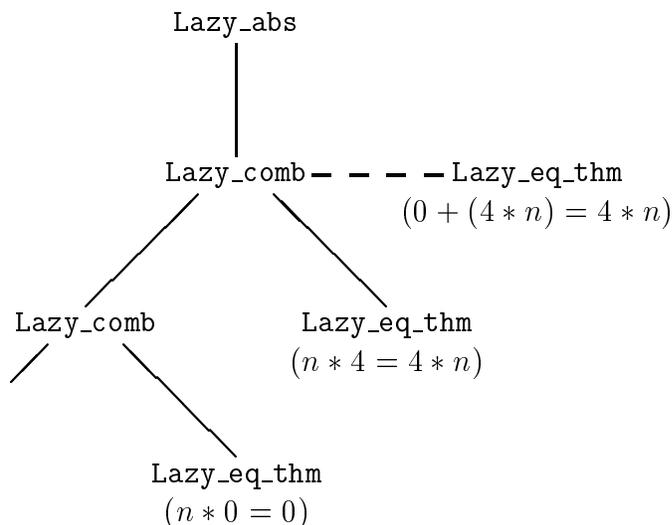


Figure 4.18: Lazy structure for the rewriting of " $\lambda n. (n * 0) + (n * 4)$ "

There are certain constraints on the order of evaluation of lazy structures. For example, the `Lazy_eq_thm` $(0 + (4 * n) = 4 * n)$ node cannot be evaluated before either of the other `Lazy_eq_thm` nodes. However, there is also some flexibility. For example, the two branches of a `Lazy_comb` node can be evaluated in either order. These flexibilities and constraints are important because they control what methods can be used for generating a theorem from a lazy structure. In particular, they control how substitution can be used.

Figure 4.19 represents a hypothetical lazy structure in which the nodes are numbered in accordance with the order of evaluation implemented by the function `prove_lazy_eqs` of Section 4.6.4. Figure 4.20 illustrates the three substitutions required to generate a theorem for the structure. The edges 1–4 and 6–8 are ‘in parallel’ because nodes 1 and 6 can be ‘evaluated’ by the same substitution (a). Edge 9–10, on the other hand, is ‘in sequence’ with edges 1–4 and 6–8 because they have to be processed as part of the evaluation of node 9. These ideas can be formalised by introducing a notion of *sequencing depth*.

The sequencing depth of a lazy structure is defined recursively as follows:

- The sequencing depth of a `(lazy_eq)list` is the sum of the depths of the constituent `lazy_eqs`;
- The sequencing depth of a `Lazy_eq_thm` node is 1;
- The sequencing depth of a `Lazy_abs` node is the depth of its argument;
- The sequencing depth of a `Lazy_comb` node is the maximum of the depths of its arguments.

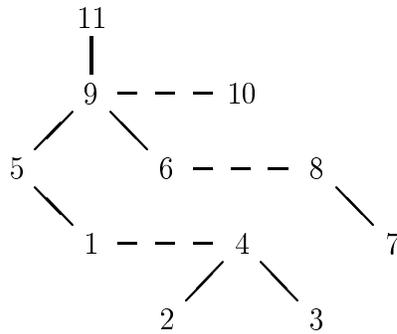


Figure 4.19: Hypothetical lazy structure

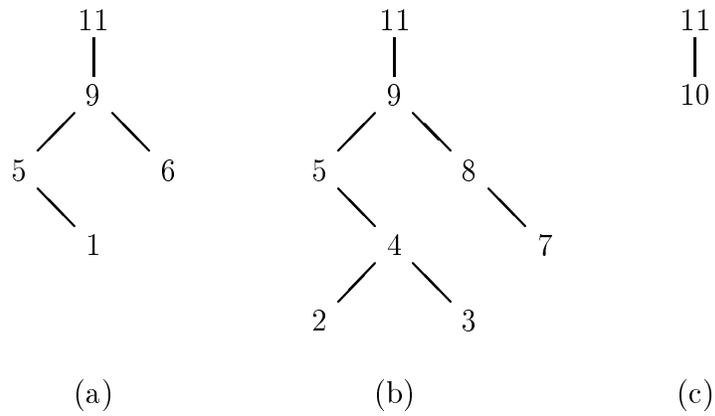


Figure 4.20: Illustration of sequencing depth

For a substitution rule that can act inside λ -abstractions and does not perform renaming, the sequencing depth specifies the number of substitutions required to generate a theorem from the lazy structure.

4.6.6 Evaluation Using Substitution

A theorem can be generated from a lazy structure by means of a substitution rule. The rule can either be applied repeatedly to the whole term, or it can be used on subterms as with the congruence rules. It is not easy to determine which approach is most efficient. Using the rule on the whole term in general requires less applications and so less overheads associated with each application of a primitive inference rule. On the other hand, the terms to which the rule is being applied are larger, so each application involves more computation in that respect.

When the substitution rule is applied repeatedly to the whole term, the number of substitutions required is determined by the sequencing depth of the lazy structure. An application of the reflexivity rule may also be required to generate an initial theorem on which substitution can take place. Thus, the evaluation of the lazy structure in Figure 4.18 proceeds as follows:

$$\begin{aligned} &\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. (n * 0) + (n * 4) \\ &\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. 0 + (4 * n) \\ &\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. 4 * n \end{aligned}$$

The first theorem is generated by the reflexivity rule. The subsequent theorems are generated by substituting in the right-hand side. The substitution rule requires the following arguments:

- a list of substitution theorems associated with template variables;
- a template term;
- the theorem in which substitution is to occur.

For the first of the substitutions in the example, the three arguments are:

- $[((\vdash n * 0 = 0), v_1); ((\vdash n * 4 = 4 * n), v_2)]$
- $\lambda n. (n * 0) + (n * 4) = \lambda n. v_1 + v_2$
- $\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. (n * 0) + (n * 4)$

The first two arguments are generated from the lazy structure by traversing the structure in parallel with the right-hand side of the theorem in which substitution is to occur. (This is already known.) When a `Lazy_eq_thm` node is encountered a substitution theorem is obtained from it, and a new variable with a unique name is generated. The theorem and variable are added as a pair to the substitution list (first argument) and the variable is returned as the template term. When an empty sequence is encountered in the lazy structure, an empty substitution list is

returned together with the current subterm as the template term. At branch nodes in the lazy structure, the substitution lists generated by processing the subtrees are concatenated and the template terms are combined using the appropriate term constructor.

Since the substitution rule is expensive it is worth considering two special cases. First, if the substitution list is empty the theorem can be returned unchanged. Second, if there is only one substitution theorem and the right-hand side of the template term is simply the unique variable associated with that theorem, then the required result can be obtained by transitivity between the original theorem and the substitution theorem; the substitution rule does not have to be used.

The difficulty with computing the arguments for the substitution rule is in determining which of the `Lazy_eq_thm` nodes are to be used for this particular substitution. This boils down to choosing a path to follow when a `(lazy_eq)list` of more than one element is encountered. One technique is to explicitly generate the ‘flat’ lazy structures as illustrated by the three structures in Figure 4.20. However, this approach is likely to be inefficient because of the structure building and garbage collection required.

An alternative is to exploit sequencing depth information on-the-fly. This can be done if the sequencing depth of each substructure is known. The function used to obtain the arguments for the substitution can then navigate through the structure. For example, in generating the arguments for the second substitution above, the function knows that it is working on substitution 2. Since this number is greater than the sequencing depth of the substructure containing the equations $n * 0 = 0$ and $n * 4 = 4 * n$, the function knows that it must ignore this substructure and process the `Lazy_eq_thm (0 + (4 * n) = 4 * n)` substructure instead. While processing this substructure the substitution number must be relative and so the depths of any substructures skipped over are subtracted from the original value. (In the example, the original value 2 is reduced by 1.)

The above discussion assumes that the substitution rule can operate inside λ -abstractions. In HOL, this is not true of the primitive rule. However, a suitable rule can be defined, and could be made a primitive. The existing substitution primitive in HOL can be used in conjunction with the `ABS` rule. For the example in Figure 4.18, the following theorems can be generated by reflexivity and substitution:

$$\begin{aligned} &\vdash (n * 0) + (n * 4) = (n * 0) + (n * 4) \\ &\vdash (n * 0) + (n * 4) = 0 + (4 * n) \\ &\vdash (n * 0) + (n * 4) = 4 * n \end{aligned}$$

No abstractions are present in the terms, so the HOL primitive can be used. Then, the theorem required can be obtained by application of the `ABS` rule (the congruence rule for abstractions):

$$\vdash \lambda n. (n * 0) + (n * 4) = \lambda n. 4 * n$$

This example is somewhat misleading because the abstraction is at the root of the term. In general, there may be abstractions anywhere within the term. The

substitution can only be applied to the abstraction-free subterms (the bodies of abstractions). Each `Lazy_abs` node is then processed as if it were a `Lazy_eq_thm` node: The theorem generated by the application of `ABS` is placed in the substitution list with a unique variable. More substitutions are required since two or more `Lazy_eq_thm` nodes that are ‘in parallel’ cannot be processed by a single substitution if there are intervening abstractions.

The other difficulty with the HOL substitution primitive is that it may rename bound variables. This is rare because it only takes place when there is a variable free on the right-hand side of one of the substitution theorems that is not free on the left-hand side. However, when renaming does occur, it causes the evaluation of the structure to fail because of mismatches with terms in other parts of the structure. One approach is to abandon the use of substitution if any renaming occurs and use the congruence rules instead. This is a brute-force approach but it is reliable. Attempts to reverse the renaming have been partially successful but there are still rare occasions in which the evaluation fails. The real solution is to have a substitution rule that does not rename.

Experiments have shown that, even for a substitution primitive that can operate within abstractions, using substitution is not as fast as using congruence rules to evaluate lazy structures. The number of primitive inference rules used is reduced, but each substitution is considerably more expensive than an application of a congruence rule since substitutions traverse and rebuild entire terms as opposed to just a single node of a term. In addition, there are more overheads associated with the substitution approach; the arguments for the substitution and the sequencing depths have to be computed.

There is not a vast difference between the performances of the two approaches, and the exact figures are dependent on the particular implementation. For example, the congruence rules in HOL88 are included as primitives when it might be argued that they should be derived (since it is possible to do this from the real primitives of the logic). The algorithm used for substitution is also a factor, as is the implementation language. It might be the case that in HOL90 (see Section 6.1), which is written in Standard ML, the relative performances are reversed. The research required to answer this question is beyond the scope of this thesis.

4.6.7 Duplicated Lazy Conversions

The result of applying a lazy conversion is an ML entity that is not fully evaluated. Thus, as for lazy theorems (Section 3.3) there is the potential for repeated evaluation. This can be avoided by the same approach used for lazy theorems: placing the lazy structure inside a reference cell. However, the overhead of this is probably not worthwhile since the result of a conversion is normally used only once. It is either sequenced with the results of other conversions or it is transformed into a theorem before its value is duplicated. The structure of the result may be examined more than once but this information does not come from the unevaluated part of the result and so does not force evaluation.

If lazy theorems and lazy conversions are being mixed in the same system then

a lazy theorem rather than a theorem may be obtained from the result of a conversion. The call to evaluate the lazy structure (using the function `prove_lazy_eqs`) is enclosed in a function with a dummy argument in order to maximise the amount of computation that is delayed. This function is used as the justification function of the lazy theorem. In this case, repeated evaluation of the lazy structure is prevented because it is inside the reference cell of the lazy theorem.

4.7 Destructive Rewrite Rules

Section 2.4 discusses the issue of inferences that do not contribute to the final theorem because later operations make them unnecessary. In particular, the notion of a ‘destructive’ rewrite rule is introduced. A rewrite rule $l \longrightarrow r$ is *destructive* if there is a free variable in l that does not appear free in r . Any terms matched by such a variable are discarded by the rewrite. The following equation is an example of such a rule:

$$0 * y = 0$$

Discarded variables do not always give rise to unnecessary inferences. The inferences can only be safely eliminated if the discarded variable appears only once in the left-hand side. Otherwise the successful application of the rewrite rule may be dependent on the preceding inferences. The following equation illustrates this:

$$(x - x) * y = 0$$

Here, both x and y are discarded variables, but only manipulations of the subterm corresponding to y can be discarded. Consider the term:

$$((2 * 2) - (1 + 3)) * (50 + 10)$$

Conversions can be applied to subexpressions in order to simplify the term to:

$$(4 - 4) * 60$$

The equation can then be applied as a rewrite rule to yield zero. The inferences used to compute the sum of 50 and 10 could have been avoided, but those used to simplify $(2 * 2)$ and $(1 + 3)$ could not, because the variable x cannot match both of these expressions.

The criteria for safely eliminating inferences are still more complicated for conditional rewrite rules. In particular, a variable that appears free in the condition should not give rise to an elimination.

```

rectype lazy_eq = Lazy_eq_thm of term → thm
                | Lazy_comb of (lazy_eq)list × (lazy_eq)list
                | Lazy_abs of (lazy_eq)list
                | Lazy_kill

```

Figure 4.21: ML type for extended lazy structures

4.7.1 Extended Lazy Structures

Situations in which inferences can be eliminated because of destructive rewrite rules could be tested for explicitly within a proof procedure. However, the optimisation can be done transparently by extending the lazy structure introduced in Section 4.6. The ML type declaration for the extended structure is given in Figure 4.21.

A new constructor, `Lazy_kill`, is used to mark the position of a discarded variable. All preceding operations on the corresponding subterm are to be discarded. The functions for sequencing lazy structures (Figure 4.16) are modified so that they throw away the structures for those operations. The `Lazy_kill` itself is retained by the sequencing functions because a further sequencing may provide the opportunity to discard more structures. The other change to the data type is that the function used to generate a theorem in a `Lazy_eq_thm` node now takes a term as its argument instead of a dummy value. The reason for this is discussed in Section 4.7.2.

With this new form of lazy structure, the rewriting example in Section 4.3 gives rise to the structure in Figure 4.22. The `Lazy_kill` node appears before the node containing the theorem $\vdash 0 * n = 0$. These two nodes are introduced together by the conversion that rewrites with Equation 4.2 (Page 43). A good feature of the technique is that the structure containing the `Lazy_kill` node can be computed prior to the application of the rewrite rule because it does not depend on the term to be rewritten. (The annotation ‘ (n) ’ in the diagram is present only to assist the reader.) Hence the computation only has to be done once for each destructive rewrite rule. An empty structure is used for other rewrite rules.

The example illustrates a problem with the extended lazy structures, namely that they can be a deoptimisation rather than an optimisation. The example is not optimised because there are no operations on the subterm corresponding to n prior to the application of the destructive rewrite. It is deoptimised because the presence of the substructure containing the `Lazy_kill` node prevents the two `Lazy_eq_thm` nodes from being merged into one. Extra applications of the transitivity rule are therefore required (or if substitution is being used, more substitutions are required because the sequencing depth has increased).

For the sequencing function to be simple and efficient it is important for the structures to conform to certain restrictions. Most notably, a `Lazy_kill` should only ever appear as the first element in a `(lazy_eq)list`. This can readily be achieved within the basic conversion functions so that application programmers do not have to concern themselves with maintaining the structures in a normal form.

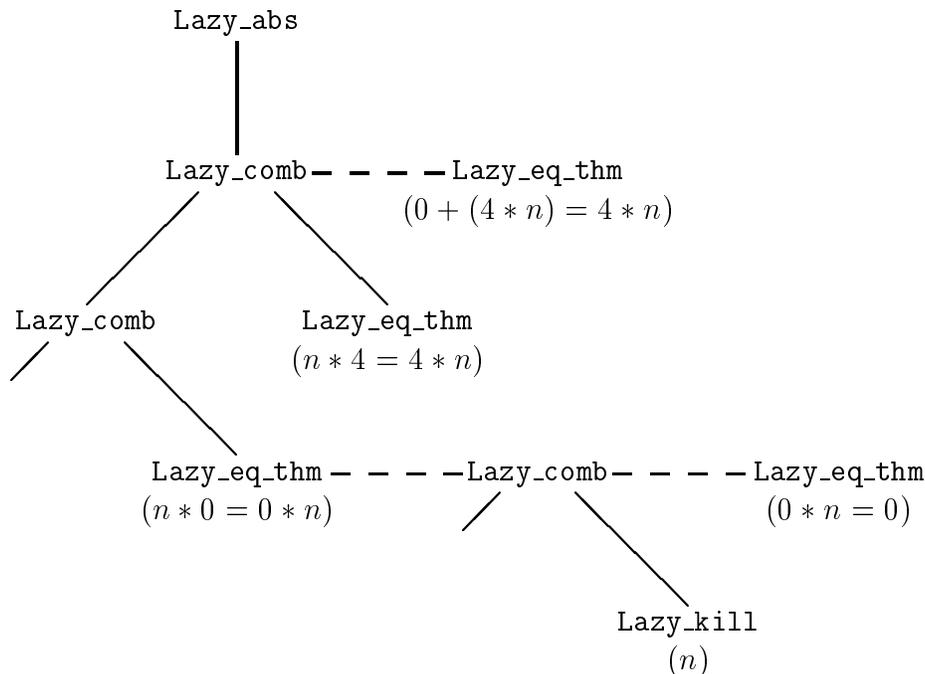


Figure 4.22: Extended lazy structure for the rewriting of " $\lambda n. (n * 0) + (n * 4)$ "

4.7.2 Dynamic Dependency

The introduction of discarding into lazy structures has a significant impact on the evaluation process. The theorems stored in the `Lazy_eq_thm` nodes become dependent on the discarding, and so cannot be fully evaluated when the nodes are created. This means that, not only must the evaluation be delayed (It is no longer an option associated with the use of delayed evaluation throughout the system.) but also that the term forming the left-hand side (and on which the right-hand side also depends) must be provided when the structure is evaluated and not before. In addition, any hypotheses introduced by the discarded operations will not appear in the theorem generated by evaluation of the lazy structure. Action must be taken to introduce these hypotheses so that the theorem has the same form as it would have done without the optimisation. Such hypotheses are logically redundant, but must be introduced if the optimisation is to be transparent.

The need for terms to be provided dynamically (i.e., during evaluation) can be seen from the simplification of the following expression:

$$0 * (50 + 10)$$

The conventional lazy structure for the simplification is given in Figure 4.23(a). Figure 4.23(b) illustrates the extended lazy structure. The `Lazy_eq_thm` node for the equation $50 + 10 = 60$ has been discarded because it was sequenced with a `Lazy_kill` node. The remaining `Lazy_eq_thm` node has been annotated with the

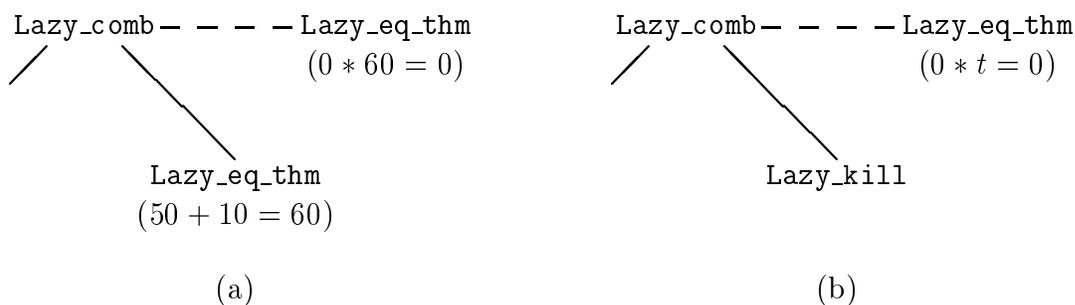


Figure 4.23: Lazy structures for the simplification of "0 * (50 + 10)"

```

case (leq1,leq2)
of (Lazy_eq_thm f1, Lazy_eq_thm f2) .
  [Lazy_eq_thm
   (λtm. let th = (f1 tm) in th TRANS (f2 (rhs (concl th))))]

```

Figure 4.24: Sequencing leaf nodes of extended lazy structures

equation $0 * t = 0$. The variable t has been used to illustrate the dynamic dependency. If the theorem for this equation is generated when the node is generated, then t will be 60. This will give rise to an error because an attempt will be made to apply the transitivity rule to the following equations:

$$\dots = 0 * (50 + 10) \quad 0 * 60 = 0$$

This arises because the simplification of $50 + 10$ has been discarded. In fact, t should be $50 + 10$, but there is no way to determine this when the `Lazy_eq_thm` node is generated because the function does not know what discarding will take place. The straightforward answer to this is to postpone the evaluation of the theorem until the structure is evaluated and then pass the term $0 * t$ (with t suitably instantiated) as an argument to the function stored in the `Lazy_eq_thm` node. The ML fragment in Figure 4.24 shows how two `Lazy_eq_thm` nodes can be sequenced under the new régime.

The other problem, of hypotheses not being introduced, is not common because it is unusual for hypotheses to appear in the equational theorems returned by conversions. Conditional rewrite rules are the most likely source of hypotheses. The following is an example of such a rule:

$$(x \neq 0) \Rightarrow (x/x = 1)$$

If this rule is used to simplify the expression $0 * (3/3)$ to $0 * 1$ and hence to 0, the theorem produced would have $3 \neq 0$ as a hypothesis:

$$3 \neq 0 \vdash 0 * (3/3) = 0$$

However, using extended lazy structures, the application of the conditional rewrite rule would be discarded, so the hypothesis would not be introduced.

The absence of hypotheses does not cause problems during equational reasoning since the operations of equational reasoning do not depend on the hypotheses in any crucial way; at most the rules form the union of the hypotheses of the theorems involved. The only exception is the congruence rule for abstractions. This fails if the bound variable of the abstraction appears free in the hypotheses. One might think, therefore, that the absence of certain hypotheses might allow the rule to succeed when it should not. In fact, this does not happen because the function `ABS_CONV` makes the test while the lazy structure is being generated, using the term structure which is generated at the same time. (See Figure 4.15.) The hypothesis list in the term structure has all the hypotheses present. Discarding only affects the lazy structure.

So, the absence of hypotheses only has to be rectified when the theorem is to be used in other kinds of reasoning (i.e., reasoning that does not use conversions). At this point the theorem generated from the lazy structure is compared to the associated term structure. If the two correspond, the theorem is returned. If the only difference is the absence of some of the hypotheses, then the missing ones are added. Otherwise, an exception is raised to indicate that a faulty proof procedure has been used. The hypotheses are added by discharging and undischarging. Consider the following theorem:

$$h_3, h_1 \vdash c$$

where h_3 , h_1 and c are metavariables representing certain terms. Now suppose this theorem (generated from a lazy structure) should have had hypotheses h_1 , h_2 and h_3 . Discharging these hypotheses in reverse order produces the theorem:

$$\vdash h_1 \Rightarrow h_2 \Rightarrow h_3 \Rightarrow c$$

(The HOL discharging rule introduces a term as the antecedent of an implication even when it does not appear in the hypotheses.) Undischarging three times then gives the required theorem:

$$h_1, h_2, h_3 \vdash c$$

This process does of course involve some additional inferences which is why the theorem is tested to see if it is correct before the process is applied.

4.7.3 Optimising Performance

The extra inferences performed due to the presence of `Lazy_kill` nodes in the final structure (Section 4.7.1) can be avoided by the use of exceptions when generating the theorem. An ‘unchanged’ exception is raised at `Lazy_kill` nodes and trapped at branch nodes in a similar way to the code in Section 4.4. However, in this case,

the use of exceptions is confined to the function that evaluates lazy structures, and so does not cause any difficulties to application programmers.

Experiments have shown that the use of exceptions in this way introduces extra costs irrespective of whether discarding is taking place. Since discarding is likely to be rare in practice it seems better just to accept the few additional primitive inferences when discarding does occur. These may be offset by the saving of inferences from the discarded operations.

Another alternative is to process the lazy structure just before generating a theorem so that the `Lazy_kill` nodes are eliminated. (They are no longer required at this stage.) A new lazy structure is generated. This requires extra computation and introduces more garbage, so it is questionable whether the saving in inferences would outweigh the overheads of this approach either.

More generally, it is not clear that destructive rewrite rules are sufficiently common to justify the extra infrastructure required to support the discarding optimisation. Section 5.3 discusses a realistic application in which discarding does take place.

4.8 Abstract Type for Equational Reasoning

Within this chapter, many different ways of implementing equational reasoning in HOL have been presented, but in each case the behaviour of the functions `ALL_CONV`, `NO_CONV`, `THENC`, `ORELSEC`, `RATOR_CONV`, `RAND_CONV`, and `ABS_CONV` has remained largely the same. This forms the basis for an abstract type of equations for which users have no access to the underlying representation. In this section, the complete set of functions required to permit flexible programming while retaining portability across the various optimisation techniques is presented. A number of additional functions that are commonly used and whose performance benefits from being defined directly are also presented.

4.8.1 Signature for Equations

The ML type `equation` was introduced in Section 4.5 as the result type of conversions. By making this type abstract, the implementation of conversions can be optimised in the ways already discussed, and in other ways not yet thought of, without the need to change any of the higher-level system code or users' code. The abstract type for equations is presented in Figure 4.25 as a Standard ML signature. The purpose of each function in the abstract type is specified below.

The functions `THM_OF_EQN` and `EQN_OF_THM` convert an equation to a theorem and vice versa. The former may involve proof. It cannot be implemented properly when exceptions are used as the optimisation technique. This is because an `equation` for an unchanged term is an exception rather than a concrete value. Similar remarks apply to conversions based on an ML data type when unchanged terms are not included explicitly. Thus, the use of `THM_OF_EQN` is to be avoided if one wants to allow these optimisation techniques.

```

signature Equation =
sig
  type equation

  val THM_OF_EQN      : equation → thm
  val EQN_OF_THM     : thm → equation

  val RULE_OF_CONV   : (term → equation) → (term → thm)
  val CONV_OF_RULE   : (term → thm) → (term → equation)

  val EQUATION       : (((term)list × term × term) × (term → thm)) →
                        equation
  val EQN_HYP        : equation → (term)list
  val EQN_LHS        : equation → term
  val EQN_RHS        : equation → term
  val EQN_PROOF      : equation → (term → thm)

  val SYM_EQN        : equation → equation

  val NO_CONV        : term → equation
  val ALL_CONV       : term → equation
  val THENC          : (term → equation) → (term → equation) →
                        (term → equation)
  val ORELSEC        : (term → equation) → (term → equation) →
                        (term → equation)
  val RAND_CONV      : (term → equation) → (term → equation)
  val RATOR_CONV     : (term → equation) → (term → equation)
  val ABS_CONV       : (term → equation) → (term → equation)

  val REWR_CONV      : thm → (term → equation)

  val COMB_CONV      : (term → equation) → (term → equation)
  val ARGS_CONV      : ((term → equation) list) → (term → equation)
end

```

Figure 4.25: SML-style signature for equations

The functions `RULE_OF_CONV` and `CONV_OF_RULE` perform similar tasks to those of `THM_OF_EQN` and `EQN_OF_THM`, but at the level of rules. The former produces a HOL rule from a conversion and is equivalent to the function `UCONV` described earlier in this chapter. The functions provide an interface between equational reasoning and more general reasoning. They must be added to existing ML code wherever there is a switch between these two kinds of reasoning. In the current versions of HOL with no optimisation of equational reasoning, the type `equation` corresponds to the type `thm` of theorems and the four interface functions are simply the identity function.

Switching between equational and other forms of reasoning is to be discouraged, not only because the interface functions clutter the code, but also because much of the optimisation of equational reasoning is lost when a switch takes place. It may, however, be possible to determine that optimisation is not possible at a particular point. For example, if it is known that some equational reasoning must change the term (or fail) then there can be no loss of optimisation for unchanged terms by leaving equational reasoning at that point. Similarly, if the next operation changes the kind of constructor appearing at the top level of the term, or is a rewrite on the whole term, then there can be no optimisation of repeated traversals. A beta-reduction can, for example, change the top-level constructor from an application to an abstraction:

$$(\lambda x y. x + y) 1 = \lambda y. 1 + y$$

The potential for other kinds of optimisation in a particular situation can be assessed in similar ways.

The function `EQUATION` is used when defining primitive conversions. A proof of the theorem is constructed using general HOL proof rules. This is left dependent on a term. A triple is also constructed consisting of the hypotheses, left-hand side, and right-hand side that are expected as the result of applying the proof. Thus the proof is a function which justifies the triple by generating a theorem with the same structure. This approach is taken so as to allow the inferences involved in the proof to be delayed. In this sense the type of equations is not as abstract as it could be. However, the underlying representation of equations is still hidden.

The functions `EQN_HYP`, `EQN_LHS`, `EQN_RHS`, and `EQN_PROOF` reverse the operation of `EQUATION` by extracting components. A function for each component is provided, rather than a single inverse function, in order to avoid building structures when extracting components. The internal structure of equations is not known, so it may be the case that returning a structure of type

```
((term)list × term × term) × (term → thm)
```

requires destruction and construction, while obtaining just one component only requires destruction. Thus, using separate functions leaves the issue of unnecessary garbage in the hands of the user.

It is often easier to perform certain kinds of equational reasoning by proving the symmetric equation to the one required and then reversing the left and right-hand sides. A simple example of this is the introduction of a double negation. Assuming

a procedure already exists to eliminate double negations, the easiest way to write a procedure to introduce double negations is to construct a double negation around the argument term and apply the elimination procedure to this new term. Thus, for a term t the following equation is obtained:

$$\neg\neg t = t$$

Reversing this using the symmetry rule produces the required equation. The problem with this is that the symmetry rule operates over values of type `thm` as opposed to values of type `equation`. So, a conversion using this technique has to switch between equational and general reasoning, thus limiting optimisation. The function `SYM_EQN` is included in the abstraction so that equations can be reversed directly. However, it is still not easy to allow optimisations to propagate. For example, if using lazy structures in the implementation they would have to be reversed. It is not clear how to do this in general, or that it is even possible. (Consider the unidirectionality of discarding; Section 4.7.) However, the most common optimisation to be propagated is the one for unchanged subterms. In this case, the lazy structure is an empty list, so it does not need to be reversed. In other cases it may be necessary to generate the theorem in order to reverse the equation.

The purposes of `NO_CONV`, `ALL_CONV`, `THENC`, `ORELSEC`, `RAND_CONV`, `RATOR_CONV`, and `ABS_CONV` were explained in Section 4.2. `REWR_CONV` is the primitive rewriting conversion. It takes a theorem (which must be an equation) and uses it as a rewrite rule. In most circumstances, `REWR_CONV` could be defined using `EQUATION`. However, when discarding is being used as an optimisation for destructive rewrite rules, `REWR_CONV` has to access the lazy structure directly so that it can introduce a `Lazy_kill` node. Hence, `REWR_CONV` has been included in the signature for equations.

`COMB_CONV` and `ARGS_CONV` apply conversions to certain subterms. They could be derived from other functions in the signature. The reason for their inclusion is discussed in Section 4.8.3.

4.8.2 Defining New Conversions

There are two ways to define a new conversion. The first is to construct a proof (a justification function) and apply `EQUATION` to it. This requires the result (right-hand side of the equation) to be computed by some other means since `EQUATION` also requires this information. The second approach is to build up the conversion from others. The first approach has the advantage that all the computation required for justification is delayed (assuming the implementation of equational reasoning allows this). The second has the advantage that non-local optimisations may be possible when the conversion is used as part of more extensive equational reasoning. However, in this case, some computation (such as the generation of lazy structures) will have to take place immediately so that the value of the right-hand side can be obtained.

The nature of the particular conversion being implemented determines which of the two approaches is better. If the equational reasoning implemented by the

conversion can be done metatheoretically, then the first approach is probably the one to use. This is because the metatheoretic algorithm can be used to rapidly compute the right-hand side. However, if the time to compute the right-hand side is a significant proportion of the time taken to evaluate the justification function, then it is better to build up the conversion from simpler ones. The right-hand side is then computed as part of the justification, which avoids duplicating work, and transparent optimisation may be possible too.

4.8.3 Complex Primitives

In Section 4.2 the function `COMB_CONV` was introduced. This applies a conversion to both the operator and the operand of a combination. As stated in that section, `COMB_CONV` can be defined in terms of other basic conversion functions:

```
let COMB_CONV conv = (RATOR_CONV conv) THENC (RAND_CONV conv);;
```

However, this definition may not be as efficient as defining `COMB_CONV` as a primitive. That is why `COMB_CONV` is included in the signature for equations. The inefficiency of the derived version manifests itself when both the operator and the operand are changed by the conversion. Consider the term " $f x$ " and suppose that the conversion `conv` transforms f to g and x to y . The application of `COMB_CONV` to this conversion and the term proceeds as follows:

1. $\vdash f = g$ by application of `conv` to the operator
2. $\vdash f x = g x$ by application of `AP_THM` to 1
3. $\vdash x = y$ by application of `conv` to the operand
4. $\vdash g x = g y$ by application of `AP_TERM` to 3
5. $\vdash f x = g y$ by transitivity between 2 and 4

Three inferences are required in addition to those involved in the application of `conv` to f and x . A primitive definition of `COMB_CONV` can exploit the combined congruence rule for applications (called `MK_COMB` in HOL) so that only one inference is required in addition to those used by `conv`.

When lazy structures are used to optimise conversions, the derived version of `COMB_CONV` is optimised transparently so that `MK_COMB` is applied instead of the three rules. So, in this case, there is no need to have `COMB_CONV` as a basic conversion function, but it is included for the benefit of other implementations. This also demonstrates the effectiveness of the optimisation using lazy structures.

The function `ARGS_CONV` is also included in the signature for equations. It applies conversions to the arguments of a compound application, e.g.:

```
ARGS_CONV [conv1; conv2; conv3] "f x y z"
```

applies `conv1` to x , `conv2` to y , and `conv3` to z . The definition of `ARGS_CONV` can exploit similar optimisations to those for `COMB_CONV`, which is why it is included in the signature. In principle, such minimisation of inferences is possible for more

Conversion type	Discarding	Evaluation by		Run	GC	Total	PInfs
Ordinary			E	84.0	8.7	92.7	16084
			T	73.6	11.0	84.6	12296
			I	61.7	8.8	70.5	0
Exceptions			E	84.2	8.7	92.9	15941
			T	73.8	10.9	84.7	12153
			I	62.2	8.7	70.9	0
ML data type			E	85.3	8.7	94.0	15941
			T	74.2	11.0	85.2	12153
			I	62.4	8.8	71.2	0
Lazy structures	No	congruence rules	E	83.6	9.1	92.7	15735
			T	73.4	12.6	86.0	11947
			I	53.3	8.3	61.6	0
		substitution	E	94.1	14.1	108.2	12877
			T	84.9	16.6	101.5	9090
			I	53.5	7.7	61.2	0
	Yes	congruence rules	E	82.6	9.0	91.6	14979
			T	72.9	10.3	83.2	10232
			I	55.1	8.2	63.3	0
		substitution	E	91.7	13.4	105.1	13290
			T	81.5	14.1	95.6	8544
			I	55.0	7.5	62.5	0

Table 4.1: Comparison of implementations of conversions

complex conversions. However, a compromise has to be made between performance and the convenience of implementing and maintaining the code. `COMB_CONV` and `ARGS_CONV` are sufficiently simple and of general use to warrant their inclusion in the abstract type for equations.

A function for applying a conversion to both arguments of a binary operator is often useful. Such a function can be defined using `ARGS_CONV` without any loss of efficiency (at least as far as number of inferences is concerned).

4.9 Results

Table 4.1 is a comparison of the various techniques for optimising equational reasoning described in this chapter. The results are for the multiplier benchmark running in Lazy HOL on a SparcStation ELC with 40 Mbytes of real memory. The table should be interpreted as for Table 3.1. Note also that the figures labelled ‘E’ are for tests done in eager mode, and ‘T’ and ‘I’ indicate total and initial computation figures, respectively, for lazy mode. The results in Table 3.1 for ordinary conversions in Lazy HOL are included to assist comparison.

When making comparisons the reader should bear in mind that only part of the benchmark is equational reasoning, and that the figures labelled as ‘Ordinary’ actually use exceptions to optimise depth conversions (and hence rewriting); it is only specialised conversions that are not optimised. It is also worth noting that the garbage collection times should not be trusted too far since the garbage collector may be invoked at different points in the execution on different runs.

The substitution rule used for evaluation of lazy structures in some of the tests is an ML implementation of a primitive rule which does not rename bound variables but does allow substitution inside λ -abstractions whenever it is valid to do so.

Chapter 5

Decision Procedures

The preceding chapters have presented a number of techniques for improving the efficiency of fully-expansive theorem provers. These techniques are illustrated in this chapter by describing their use in proof procedures. The main example presented is a decision procedure for a subset of linear arithmetic. First, however, some general issues concerned with choosing a suitable algorithm are discussed.

5.1 Choosing a Decision Procedure

One of the main areas of automated reasoning research is the search for algorithms which can compute a true/false value for formulas of decidable theories or subtheories. The goal is not simply to find an algorithm but to find the fastest one, so that automated reasoning systems can be more effective. Of course, the algorithm is not the only factor determining the speed of the system. Implementation language and coding techniques are also significant.

In most of the research to find faster decision procedures an assumption has been made that is not applicable in the realm of fully-expansive theorem proving, namely that only a true/false answer is required. The algorithm designer is free to employ any techniques that obtain that answer. In a fully-expansive system the algorithm must also provide a proof of the answer.

It is, of course, important that any algorithm to decide a formula does so correctly and so it is normal to find a correctness proof given in the literature whenever a new algorithm is presented. However, these proofs may not meet the standards of rigour required by a fully-expansive theorem prover. They often rely on arguments which though readily seen to be correct are difficult to formalise.

In a fully-expansive system there are three possible approaches:

- Trust the hand-written correctness proof by building the algorithm into the system as a primitive rule of inference;
- Formalise the data structures and algorithms in the theorem prover and use reflection or some similar technique;
- Have the algorithm generate a full proof for each input formula.

The first approach is by far the simplest but dramatically increases the complexity of the code in the trusted core of the system. This makes any verification of the core significantly more difficult. Similarly, the second approach requires that the procedure be verified. Only the last approach avoids this burden. However, it is at the cost of efficiency and a different algorithm may be required. Efficiency is lost because the correctness proof of the algorithm is repeated each time the procedure is called. A different algorithm may be required because the proof has to be in the object logic and so cannot exploit metatheoretic properties.

Reasons why it may be important to have a full proof for each formula have been given in Chapter 1, and since efficiently generating such proofs is the topic of this thesis the remainder of this discussion will focus on the last approach of the three. It turns out that a full proof can be generated with surprising efficiency provided a suitable algorithm is chosen. The criteria for choosing this algorithm are very different from the conventional ones used for decision procedures. To a first approximation they can be stated as, “Minimise the number of primitive inferences required.” Thus, simple algorithms may be favoured over ones which use sophisticated techniques. The reason for this is that the cost of checking all the conditions on the primitive inferences may far outweigh the cost of deciding the formula.

The other main point is that algorithms which use metatheoretic properties are ruled out unless the parts that use them can be replaced by object-level inferences.

Some of the criteria for selecting an algorithm for fully-expansive use are illustrated in the next section in which a decision procedure for a subset of linear arithmetic is presented.

5.2 A Decision Procedure for Linear Arithmetic

5.2.1 Linear Arithmetic

Linear arithmetic is the theory consisting of formulas made up from numeric constants, variables, addition, multiplication by a constant, the arithmetic relations ($<$, \leq , $=$, \geq , $>$) and the standard logical connectives (\neg , \wedge , \vee , \Rightarrow , \Leftrightarrow , \forall , \exists). More precisely, there are theories of linear real arithmetic, linear rational arithmetic, linear integer arithmetic, and linear natural arithmetic, according to the domain to which the constants and variables belong.

The theory of linear integer arithmetic was shown to be decidable by Presburger in 1929 and for this reason is often referred to as Presburger arithmetic. Decision procedures are known for full Presburger arithmetic (e.g., [Coo72]) and these are also applicable to the smaller subset consisting of only those formulas which when placed in prenex normal form are free of existential quantifiers. However, faster procedures are known for the smaller subset. The best are of considerably smaller time complexity than those for full Presburger arithmetic. Since many arithmetic formulas which arise in verification proofs fall into the smaller subset, there has been an emphasis on procedures for that subset. Perhaps the best known is the SUP-INF method due to Bledsoe [Ble75] and improved by Shostak [Sho77]. The SUP-INF method is really a procedure for purely-universal linear rational arithmetic. Results

in that domain are used to decide properties in the integer domain. This leads to some incompleteness of the procedure.

The problem of deciding linear arithmetic formulas can be reduced to a linear programming problem and so techniques from that field, such as the Simplex method, can be used as decision procedures. However, researchers into mechanical theorem proving have developed their own techniques, of which the SUP-INF method is one. Another technique which seems to be at the heart of a number of implementations is that of variable elimination. This technique can be traced back to Fourier (1824). Hodes [Hod71] describes a decision procedure for linear real arithmetic (with integer constants) based on the technique, and Boyer and Moore [BM88b] have used it in their procedure for linear natural arithmetic.

Bledsoe and Hines have applied variable elimination to a resolution-based theorem prover [BH80], and Käufel has implemented decision procedures in a tableau-based system [Käu88]. The former is for dense linear orders without endpoints (e.g., the reals). The latter handles both the rationals and the integers. It is an extension of the SUP-INF method designed to increase the coverage (completeness) when used on the integers. Multiple techniques are combined, with the cheaper ones being tried first. Then, only if these fail to decide the formula are the more expensive methods applied.

In addition to developing Bledsoe's SUP-INF method, Shostak explored another approach to deciding certain linear inequalities [Sho78]. He also developed procedures for other theories and combinations of theories. His code is still used today in the PVS proof checker [SOR93] from the Stanford Research Institute.

At roughly the same time as Shostak was developing decision procedures for use in program verification, Nelson and Oppen were doing the same. Nelson describes a satisfiability procedure for linear real arithmetic in his thesis [Nel80]. This is based on the Simplex algorithm.

It is actually possible to decide a larger subset of real arithmetic than just the linear formulas, namely the theory of elementary real arithmetic. Formulas of this theory are constructed using the standard logical connectives (where quantification is over the reals), the arithmetic relations, addition, subtraction, negation, multiplication, real variables, and *rational* constants. There has been much research into developing practical decision procedures for this theory within the algebraic manipulation community but the computational complexity of the problem is such that even the best algorithms today are only capable of dealing with fairly simple problems in a practical amount of machine time. Harrison [Har93] has recently been developing a fully-expansive procedure for the theory in HOL.

5.2.2 Selecting the Algorithm

The procedure described in this section is based on the variable elimination method adapted for natural numbers. It deals only with purely-universal prenex formulas. The procedure is not complete for properties which depend crucially on the integral nature of natural numbers. The absence of negative values also introduces some extra difficulties especially in the realm of fully-expansive theorem proving. The

initial normalisation stage of the procedure was inspired by Shostak’s description of normalisation for SUP-INF [Sho77].

A procedure for natural arithmetic was chosen because this was the domain for which there was the most pressing need for a decision procedure in HOL. Natural arithmetic is commonly used in verification proofs. In hardware verification, natural numbers are often used to represent time or steps in an execution. They are also used for parameterising generic hardware, e.g., n -bit adders.

The variable elimination technique was chosen because it offers a simple object-level algorithm which can readily be extended with applications of primitive inference rules to generate a proof (over time — an actual proof object is not returned). The algorithm operates over formulas rather than some less tangible representation. This makes it easier to see how to generate a full proof. The SUP-INF method appears to be much more difficult to extend and the resulting code would probably involve greater numbers of primitive inferences.

The SUP-INF method performs a computation in the domain of rational numbers in order to prove an integer formula. Expressions are maintained during execution and these may involve rational coefficients. The main part of the algorithm consists of two mutually-recursive functions, SUP and INF. To use the algorithm in a fully-expansive theorem prover would require one of the following approaches to be taken:

- Follow the algorithm closely with applications of inference rules;
- Formalise the data structures and functions in the object logic and execute the functions by rewriting.

The complexity of the algorithm and the presence of rational constants makes the former approach very difficult. The latter approach is made impractical by the large numbers of primitive inferences involved in the rewriting and also by the need to deduce a theorem for the formula over the integers (or naturals) from the theorem over the rationals¹. This deduction is straightforward at the metatheoretic level but not at the object level.

Whether it is possible to write a fully-expansive procedure based on the SUP-INF method or one of the other complex techniques is an open question. In any event, the implementation using variable elimination illustrates many of the optimisation techniques described in this thesis and these techniques are likely to be just as applicable to other procedures.

Finally, Shostak describes how the SUP-INF algorithm can be used to obtain counter-examples to a formula. Again, this process is not complete, but it does allow certain purely-existential formulas to be proved simply by instantiating them with the counter-examples. The use of SUP-INF in this way is described in more detail in Section 5.2.6.

¹Not all valid formulas over the integers or naturals are valid over the rationals, so there may be no theorem to deduce a final result from. This is the source of the incompleteness of the method.

5.2.3 The Algorithm

The procedure operates in two phases. The first phase normalises the negation of the formula using conversions. The second uses more general rules to simplify the normalised term, hopefully to false.

The first phase converts the formula of linear natural arithmetic into prenex normal form (all quantifiers are at the outermost level of the formula). An exception is raised if the resulting formula contains existential quantifiers. Otherwise, the result is a formula of the form:

$$\forall x_1 \dots x_n. f[x_1, \dots, x_n]$$

The procedure then attempts to prove that $\neg f[x_1, \dots, x_n]$ is false by normalising it. If this normalised formula can be shown to be false, the validity of the original formula is a straightforward consequence.

The normalisation of a formula proceeds as follows. The logical implications and Boolean equalities are replaced by conjunctions, disjunctions and negations. The negations are then pushed as far down the term as possible and the term is put into disjunctive normal form. The aim is to falsify this term, so each disjunct has to be shown to be false.

Each disjunct can be viewed as a set of inequalities. Each element of such a set is either a linear inequality or the negation of a linear inequality. The elements are normalised to \leq inequalities using rules such as:

$$\begin{aligned} (m < n) &= (1 + m \leq n) \\ (m = n) &= (m \leq n) \wedge (n \leq m) \\ \neg(m > n) &= (m \leq n) \end{aligned}$$

The resulting inequalities are further normalised so that each variable appears at most once in each inequality, and on the appropriate side of the relation for its coefficient to be positive.

The second phase of the procedure uses variable elimination to attempt to show that a set of inequalities (a conjunction) is false. If any of the inequalities contain only constants, they can be evaluated directly to either true or false. A false inequality completes the proof. A true inequality can be discarded. Variable elimination is then performed on the remaining set of inequalities². Consider the following example:

$$(m + n \leq p) \wedge (2m \leq 1 + n) \wedge (3 + p \leq 3m)$$

The procedure chooses a variable to eliminate with a view to minimising the number of inequalities in the resulting set. Suppose it decides to eliminate m . The lowest common multiple of all the coefficients of m is computed. Each inequality is then multiplied through by a constant so that the coefficient of m becomes that value:

$$(6m + 6n \leq 6p) \wedge (6m \leq 3 + 3n) \wedge (6 + 2p \leq 6m)$$

²It may be necessary to add an inequality for each variable asserting that it is non-negative, e.g., $0 \leq n$. Such inequalities are unconditionally true for the natural numbers.

All the possible pairs of inequalities such that one of the pair has m on the right-hand side and the other has m on the left are formed:

$$(6 + 2p \leq 6m) \wedge (6m + 6n \leq 6p)$$

$$(6 + 2p \leq 6m) \wedge (6m \leq 3 + 3n)$$

Addition of terms to both sides of selected inequalities then allows transitivity³ of \leq to be used to eliminate m :

$$(6 + 2p + 6n \leq 6p)$$

$$(6 + 2p \leq 3 + 3n)$$

These new inequalities are simplified and, together with those inequalities in the original set that did not involve m , are passed forward for further elimination:

$$(6 + 6n \leq 4p) \wedge (3 + 2p \leq 3n)$$

The original conjunction implies this one, but they are not equivalent logical statements. The variable n is eliminated in a similar way:

$$(6 + 6 + 4p \leq 4p)$$

On simplification, p disappears, resulting in an inequality between constants which can be evaluated to false:

$$(12 \leq 0)$$

The aim of this process of variable elimination is to prove that each set of inequalities is false. Unfortunately the procedure is incomplete because, in general, the steps of the elimination are implications, not equalities as in Hodges' method. This is because natural numbers are being used rather than rationals or reals. For example, the following is a theorem in the rational domain but not for natural numbers:

$$(\exists n. m \leq 2n \wedge 2n \leq p) = (m \leq p)$$

When m and p are both 1, there is a rational n that gives this equality, but no natural number n . In the natural domain only an implication holds:

$$(\exists n. m \leq 2n \wedge 2n \leq p) \Rightarrow (m \leq p)$$

It is not possible, therefore, to prove anything about the original set of inequalities unless false is derived at the end of the elimination. Deriving true provides no information.

³Note that to use transitivity the right-hand side of the first inequality must be identical to the left-hand side of the second. Some kind of normalisation such as sorting on the names of the variables must be used.

5.2.4 Implementation of Normalisation

A substantial part of the normalisation procedure is equational reasoning. The implementation makes heavy use of the conversion functions presented in Chapter 4, so the optimisations described in that chapter have a significant impact on the performance of the normalisation procedure. Nevertheless, additional measures have been taken to minimise the number of inferences. This section describes the implementation with particular emphasis on these measures.

Elimination of Implication and Boolean Equality

The first step in the normalisation is the replacement of implications and Boolean equalities using the following rules:

$$\begin{aligned}x \Rightarrow y &= \neg x \vee y \\x \Leftrightarrow y &= (\neg x \vee y) \wedge (\neg y \vee x)\end{aligned}$$

Observe that Boolean equalities are expanded in one step rather than first expanding to a conjunction of implications and then expanding the implications. This saves several costly inferences and is an example of using pre-proved lemmas (Section 2.5). The form of the rules has also been chosen so that the negations are as deep down in the term as they can be. This is to save inferences in the next stage of the normalisation in which negations are pushed down the term.

Pushing Down Negations

As already said, the second step in the normalisation is to push negations down the term using the following rules:

$$\begin{aligned}\neg\neg x &= x \\ \neg(x \wedge y) &= \neg x \vee \neg y \\ \neg(x \vee y) &= \neg x \wedge \neg y\end{aligned}$$

The procedure assumes that no other connectives are present. The rules are applied before processing the subterms (i.e., $\neg x$ and $\neg y$ for the second and third rules) so that the negations are swept down in one pass. The recursive calls for the binary connectives are made using the function `ARGS_CONV` (described in Section 4.8.3) so as to maximise efficiency.

When negations reach the level of arithmetic equalities and inequalities they stop. These negated and non-negated formulas must themselves be normalised. This could be done in another pass over the term but it is more efficient to do it while dealing with the negations (see Section 2.2). It is not, however, desirable to combine the processing of negations with the normalisation of these formulas in the same ML function. The solution is to parameterise the conversion for pushing down negations with another conversion which it applies when it reaches the equalities and inequalities. The type of the conversion is thus:

$$(\text{term} \rightarrow \text{equation}) \rightarrow (\text{term} \rightarrow \text{equation})$$

The two conversions are combined in the body of the main normalisation function using function application (as opposed to sequencing).

Reducing Equalities and Inequalities

So, within the second pass over the term, the equalities and inequalities are normalised. The first step in this process is to reduce each one to a formula involving the \leq relation. Examples of the reductions are given above in the description of the algorithm. Once again, the rules are designed to do as much manipulation as possible in one step. Thus, there are rules for the negation of each possible relation ($<$, \leq , $=$, \geq , $>$) as well as for the relations themselves. Note that the use of rules such as:

$$(m < n) = (1 + m \leq n)$$

is only valid because of the integral nature of natural numbers. Without such rules, the inequalities could not be reduced to formulas involving only the \leq relation.

When the equality relation is reduced, subterms are duplicated. It is therefore important to process each side of the equation before the rule is applied (Section 2.3). It would also be more efficient to maintain equalities in addition to the \leq (or $<$) relation because the resulting terms would be smaller and could be dealt with more quickly by the variable elimination procedure. However, maintaining equalities significantly complicates the elimination.

The rules for equality also introduce either a conjunction or a disjunction. For this reason, it is important to delay the stratification of the whole formula into disjunctive normal form until the equalities have been normalised.

After applying the rules to reduce equalities and inequalities, the new inequalities must be processed so that each variable appears only once in each inequality. The normalisation of the subterms prior to application of the rules only ensures that the variable appears at most once in *each side* of the inequality. Without this gathering, the elimination process would be very difficult to implement. Once again, the difficulties of fully-expansive theorem proving become apparent: Concrete terms have to be manipulated precisely; it is not sufficient to keep an aggregate record of the values of the coefficients.

Since normalisation procedures are applied both before and after the rules are applied, the conversion for processing equalities and inequalities is parameterised over two conversions. The two subprocedures are described below.

Normalising Sums and Products

Although multiplication is only permitted within linear arithmetic when at least one of the arguments is a constant, expressions may arise which are fundamentally linear but which require reduction to this state, e.g.:

$$(3 * x) * (4 + 2)$$

So, the first step in normalising each side of an equality or inequality is to expand them into a sum of products in which each ‘product’ is either a constant multiplied by a variable, or a constant or variable on its own. The multiplication of one constant by another is performed by a procedure that exploits memoisation (Section 2.6). Details of the expansion procedure are omitted since they do not raise any new efficiency issues.

Once the expression has been expanded into a sum of products it is made into a ‘linear’ sum in which the left-hand argument of every plus symbol is a product, a constant or a variable. This makes it easier to gather the subexpressions together so that each variable appears only once and there is only one constant term. The gathering function for variables assumes that coefficients are always the left-hand argument of the product. The general case uses the following rule:

$$(a * x) + (b * x) = (a + b) * x$$

This is followed by an evaluation of $(a + b)$. The function also has to handle the cases when either a or b or both are not present. At the same time as the gathering, the subexpressions are sorted lexicographically by variable name and the constant is moved to the left-most position in the sum. The key operation that allows these manipulations to take place ‘by proof’ is as follows. First an associativity rule is applied to make the top two summands accessible:

$$p + (q + r) = (p + q) + r$$

Then a manipulation such as reordering or gathering is applied to the sum $(p + q)$. If the result of this manipulation is itself a sum, then the expression is made linear again by applying the above rule in the opposite direction. Ideally, the applications of the rule should be avoided if the manipulation of $(p+q)$ turns out to be the identity function. This undoing of previous changes is difficult to detect automatically but, once noticed, is usually simple to implement as a special case of the code.

The operations involved in normalising the sums are generally difficult to both get right and do efficiently. It can be easy to take a brute-force approach, attacking the sum with a bunch of rewrites which have the desired normalising effect. The only concern is likely to be ensuring termination. To do the job efficiently, though, requires a delicate approach which may involve dealing with lots of special cases. For example, the rule above is not applicable when the sum has less than three subexpressions. As an added complication, the manipulation may alter the number of subexpressions. One positive point about writing the procedure in a fully-expansive theorem prover is the knowledge that any coding mistakes will manifest themselves as exceptions rather than as false theorems.

When performing similar operations for polynomials, Harrison [Har93] has taken the approach of defining a new logical constant to represent a polynomial generically. This allows more properties to be expressed by pre-proved theorems and hence reduces the run-time computation. For example, if a sum is represented using the constant `SUM`, then sorting of the sum could be captured by a theorem such as:

$$\vdash \forall l. \text{SUM}(l) = \text{SUM}(\text{SORT}(l))$$

where l is a list of summands. Note, however, that the function `SORT` would have to have a handle on the names of variables, which it does not have at the object level.

Weighed against the benefits is the need to fold the definition prior to the operations and unfold it again afterwards, i.e., prove the equivalence between the raw expression and its representation using the defined constant. Unless there is a big saving through using the pre-proved theorems, then on balance the technique may not be worthwhile. Extending the theory with new constants in order to implement the decision procedure might also be considered to be undesirable.

The final stage of normalisation for sums and products deals with coefficients that are either zero or one, and with a zero constant. If a variable has a zero coefficient it is eliminated from the sum. A variable appearing without a coefficient is transformed into a product, e.g., $(1 * x)$. Finally, if the constant term is zero it is eliminated. The choices for the exact form of the normalised expression are somewhat arbitrary. The important thing is for the results to be consistent so that procedures applied later know what to expect. It is difficult to determine which option will lead to the least number of primitive inferences since this depends on the form of the terms to which the procedure is applied. Explicitly introducing a unit coefficient may well be the less efficient choice because one expects the input terms to be free of such redundant coefficients, but it has the advantage of consistency: Other procedures can assume that all variables have a coefficient.

Normalising Inequalities

Once the arguments of arithmetic relations have been normalised and rules have been applied to reduce everything to \leq inequalities, a further normalisation is applied to the whole inequality. This arranges for each variable to appear on at most one side of the inequality. It also makes sure that a constant term appears on at most one side, except for the situation in which the whole of one side is zero and the other side features a constant term. An example of the normalisation is:

$$\begin{aligned} 1 + (3 * x) + (1 * z) &\leq 2 + (1 * x) + (4 * y) = \\ (2 * x) + (1 * z) &\leq 1 + (4 * y) \end{aligned}$$

Since the expressions range over the natural numbers, the reduction cannot be done using subtraction without incurring proof obligations that no negative values are generated. Instead, the amount by which each side of the inequality should be reduced is computed and this is used to generate a term for the required result. A term, t , representing the reduction is also generated. These tasks are performed without applying proof rules. Only terms are manipulated, not theorems. So, having cheaply computed the result, the following rule is used to generate a theorem stating that adding t to both sides of the inequality does not change the truth of it:

$$(p \leq q) = ((t + p) \leq (t + q))$$

The right-hand side of the resulting equation is then simplified by processing each side of the inequality using the procedure for normalising sums and products described above. The result should be the original term. The equation obtained is,

however, the wrong way round. This can easily be solved by applying the symmetry rule. Observe that this is a situation in which the function `SYM_EQN` described in Section 4.8.1 would be of use.

If a variable appears on only one side of the original inequality then it is not included in t . The same applies to the constant. Failure to observe this results in a zero being added to both sides of the inequality only to be eliminated again. This subtle point was missed in the first implementation. Once the code had been modified to treat zero as a special case there was a significant reduction in the number of primitive inferences performed with a corresponding reduction in computation time.

Disjunctive Normal Form

The final pass over the formula places it in disjunctive normal form. Some duplication of subformulas may take place as conjunctions are distributed over disjunctions. The formula is then ready to be processed by the variable elimination procedure.

Number of Passes

It is worth considering whether the whole process could be achieved in less than three passes over the term. In fact, the elimination of implications and Boolean equalities can be integrated with the pushing down of negations. As discussed below this has an additional advantage beyond avoiding an extra pass over the term.

The other possibility is to combine the pushing down of negations with the formation of disjunctive normal form. It may be possible for the latter to be done while backing-out after the negations have been pushed down. This has been avoided because of the complexity of the resulting procedure.

In any case, the repeated inferences due to making multiple passes will be eliminated transparently if the lazy conversions described in Section 4.6 are used.

The Problem of Duplication

Consider again the expansion of Boolean equality:

$$x \Leftrightarrow y = (\neg x \vee y) \wedge (\neg y \vee x)$$

Assuming there are no negations that have to be pushed down through this term, it will be passed to the disjunctive normal form procedure in this form, namely as a conjunction of disjunctions. The conjunction will, therefore, be distributed over the disjunctions to produce a term of the form:

$$(\neg x \wedge \neg y) \vee (\neg x \wedge x) \vee (y \wedge \neg y) \vee (y \wedge x)$$

The second and third disjuncts can be discarded but by this time x , y , $\neg x$, and $\neg y$ will have been normalised so that it is difficult for the procedure to see this. All four disjuncts will, therefore, be processed by variable elimination.

The problem could have been avoided by using a different formulation for the expansion of Boolean equality:

$$x \Leftrightarrow y = (x \wedge y) \vee (\neg x \wedge \neg y)$$

However, this approach is no more general than the original since if the Boolean equality happens to be negated the same inefficiency will arise. Instead, the solution is to combine the elimination with the pushing down of negations so that the most efficient formulation to use can be determined in every case. The combined procedure starts at the root of a term and pushes down any negations it encounters. In addition to the rules given above there are rules for processing implications and Boolean equalities:

$$\begin{aligned} (x \Rightarrow y) &= \neg x \vee y \\ \neg(x \Rightarrow y) &= x \wedge \neg y \\ (x \Leftrightarrow y) &= (x \wedge y) \vee (\neg x \wedge \neg y) \\ \neg(x \Leftrightarrow y) &= (\neg x \wedge y) \vee (x \wedge \neg y) \end{aligned}$$

Having applied one of these elimination rules, negations are pushed down in the subterms.

An interesting thing to note here is that even when the elimination has been combined with the pushing down of negations the efficiency is dependent on the later normalisation steps. For, if a conjunctive normal form is required then different elimination rules should be used. Also, x and y are still duplicated. However, one occurrence of each is negated and the other is not. Hence, they will be normalised in different ways and so have to be processed separately. Duplication of computation is still possible though. Consider, for example, the situation in which the x in the above rule for Boolean equalities itself contains a Boolean equality. The expansion would then be of the form:

$$((w \Leftrightarrow z) \Leftrightarrow y) = (((w \wedge z) \vee (\neg w \wedge \neg z)) \wedge y) \vee (((\neg w \wedge z) \vee (w \wedge \neg z)) \wedge \neg y)$$

Here there are two occurrences of each of w , z , $\neg w$, and $\neg z$, so computation will be repeated. Normalising the subterms prior to elimination of Boolean equalities would prevent repeated normalisation of w and z , but renormalisation would be necessary for $\neg w$ and $\neg z$. On the face of it this is not a problem because a normalised w (and z) is required anyway. However, normalising the negation of the normalised w may be more expensive than normalising the negation of the original w , and lead to bigger terms. A sophisticated procedure might be able to analyse the term to see which approach is better in each case, but this would probably be no more than a heuristic. The other possibility is to use memoisation (Section 2.6) to avoid repeating the computation.

Wider Applicability of the Techniques

Many of the considerations of efficiency presented in this section are also applicable to partially-expansive theorem provers. Indeed, some can be found in the literature,

e.g., the choice of whether to apply rewrite rules inside-out (bottom-up) or outside-in (top-down) is discussed in Chapter 9 of Bundy's book [Bun83] and is also raised in a fully-expansive context in Paulson's paper [Pau83]. However, the cost of failing to make these considerations may be much less significant in partially-expansive systems. There is a trade-off between performance and ease of programming with more weight being attached to the former in a fully-expansive theorem prover.

5.2.5 Implementation of Variable Elimination

There are three main components to the variable elimination procedure. First, functions are required to multiply inequalities by a constant and to add expressions to both sides. Second, these functions are used by a procedure which concentrates on the elimination. Finally, there is a conversion which given a conjunction of inequalities, passes it to the variable elimination procedure and processes the result into a theorem asserting that the conjunction is equal to false (assuming this is the case).

Preparing Inequalities for Elimination

Two operations are required to prepare inequalities for elimination: multiplying through by a constant, and adding an expression to both sides. The latter process has already been described in Section 5.2.4 where it is used to ensure that each variable and the constant appear at most once in an inequality.

Multiplying through by a constant involves a number of special cases but other than that is straightforward to implement. It is not valid to multiply an inequality by zero, so this case causes an exception. Multiplying by one is worth treating as a special case for efficiency. The other special cases arise because each side of the inequality may involve a constant term, variables with coefficients, or both.

The only other significant efficiency issue relates to the multiplication of constants. It has already been indicated that memoisation (Section 2.6) is used in the procedure for multiplying two natural number constants, but multiplying inequalities by a constant illustrates the potential for repetition. Consider the second elimination from the example in Section 5.2.3:

$$(6 + 6n \leq 4p) \wedge (3 + 2p \leq 3n)$$

The second inequality has to be multiplied throughout by 2. This involves the evaluation (by proof) of three products: $2 * 3$, $2 * 2$, and $2 * 3$. Clearly, the evaluation of $2 * 3$ is repeated, but it may also be the case that $2 * 2$ is present in the function's store when it comes to be processed. This is because $2 * 2$ is evaluated as a recursive call when evaluating $3 * 2$. So, by recording multiplication theorems, there is not only a gain for multiplications that have been done before, but also for new ones that make recursive calls.

The basic memoisation idea can be specialised for the multiplication procedure to improve performance further and to reduce the space required to store previously-proved theorems. There are two things that can be done. First, the commutativity

of multiplication can be used so that the smaller of the two constants is in the recursive argument position. For example, $2 * 4$ requires two recursive calls to the multiplication procedure whereas $4 * 2$ requires four. Commutativity is also exploited so that it is not necessary to remember both of the theorems $\vdash 2 * 4 = 8$ and $\vdash 4 * 2 = 8$. Only the former is maintained. Second, there is no point in maintaining theorems about zero and one, since the required multiplication theorems can be obtained by instantiating one of the following pre-proved theorems:

$$\begin{array}{ll} \vdash \forall n. 0 * n = 0 & \vdash \forall n. n * 0 = 0 \\ \vdash \forall n. 1 * n = n & \vdash \forall n. n * 1 = n \end{array}$$

Memoisation could also be used for addition of constants, but the overheads are likely to outweigh the benefits in this case. Also, it may be advantageous to maintain the theorems only for the duration of one call to the main arithmetic proof procedure since a different collection of constants is likely to be present in the next call. The stored theorems are therefore unlikely to be used, and if the store is allowed to grow too much the maintenance and searching overheads will become excessive.

Formulas involving constants of any significant size cannot be processed in a practical amount of time because of the unary representation used for natural numbers in HOL. It is not the fault of the procedure. It is a separate issue as to whether some other representation should be adopted or whether security should be compromised by trusting the machine's arithmetic. The issue is beyond the scope of this thesis.

Eliminating the Variables

In the example of elimination in Section 5.2.3 both of the initial pairings are used in the derivation of false. In general this is not the case. Some of the pairings may not contribute. It is, therefore, important for efficiency to separate the search for an elimination path from the generation of theorems. It may be necessary to construct a pairing that does not contribute in order to keep the options open for later eliminations, but it is not necessary to justify the pairing by primitive inferences. The lazy theorems presented in Chapter 3 provide a framework in which this can be achieved.

The lazy theorems used to implement the variable elimination procedure differ somewhat from those described in Chapter 3. Instead of using goals to represent the structure of the inequalities, a simpler representation is used. This is possible because the expressions under consideration are of a very limited form, being both restricted to the theory of linear arithmetic and heavily normalised. A normalised inequality can be represented using a list of variable names and coefficients. For example, the term " $1 + 3m \leq n$ " can be represented by the following ML data structure:

```
(-1, [( 'm' , -3); ( 'n' , 1)])
```

This abstraction discards all the unnecessary structural information. Computation on these coefficient lists is far faster than the equivalent computation on terms.

Properties such as associativity and commutativity become implicit. Hence, the variable elimination can be done even more efficiently. Hypotheses are not represented since they are not required for the search. Also, integers are used in the abstract representation even though the procedure is for natural numbers. Very few validity checks are made, making the process even faster. This increases the risk of the justification function failing to generate a theorem, but it is still not possible for an invalid theorem to be produced.

Generating the Theorem for each Conjunction

The final step in the elimination of variables from a conjunction of inequalities is to generate a theorem stating that the original conjunction is equal to false. Assuming that elimination has succeeded, the theorem obtained prior to this stage asserts an inequality between two constants. The theorem has some or all of the inequalities in the original conjunction as hypotheses. Only those inequalities used in the elimination are present as hypotheses. The rest can be introduced by discharging. The example in Section 5.2.3 gives rise to the following theorem:

$$(m + n \leq p), (2m \leq 1 + n), (3 + p \leq 3m) \vdash (12 \leq 0)$$

A conversion is then applied to the conclusion to evaluate the inequality:

$$(m + n \leq p), (2m \leq 1 + n), (3 + p \leq 3m) \vdash \mathbf{F}$$

The last inequality in the original conjunction is then discharged:

$$(m + n \leq p), (2m \leq 1 + n) \vdash (3 + p \leq 3m) \Rightarrow \mathbf{F}$$

Then for each of the remaining inequalities, the inequality is discharged and the following rule is applied to reconstruct the original conjunction in the antecedent of the conclusion:

$$x \Rightarrow (y \Rightarrow z) = (x \wedge y) \Rightarrow z$$

Thus, the sequence of theorems generated for the example is:

$$\begin{aligned} (m + n \leq p) &\vdash (2m \leq 1 + n) \Rightarrow ((3 + p \leq 3m) \Rightarrow \mathbf{F}) \\ (m + n \leq p) &\vdash ((2m \leq 1 + n) \wedge (3 + p \leq 3m)) \Rightarrow \mathbf{F} \\ \vdash (m + n \leq p) &\Rightarrow (((2m \leq 1 + n) \wedge (3 + p \leq 3m)) \Rightarrow \mathbf{F}) \\ \vdash ((m + n \leq p) \wedge &((2m \leq 1 + n) \wedge (3 + p \leq 3m))) \Rightarrow \mathbf{F} \end{aligned}$$

Finally, the following rule is applied to generate an equality:

$$x \Rightarrow \mathbf{F} = (x = \mathbf{F})$$

Observe that the procedure fails at this point if the evaluation of the post-elimination inequality did not yield false.

The above procedure for dealing with the hypotheses works correctly regardless of the order in which the hypotheses are introduced. This gives the elimination procedure more flexibility. Since a theorem is required at the end of the decision procedure, the hypotheses cannot be ignored; they have to be dealt with explicitly. The validity of the intermediate theorems is crucially dependent on the presence of the hypotheses.

5.2.6 Existentially-Quantified Formulas

Although the SUP-INF decision method is unsuitable for guiding the application of primitive inference rules directly, it can be used indirectly to assist in the proof of purely-existential linear arithmetic formulas. Shostak [Sho77] describes how to use the method to obtain counter-examples to purely-universal formulas. These can in turn be used as witnesses to satisfy the corresponding purely-existential formula. Thus, to prove the formula:

$$\exists x_1 \dots x_n. f[x_1, \dots, x_n]$$

the SUP-INF method is used to find a counter-example for the following formula:

$$\forall x_1 \dots x_n. \neg f[x_1, \dots, x_n]$$

The witnesses can be used to obtain a fully-expanded proof that the original formula is true; it is simply a matter of instantiating the quantifiers with them, followed by reduction to a constant. For example, consider the following existentially-quantified formula:

$$\exists m \ n \ p. \neg(m < n \wedge m < p \Rightarrow n < p)$$

The SUP-INF method can be used to obtain the witnesses: $m = 0$, $n = 1$, $p = 1$. Instantiating the formula with these values gives:

$$\neg(0 < 1 \wedge 0 < 1 \Rightarrow 1 < 1)$$

which reduces to true.

This is a situation in which an extra-logical procedure is being used to indirectly guide a full object-level proof by behaving as an oracle (Section 2.8). No theorem generation need be performed while searching for the witnesses.

Unfortunately Shostak's method only works for certain formulas in the theory, and whereas the incompleteness that arises in the procedure for universally-quantified formulas rarely shows itself in practice, many typical existential formulas cannot be proved by the method unless it is allowed to do an exhaustive search.

5.2.7 Results

There is no totally convincing way to test the performance of the techniques described in this thesis. Simple examples are informative in that they can be analysed in sufficient depth to see that less computation is being performed, but they may not give a good feeling for how much performance is gained in practice. Testing on real examples, though, raises questions about fairness: If comparing with previous implementations it is difficult to ensure that the new code is intended for the same purpose; it is often the case that a general-purpose program is slower than a more restricted one. On the other hand, writing a 'control' program (in the sense of a

System	Procedure	Run	GC	Total	PInfs
HOL88 Version 2.01 (*)	Camilleri	1713.9	231.0	1944.9	867475
HOL88 Version 2.01	Camilleri	520.4	133.3	653.7	113267
HOL88 Version 2.01	Boulton	45.3	5.3	50.6	11965
Lazy HOL (Version 2.01)	Lazy (Total)	48.6	5.1	53.7	11592
Lazy HOL (Version 2.01)	Lazy (Initial)	15.9	1.3	17.2	0

Table 5.1: Results for arithmetic decision procedures

control for a scientific experiment) with exactly the same functionality as the program under test is rightly open to the criticism that it may have been contrived to be inefficient.

The approach taken here has been to compare the implementation of the linear arithmetic decision procedure in HOL with an earlier program which has a similar specification. This is a significant example and is realistic in the sense that it is a program which meets real needs of HOL users. Even though the earlier program does not have identical functionality to the new one, it was the best thing available for automatically proving linear arithmetic formulas in HOL from the time it was written until the time at which the new procedure became available.

The old program is a procedure written by Albert Camilleri in 1985/86. It is an experimental prototype; it was not intended as a final implementation but as a means of generating data on the performance of such proof procedures, and thereby to contribute to research on how proof procedures should be implemented in fully-expansive theorem provers. It is in just this way that it is being used here.

Table 5.1 provides a comparison of the linear arithmetic decision procedures. The figures given are for a test file containing 54 linear arithmetic formulas. The examples range from very simple formulas to (more typically) formulas of a similar complexity to the ones given in Table 5.2. The columns of the table should be read as for those in Table 3.1. The figures were obtained on a SparcStation ELC with 40 Mbytes of real memory. All the versions of HOL were built using Austin Kyoto Common Lisp in which the storage parameters were modified to make better use of the available memory and hence reduce garbage collection.

Version 2.01 of HOL88 uses exceptions to optimise the functions that perform depth rewriting (Section 4.4). Since Camilleri's procedure makes extensive use of depth rewriting, figures are given for a version of HOL88 in which the optimisation has been removed (labelled with (*)) as well as for an unmodified version. As can be seen from Table 5.1, this simple modification introduces a large improvement in performance. An implementation incorporating the optimisations described in this section (third row of table) is an order of magnitude better still, in both number of primitive inferences performed and execution time.

The last two rows of the table give figures for the new implementation running in a Lazy HOL system. Optimisation of equational reasoning using exceptions is replaced by optimisation using lazy structures (Section 4.6). This eliminates a small

		Time	PInfs
$(n \leq p) \wedge (p \leq q) \wedge (q \leq r) \wedge (r \leq s) \wedge (s \leq n) \Rightarrow (n = q)$	T	1.4	261
	I	0.5	0
$(m = n) \wedge (n = p) \Rightarrow (m = p)$	T	0.9	208
	I	0.3	0
$(n \leq 4) \Rightarrow ((11 \leq 3 * n) \Rightarrow (n \geq 4))$	T	0.7	209
	I	0.1	0
$\neg(n \leq p + m) \Rightarrow ((m + p < n) \vee (m \leq 0))$	T	0.5	108
	I	0.2	0
$(m < n) \wedge (n \leq p) \Rightarrow (m < p)$	T	0.5	105
	I	0.1	0
$((m \leq n) = (n \leq m)) = (m = n)$	T	7.3	1343
	I	1.5	0
$(m < n) \wedge \neg(m + 1 = n) \Rightarrow (m + 1 < n)$	T	0.7	205
	I	0.3	0
$(m \leq p) \wedge (n \leq q) \Rightarrow (m + n) \leq (p + q)$	T	0.5	129
	I	0.2	0
$(m \leq n) = (2 * m) \leq (2 * n)$	T	1.0	295
	I	0.2	0
$n * 3 = n + (n + n)$	T	0.3	83
	I	0.1	0
$(m = 1) \wedge (n = 2) \wedge (p = 3 * n) \wedge (q = p + 2) \Rightarrow$ $((m + (5 * n) + (2 * p) + (3 * q)) = ((p * 6) + q + m + n))$	T	3.9	1074
	I	1.0	0

Table 5.2: Run times for individual examples in lazy arithmetic procedure

number of primitive inferences thanks to the ability to transparently optimise repeated passes over a term. The reduction is not very significant because great care is already taken in the arithmetic procedure to avoid repetition. The fact that there is any reduction demonstrates just how difficult it can be to avoid all inefficiency by manual analysis.

Despite the reduction in number of primitive inferences there is a slight increase in execution time due to the overheads of laziness. However, the real win from using the lazy system is that a result can be obtained in about one third of the time. The remaining computation is justification of primitive inferences and this can be postponed. Much of the delay comes from the variable elimination part of the procedure. This operates lazily even in the non-lazy version and hence makes considerable savings on unused primitive inferences. The bonus in a lazy system is that the computation involved in justifying those rules that are necessary can be postponed because the other parts of the procedure are also lazy.

Table 5.2 gives results for individual formulas using the lazy procedure. This should give some feeling for the practicality of the procedure. For each example there are two sets of figures. The figures labelled ‘T’ are for the total computation

time and those labelled ‘I’ are for the time taken to obtain a result. The difference between the two is the time taken to justify the result when the computation is forced at a later time. The column labelled ‘Time’ gives the total execution time in seconds, including garbage collection.

One feature particularly worth noting is the relatively large cost of proving the following formula:

$$((m \leq n) = (n \leq m)) = (m = n)$$

This is due to the nesting of the equality symbols. Each equality is normalised in such a way that the size of the term is approximately doubled. This increases the work to be done by normalisation and leads to a dramatic increase in the number of variable eliminations that have to be performed. Thus, the size of the original formula may not be a good guide to the execution time. The number of equality symbols present is probably a better guide.

5.3 A Symbolic Compiler

The HOL system has been used to formalise the semantics of a number of computer languages. This enables properties to be proved about specific language texts and in certain circumstances about the language itself. This also opens the way to verify compilers from one language to another. Both languages are formalised in HOL, as is the compilation program. The correctness proof involves showing that for all texts in the source language, the compiled text in the target language has the same abstract behaviour. Examples of this kind of activity can be found in the work of Joyce [Joy89], Camilleri [Cam91], and Curzon [Cur92].

It can be useful when developing the compiler to be able to execute it, i.e., apply the semantic rules to an example text to see if the correct result is obtained. This can work just as well when part of the text is left unspecified, i.e., a logical variable is used in place of a constant term. It is therefore known as *symbolic* execution. The process can involve large numbers of primitive inferences even for a small source text. Symbolic compilation by proof can thus be highly time-consuming. However, if the text of the language is represented as an abstract syntax tree in a programming language such as ML or Lisp then symbolic execution can be very much faster. There is no proof taking place when such an approach is used so it is not trustworthy, but if the ML or Lisp program is derived automatically from the semantic rules then it may be reliable enough to use for testing the compiler during development.

Symbolic compilation is not a decision procedure in the conventional sense, but it does have many similarities. In particular, it is an algorithmic rather than a heuristic process. It also demonstrates an extreme case of separating initial and justification stages of a proof. The lazy framework presented in Chapter 3 can be used to gain some of the performance benefits of symbolic compilation in ML, while retaining the ability to fully justify the result. This is done by having both an ML program and a proof procedure for the symbolic compilation. The former is used to perform the initial stage of computation and a call to the latter is used as the

justification function of the lazy theorem. Actually, the initial stage also has to include translation from the representation of the language text in the HOL logic to the representation in ML, and the reverse of this process following compilation.

The advantages of this extreme approach to laziness are that all of the justification of primitive inferences is postponed and only a small amount of memory is required to store the justification function. The justification function for the whole symbolic compilation is simply a call to the procedure that uses primitive inferences. The disadvantage is that the entire initial stage of computation is repeated in the justification stage since no information is shared between the two. This is practical for symbolic compilation because the ML version of the compiler takes only a small fraction of the time taken by a proper proof procedure.

The above approach has been tested using the work by Curzon to verify a compiler for a subset of the Vista structured assembly language. Vista was designed for use with the VIPER microprocessor, which itself has been partially verified. The basic instructions of Vista correspond to VIPER instructions, but the programmer does not have direct access to the program counter. Instead, the flow of the program is controlled by constructs similar to those found in high-level programming languages, e.g., `while` loops. It is in this sense that the language is ‘structured’. For more details about this work, see Curzon’s paper [Cur92].

Curzon has written a proof procedure for compiling Vista programs using his semantics in HOL, and together with Rajan has used Rajan’s tool [Raj92] to generate a compiler in ML. The results suggest that the ML procedure completes in about 6% of the time taken by the proof procedure. This includes the time taken to translate between the representations in logic and in ML. Thus, in Lazy HOL a result is obtained in 6% of the time it takes in HOL88.

The question arises as to the need to ever symbolically execute the compiler by proof. Why not be content with the version running in ML? Well, the user may wish to use the theorem generated by symbolic compilation in further proof. Such a proof may depend on the result being a theorem. The lazy framework enables the proof to continue without forcing the justification of the symbolic compilation process.

The proof procedure for compiling Vista programs is also of interest in another respect. It is an example of a procedure which makes heavy use of destructive rewrite rules (Section 4.7). The destructive rules arise because of unused arguments for environment information, etc. All the necessary information is passed around as arguments inside the compiler but any given language construct only requires some of it. For example, one of the destructive rewrite rules is:

```
TransCom SKIP rep st pst base cstbase = CODE ([], [])
```

The purpose of each of the arguments is not important. Simply observe that the rule is part of the translator of the syntactic category of commands, specifically the case in which the command is `SKIP`. No object code is generated in this case because the `SKIP` command does nothing. Thus, none of the other arguments are used and any inferences involved in manipulating them can be discarded. Thanks to

laziness, this optimisation⁴ is effective whether or not the arguments are ‘evaluated’ before the rule is applied, though if the rule is applied first even the initial stage of computation for the arguments is avoided. This might suggest that it is better to apply the rule before processing the arguments, but in general this is not the case, for if one of the arguments happens to be duplicated in the right-hand side of the rule, the evaluation of that argument will be repeated.

5.4 Polymorphic Lazy Theorems

There is no fundamental reason why the lazy theorems introduced in Chapter 3 should use goals (collections of terms) as the structure. Any representation that captures enough information for the proof procedures to know what to do is sufficient. This is illustrated by the abstract representation for inequalities used in the linear arithmetic decision procedure (Section 5.2.5).

The type of lazy theorems can be parameterised over the structure. In this scheme the original form of lazy theorem has the ML type `(goal)lazy_thm`. The most general type is `(*)lazy_thm` where `*` is a type variable.

In the original formulation (Section 3.3), the structure of the lazy theorem was placed inside the reference cell along with the justification function. For proved lazy theorems the structure was not kept separately, but when required was obtained from the theorem. ML does not allow polymorphism in reference values⁵, so the structure of a polymorphic lazy theorem must be maintained outside the reference cell. This leads to the new definition for the representation type of lazy theorems given in Figure 5.1. Compare this with Figure 3.2.

```
type lazy_thm_rep = Lazy_thm of void → thm
                  | Proved_thm of thm
```

Figure 5.1: Representation type for polymorphic lazy theorems

The reference cell containing the new representation type is paired with the structure (type `*`) to form the basis of the abstract type of polymorphic lazy theorems. So, for proved lazy theorems two structures are maintained: one outside the reference cell, and one inside it in the form of a real theorem. The function `prove_lazy_thm` must check that these are consistent before returning the theorem. The external structure is an additional overhead, but the results below suggest that it does not degrade performance to any great extent. In fact, having the structure available outside the reference cell may be beneficial.

The other major change required is the addition of an argument to the function `prove_lazy_thm`. Recall that this function converts a lazy theorem to a real theorem

⁴The reader is reminded that if the number of inferences discarded is small the technique may be a deoptimisation because of the overheads involved.

⁵Standard ML allows a limited form of polymorphic reference.

by (if necessary) applying the justification function. It also checks that the result corresponds to the structure that has been stored with the justification function. Since the structure may no longer be a goal, the comparison can only be made if it is first converted to a goal or the goal obtained from the theorem is converted to the type of the structure. Thus, `prove_lazy_thm` must be supplied with a function for converting either one way or the other, but which Γ

The advantage of converting the structure to a goal is that the comparison is then made on goals. This is the most rigorous comparison possible because goals contain all the structural information of a theorem. However, for just this reason, it is not always possible to convert the structure to a goal; it may not contain all the necessary information. For example, the hypotheses may be ignored by the abstract representation. So, the conversion has to be from goals to the abstract representation. The function for constructing a lazy theorem, `mk_lazy_thm`, does not require the extra argument because even in eager mode the correspondence check is now postponed until a real theorem is required.

There may be situations in which proof procedures using different abstract representations are to be combined. At the interface of two such procedures one representation must be converted to another. With the current set of functions in the abstract data type for lazy theorems this cannot be done efficiently. Hence, a new function, `restruct_lazy_thm`, is introduced that can apply a transformation to the structure without opening up the reference cell. The ML types for the full set of functions are given in Figure 5.2. (`**` is a type variable distinct from `*`.)

<code>mk_lazy_thm</code>	<code>:</code>	<code>(* × (void → thm)) → (*)lazy_thm</code>
<code>mk_proved_thm</code>	<code>:</code>	<code>(* × thm) → (*)lazy_thm</code>
<code>dest_lazy_thm</code>	<code>:</code>	<code>(*)lazy_thm → *</code>
<code>prove_lazy_thm</code>	<code>:</code>	<code>(goal → *) → (*)lazy_thm → thm</code>
<code>restruct_lazy_thm</code>	<code>:</code>	<code>(* → **) → (*)lazy_thm → (**)lazy_thm</code>

Figure 5.2: Functions for manipulating polymorphic lazy theorems

The original type of monomorphic lazy theorems (Figure 3.3) can be defined from the new type by instantiating `*` to `goal` and by using the identity function for the first argument of `prove_lazy_thm`. The only constraint on this argument is that it be of type `goal→goal`. However, to use anything other than the identity function for the type over which the lazy primitive rules are defined may be a breach of security. The reader is referred back to Section 3.4 to consider why this is the case.

The new version of `mk_proved_thm` requires that a structure be specified in addition to the proved theorem. This is easily dealt with by feeding it the goal structure of the theorem.

Defining monomorphic lazy theorems in this way does not appear to degrade performance significantly. Results for the benchmark are given in Table 5.3. This should be compared with Table 3.1. The only surprising difference is in the results for

	Run	GC	Total	PInfs
HOL88 Version 2.01	83.2	8.7	91.9	16054
Lazy HOL (in eager mode)	76.5	7.2	83.7	16084
Lazy HOL (total computation)	76.9	10.7	87.6	12296
Lazy HOL (initial computation)	63.3	8.4	71.7	0
Lazy HOL (total) / HOL88	92%	123%	95%	
Lazy HOL (initial) / HOL88	76%	97%	78%	

Table 5.3: Results for the multiplier benchmark with polymorphic lazy theorems

eager mode. The use of polymorphic lazy theorems reduces the execution time. This has been traced to the function `dest_lazy_thm`. In the polymorphic implementation the goal structure of proved theorems is maintained separately from the theorem and can be accessed directly. In the original implementation the theorem had to be destructed. It turns out that destructing a theorem is an expensive operation (at least in HOL88), and since in eager mode all lazy theorems are actually proved theorems, there is a significant saving. The fact that eager mode is now faster than lazy mode is probably due to the generation of both `Lazy_thm` and `Proved_thm` structures in lazy mode whereas only the `Proved_thm` structures are generated in eager mode.

In conclusion then, polymorphic lazy theorems allow efficient abstract representations, like the one described in Section 5.2.5, to be used in the same framework as general laziness. In all cases, the justification function operates over the full goal structure of a theorem, but the visible structure needs to contain only the information required to guide the proof procedures. As discussed in the next section, there also appears to be a way of combining decision procedures for disjoint theories without losing the benefits of abstraction.

5.5 Combining Decision Procedures

One might ask to what extent the optimisations proposed in Section 5.2 can be retained when integrating the decision procedure for arithmetic with other procedures. The experience of Boyer and Moore when integrating a similar procedure with a heuristic theorem prover [BM88b] suggests that it would be difficult to retain the optimisations. They claim that for the decision procedure to be used effectively by the heuristic prover there must be tight integration; the decision procedure cannot be maintained as a black box. On the other hand, Nelson and Oppen [NO79] have proposed a technique for combining decision procedures which only requires equalities to be passed between the constituent procedures; they remain as separate units. Thus, with this approach it should be possible to retain many of the optimisations, including the abstract representation for arithmetic inequalities, which provides much of the speed enhancement.

The Nelson-Oppen technique operates over conjunctions of literals from a com-

bination of disjoint theories. Two theories are disjoint if they have no constants in common. Since each literal may involve constants from several of the component theories it is not in general possible to apply a decision procedure for just one of those theories. Nelson and Oppen overcome this by replacing subterms with new variables. An equation is added to the conjunction for each new variable and the subterm it replaces. This process continues until every literal is a pure formula of one of the component theories. See their paper for details.

The important feature of the Nelson-Oppen technique is that in order to decide the processed formula the component procedures need only exchange equations, specifically equations between variables. Thus, to combine procedures operating over polymorphic lazy theorems (Section 5.4) all that should be required is functions to convert between the different representations of equations. A lazy theorem could then be passed from one procedure to another by applying one of these functions using `restruct_lazy_thm`. It should be stressed that this idea has not been tried out in practice, and getting the details right, such as handling hypotheses, is likely to be very difficult. However, it would be interesting, and useful, to implement a combined procedure in HOL for, say, linear arithmetic, lists, and uninterpreted function symbols.

Chapter 6

Other Theorem Provers

The techniques described in this thesis have concentrated on the HOL system. Their applicability to other fully-expansive theorem provers is likely to be of interest. This chapter describes some other systems and attempts to relate the techniques to them.

6.1 HOL in Standard ML

HOL90 [Sli91] is a reimplementation of the HOL system using Standard ML [MTH90]. From a theorem-proving point-of-view there is little difference between HOL90 and the version of HOL used for the research described in this thesis (HOL88 [GM93]). However, the use of Standard ML as the implementation language may affect some of the performance trade-offs and allows a number of the techniques to be coded more cleanly.

Part of HOL88 is written in Lisp, the rest in a version of ML. The ML code is compiled (often not very efficiently) to Lisp and from that to assembler. The commonly-used implementations of Standard ML are compiled more efficiently but vary in a number of respects, especially memory management. Since the lazy techniques in Chapter 3 exploit large memories, their effectiveness may vary considerably not only between HOL88 and HOL90 but also between versions of HOL90 built using different implementations of Standard ML. Also, where timing comparisons have been made between various techniques and the results were similar, the conclusion as to which technique is better may change.

Standard ML has a more sophisticated notion of exceptions than the one in HOL88. Exceptions are declared and can carry information with them rather than simply being strings. This allows the optimisation using exceptions described in Section 4.4 to be implemented more cleanly in Standard ML. A new exception can be declared explicitly for the purpose so conflicts with users' exceptions can be avoided.

Another advanced feature of Standard ML is its modules system. Data types, functions, etc., can be encapsulated in *structures*. Structures can be parameterised over other structures using *functors*. This facilitates the writing of generic code. Thus, the HOL system could be parameterised over a type of *pre-theorems* and associated primitive inference rules and destructor functions. Instantiating this to the

type of real theorems (which has to be retained in all implementations for security) would produce a system similar to the conventional HOL. Instantiating with lazy theorems would produce a Lazy HOL system. Similarly, the proposal for parameterising the system over equational reasoning (Section 4.8) could be implemented cleanly. Essentially, these are software-engineering issues.

6.2 Isabelle

Isabelle [Pau90] is a generic theorem prover written in Standard ML. It is generic in the sense that users can define their own logic using a metalogic. The syntax of the object logic is specified in the typed λ -calculus. New types are introduced for each syntactic category, together with constant symbols for the logical connectives, quantifiers, etc. These constants construct values of the new types. The primitive inference rules of the object logic are then introduced as axioms in the metalogic (intuitionistic higher-order logic). The metalevel implication expresses logical entailment, universal quantification expresses generality and bound-variable constraints, and metalevel equality is used for making object-level definitions.

Isabelle rules consist of a collection of premises and a conclusion. They have the same basic structure as Horn clauses in Prolog. Proofs are constructed by resolving the conclusion of a rule with one of the premises of another. Higher-order unification is used allowing function variables to be instantiated. The rules are explicit structures rather than metalanguage functions as in LCF and HOL. A single tactic is sufficient to apply all rules whereas in LCF a separate tactic is required for each rule. The states in a backward proof have the same form as rules. The initial state is a trivial rule with a single premise identical to the conclusion. By resolving a rule with the premise, a new state is obtained with the original conclusion and premises obtained from the rule. These premises can be thought of as the subgoals. The aim is to obtain a state in which there are no premises. Due to the unification, it is not exactly the original conclusion and premises that form the new state, but instantiated versions of them. Thus, it is possible to have metavariables in the original goal that become instantiated as the proof proceeds. This includes function-valued metavariables which can be used to represent some (initially) unknown term structure.

Higher-order unification is undecidable and there may be an infinite set of possible unifiers. The undecidability does not appear to arise in practice, and infinite lazy lists are used to capture all the possible resultant proof states.

So, Isabelle has a number of things in common with HOL: a programming metalanguage, tactics and tacticals, and a fully-expanded proof is generated (over time). There are also significant differences, e.g., rules are not implemented as ML functions but are represented as explicit structures. This has the consequence that derived rules can be applied just as efficiently as primitives. There is an analogy here with the ability in HOL to capture certain subproofs using pre-proved theorems (see Section 2.5). In Isabelle, the technique is more widely applicable.

It might seem, then, that the research presented in this thesis is not relevant to

Isabelle. However, applications of rules by resolution are combined by metalanguage operations such as tacticals. Various search strategies are implemented. Thus, proofs in Isabelle consist of large numbers of expensive inferences just as in HOL, even though the inferences may not all be primitive. It seems reasonable to suppose, therefore, that there is scope for optimisation.

In HOL, using term structures or more abstract representations to guide the path of a proof procedure can lead to performance enhancements because costly justification of theorems is avoided down blind alleys of a search. Perhaps an analogous technique could be used in Isabelle to avoid the cost of higher-order unification in the same circumstances. This is likely to be less general than the idea of lazy theorems in HOL because in many cases the unification will have to be applied simply to determine the information required to guide the proof.

Nipkow [Nip89] describes an implementation of rewriting for Isabelle that uses higher-order unification. The rules for reflexivity, symmetry, and transitivity have metavariables for the arguments of the equality relation. There is also a congruence rule for each operator. The rules are applied using the resolution tactic but the full power of higher-order unification is not required; first-order unification is sufficient. Specialised tactics are generated for each of the rules. A rewriting engine is obtained from these using tacticals structured in a very similar way to the depth conversions in HOL and LCF [Pau83]. It is conceivable, therefore, that optimisations along the lines of those described in Chapter 4 would be applicable.

Nipkow also describes an implementation that exploits the higher-order capabilities of Isabelle's resolution. In this, the congruence rules are represented generically by a single rule that uses a function-valued metavariable in place of the operator. The metavariable acts as a general context, allowing depth rewriting to be done with a single higher-order unification. However, like the use of substitution in HOL (Section 4.6.6), the complexity of this operation compared to applications of simple congruence rules might make it just as costly.

6.3 LAMBDA

LAMBDA [AHL91] is a commercial system similar to Isabelle but specialised to a classical higher-order logic with polymorphism like the one mechanised in the HOL system. Early versions implemented a constructive logic of partial terms. The system comes with a graphical interface for hardware design called DIALOG. Higher-order resolution is used for proof construction, as in Isabelle. Thus, the remarks concerning the potential for optimisation of Isabelle are also applicable to LAMBDA.

6.4 Veritas

VERITAS [HDH92] is a logic intended to support formal design methods. It is motivated by Martin-Löf's intuitionistic (constructive) type theory [Mar85]. It is higher-order and provides dependent types and subtypes, but it differs in that it is

a classical rather than a constructive logic. The VERITAS-90 version is implemented in both Standard ML and Haskell. As stated in Section 1.5.3, the intention behind the implementation in the purely-functional language Haskell is to enable parallel evaluation. This may provide optimisation at two levels. At a low level, different regions of the search space of a tactic are evaluated in parallel. At a higher level, several tactics could be tried concurrently. Using a lazy language may also optimise when intermediate results are not used in the final result.

As in LCF, abstract data types are used to securely implement the various syntactic categories of the logic. In addition to terms and theorems being implemented in this way, so is the representation of theories, which are first-class objects in VERITAS. In LCF and HOL, theories are structures that are modified imperatively. In VERITAS, a purely-functional view is taken. The lack of imperative features causes some problems for input/output and for implementing tactics that conventionally use exceptions. These difficulties have been overcome. Exceptions, for example, are simulated using a disjoint sum type. The data type described in Section 4.5 performs a similar rôle but with somewhat different motivation.

The VERITAS logic is considerably more complex than HOL's. The terms, etc., have many more constructors, and there are more primitive inference rules. Thus, techniques such as those for optimising equational reasoning are likely to be significantly more difficult to implement. For example, the simplicity of the HOL logic in having only two kinds of internal node in terms is a big advantage for lazy structures (Section 4.6.1).

Dependent types enable particularly natural specifications to be written, but there are prices to pay. Type inference becomes undecidable, so the user must provide terms with type information. However, much of the burden of this can be relieved by heuristics in the parser. Another problem is that a typed subterm cannot always be extracted from a typed term because the latter is constructed from an untyped term and a type. This can be solved by annotating the untyped term. Goal-directed (backward) proof is particularly advantageous for a dependently-typed logic because the numerous simple subgoals that arise can be dealt with automatically. The VERITAS implementations feature a cache of trivial results to assist tactics in this process.

6.5 Nuprl

Like VERITAS, the logic implemented by Nuprl [C⁺86] is based on Martin-Löf's type theory [Mar85], which it follows more closely than VERITAS in that it is constructive. Once again, the complexity of the logic is likely to make optimisation more difficult than in HOL. The infrastructure of Nuprl has much in common with LCF, but differs in that proofs are represented explicitly as objects. However, these are not full proofs in terms of primitive inferences; each node of the proof tree corresponds to the application of a tactic which might be at a high level. Nevertheless, this helps users to examine the proof, and facilitates backtracking and alternative proof attempts. Jackson's tutorial paper [Jac92] includes a description of a recent version

of the Nuprl system. The discussion below is based on this.

The expressiveness of Nuprl’s logic enables lemmas (rather than tactics) to be used for summarising patterns of inference, but technical difficulties prevent this efficiency technique from being universally applicable. On a similar subject, proofs of the well-formedness of terms (which are ubiquitous in Nuprl’s logic) are shared by using directed acyclic graphs rather than trees for the proof structures.

There are over one hundred ‘primitive’ rules of inference in Nuprl. This amounts to a significant quantity of code to be verified to ensure the soundness of the system. It is therefore questionable whether Nuprl should be considered to be a fully-expansive theorem prover. It also makes use of decision procedures for equality and integer arithmetic that are hard-coded in Lisp, which further reduces confidence in the system. The arithmetic procedure is not for full linear arithmetic, but a verified implementation of the SUP-INF method is planned. The verification would use a model of the Nuprl logic within the logic itself. The intention is not just to verify the linear arithmetic procedure but also other inference procedures. These would be invoked using a reflection rule.

Nuprl provides two modes of operation for some tactics: one fast and unsafe, the other slow and safe. These are similar to the draft and eager modes described in Section 3.7. The idea is for users to develop proofs interactively in the fast mode, and then securely check the proof by replaying it overnight in the slow mode. One difficulty with this is that the two versions of the tactics must be kept consistent. A framework similar to the lazy one described in Chapter 3 might go some way to alleviating the difficulty.

Rewriting in Nuprl uses conversions. The rewriting package can be used with user-defined equivalence and order relations. This includes logical implication when it is considered as an order relation on propositions. The rewriting steps require a justification proof just as in HOL. The computation of the effect of a rewrite is separate from the justification and usually much faster, so the rewriting tactics are particularly suitable for use with the fast and slow modes of operation. Nevertheless, the optimisation techniques described in Chapter 4 of this thesis would seem applicable and would offer improved execution times in the slow mode.

6.6 Nqthm

Nqthm is a theorem proving system for a computational logic [BM88a]. The logic is (intentionally) similar to the core of the Lisp programming language. It is untyped, quantifier-free, and first-order. Nqthm provides a high degree of automation in terms of heuristics for simplification, induction, and related activities. A proof in the system is generated by making definitions and then calling the heuristics on the conjecture to be proved. The proof attempt will typically fail, but examination of the prover’s log (which is in English) will reveal either a flaw in the conjecture or a lemma that the prover needs to be aware of to succeed. The user then asks the system to prove this lemma (possibly requiring other lemmas to be proved) and repeats the attempt to prove the original conjecture. This process continues until

the prover succeeds on the conjecture.

Nqthm is not a fully-expansive theorem prover in the sense used in this thesis. It has no notion of a primitive inference rule. However, many of the heuristics and procedures described in Boyer and Moore's first book [BM79] have been implemented in HOL [Bou92]. The relative ease with which this was done suggests that the operations are mostly at the object level. Where metalogical properties are exploited, a corresponding object-level proof can be generated.

The implementation of the Boyer-Moore procedures in HOL was not done with efficiency as a high priority. The main purpose of the exercise was to determine the feasibility of Boyer-Moore automation in HOL, so getting the implementation right was the primary objective. Nevertheless the number of primitive inferences used in proving some of Boyer and Moore's more simple examples is not huge. This may be related to the fact that much of the computation in the Boyer-Moore procedures involves deciding how to do the proof rather than actually doing it. Even so, some of this work in the HOL implementation involved the application of primitive inference rules when it need not have done (for ease of coding). For example, back-chaining with conditional rewrite rules is done with primitive inference rules but it is not always successful.

It seems reasonable to conclude, therefore, that the system described in Boyer and Moore's book could be made fully-expansive. However, the Nqthm prover is more sophisticated. In particular, it incorporates a decision procedure for linear arithmetic. This alone changes the whole nature of the task. Nevertheless, the fact that the use of clever heuristics finds short proofs is to be admired, not just for the automation, but also for the efficiency.

Chapter 7

Conclusions

Theorem provers that fully expand proofs into primitive inferences are necessarily slower than those that do not. It has been claimed that fully-expansive theorem provers are orders of magnitude less effective [Rus90]. This may be true. However, a more important question is whether fully-expansive systems are practical. As hardware becomes faster, so do both fully-expansive and partially-expansive theorem provers. Partially-expansive systems will always have a significant edge, but the research described in this thesis suggests that there is much that can be done to reduce the gap. There is no real evidence to suggest that fully-expansive theorem provers are more than a constant factor less efficient than comparable partially-expansive systems.

The above is hardly a convincing argument for the use of fully-expansive systems. However, they have two substantial benefits that should also be taken into consideration. First, it is easier to trust or verify a fully-expansive system because the soundness depends on only a small number of simple procedures. The inclusion in the critical code of a proof procedure for arithmetic, for example, dramatically increases its complexity.

Just as significant is the flexibility of the fully-expansive approach. Since all proof ultimately takes place inside the primitive rules, users can be given the freedom to write their own procedures or develop the system as they wish without the risk of making the system unsound. The fact that users want to do this is demonstrated by the popularity of the HOL system as a tool for supporting and reasoning about hardware description languages [BGG⁺92], programming languages [Age92, Gru92, Sym93], and specialised logics and formalisms [vW91, Nes92]. Partially-expansive theorem provers do not readily provide this flexibility to third-party developers, yet writing an entirely new theorem proving system to support each new computer language, though possible, is a waste of resources.

How, then, can greater efficiency be achieved in the HOL system? Verification of directly-implemented proof procedures is one route to high efficiency without loss of trust. A disadvantage of this approach is that a fully-expanded proof is no longer obtained; some applications require a proof in terms of primitive inferences. Another disadvantage is the difficulty of the verification task. Thus, it is unlikely that all proof procedures will be verified. Users are likely to want to develop their

own high-level procedures without having to prove their correctness.

This thesis identifies some of the causes of inefficiency in the HOL system and presents some solutions. A framework is described for improving the productivity of users. This can also provide optimisation in certain circumstances. Various techniques are described for transparently optimising equational reasoning. These techniques and some more specific optimisations are illustrated by an implementation of a decision procedure for a subset of the theory of linear natural arithmetic. This is almost certainly the first efficient implementation of a linear arithmetic decision procedure that works entirely by application of primitive inference rules. The other techniques described in the thesis have also been implemented.

The next section summarises the research. This is followed by conclusions as to which techniques are worth implementing and under what circumstances. There then follows a discussion of laziness, and finally prospects for further work are considered.

7.1 Summary of Research

The slowness of proof procedures in the HOL system compared to many other theorem provers can be attributed in part to the need to write them using the primitive inference rules. However, there is also a degree of inefficiency due to a poor choice of data structures and algorithms. The programmer is constrained to some extent by the need to use primitive inferences and some algorithms that are perfectly reasonable in other contexts can be quite inefficient under these constraints. These inefficiencies may not be at all obvious at first sight. One of the aims of this research has been to identify and highlight them. Techniques are proposed for overcoming the inefficiencies, and wherever possible to do so in a manner that is transparent to both the programmer and the user.

In some cases, the overheads introduced by more sophisticated techniques outweigh the benefits of reducing the number of primitive inferences. However, this reduction can be beneficial in other ways, e.g., the use of a proof checker becomes more feasible because proof scripts are smaller and the checking process is faster.

7.1.1 Sources of Inefficiency and Some Solutions

Inefficiency in HOL is due in part to unnecessary application of primitive inference rules. Some applications may not be required to generate the final theorem. Others may be repeated when they could have been executed just once. Procedures that involve search for a proof are particularly likely to apply inferences unnecessarily. The most natural way to write these procedures is by applying inference rules to manipulate the logical terms. However, this is much more expensive than manipulating terms directly and so is wasteful down the unsuccessful branches of the search.

Search procedures are high-level so the inefficiency is fairly easy to detect and avoid. Similar inefficiencies arise at lower levels. Perhaps the best and most costly example of this is the rewriting strategy used by HOL for many years. Amongst

other things it reconstructed whole subterms that were untouched by the rewrite rules. This is wasteful when the terms are being constructed directly but when the operations are being justified formally it is highly inefficient. Typically, the rewriting takes three times longer than it need do. Other inefficiencies arise in equational reasoning due to repeated term traversals, duplication of subterms and discarding of subterms. Some repeated term traversals can be almost impossible for the programmer to detect because they arise when two or more programs are put together. However, by delaying part of the computation, much of the inefficiency can be detected and avoided by the basic procedures from which equational reasoning programs are constructed. Duplication and discarding of subterms can give rise to repeated and unused inferences, respectively. Some progress has been made towards eliminating this inefficiency transparently.

Repeated inferences commonly arise due to the repeated proof of lemmas. It is well-known amongst HOL users that pre-proving and storing such lemmas can optimise procedures that involve them. The lemmas are not always easy to detect. For example, a different lemma may be required for each call of a procedure but within each call the lemma is used several times. The well-known technique of memoisation may be an applicable optimisation in such circumstances.

The primary concern of HOL users is rarely theorem proving per se. Formal verification and similar activities are usually the primary objective. Thus, they do not always have a lot of time to spend on developing proof procedures. There is a tendency, therefore, to attack problems with brute-force procedures, especially rewriting. The techniques described in this thesis go a long way to optimising such procedures, but there is even more efficiency to be gained from careful programming. However, this is both difficult and time-consuming. There is scope here for further research in terms, say, of a program for automatically generating efficient proof procedures from high-level specifications.

Two additional techniques that are sometimes applicable are the exploitation of domain-specific information and metatheoretic guidance. The former can be applicable when the terms have a special form such as corresponding to the abstract syntax tree of an embedded language. The latter uses an external program (in the sense that it does not apply inferences) to compute some data that enables a shorter proof, e.g., a witness for an existential quantifier.

7.1.2 Using Delay

Delaying computation is one of the major techniques in this thesis. The idea is for proof procedures to manipulate terms directly to obtain the structure of the theorem, while retaining the ability to justify it by primitive inferences. The justification is postponed. This is a hybrid approach, combining the speed of partially-expansive systems for the initial computation, with the security of ultimately performing justification by primitive inference. Metatheoretic reasoning may be used in the initial stage of the computation. It is possible to delay computation in ad hoc ways but this technique provides a general framework in which to do it. The delayed computation from subprocedures can therefore build up into a large delay enabling users to make

better use of their time. Any delayed computation that does not contribute to the final result will never take place. This facilitates the writing of search procedures that do not perform unnecessary justification. Delay of justification also enables some non-local optimisations to be made.

The basic idea of the framework for delay is the lazy theorem. A lazy theorem consists of the structure of a theorem together with a function for justifying it. The function takes a dummy argument. The only purpose of the argument is to delay evaluation of the justification code. This is the standard technique for simulating lazy evaluation in an eager language [Pau91, §5.12]. The justification function is placed in an updatable cell so that when it has been evaluated once all other references to it access the evaluated result instead of repeating the evaluation. This is necessary because a lazy theorem may be used more than once.

Having established the data type of lazy theorems, lazy versions of the primitive inference rules are defined. For efficiency, these have to be implemented directly. If they were to be defined in terms of the ordinary primitive rules, the validity checks would be performed in both the initial computation and in the justification function when this calls the ordinary primitives. Because they are implemented directly, the lazy primitive rules are part of the critical code. It also means that any lazy theorems passed as arguments to the primitives must be proved by the justification function before it asserts the result to be a theorem. Otherwise, it is possible for an invalid formula to become a theorem. All the validity checks have to be performed in the initial stage of computation so that if the lazy rule is going to fail it will do so immediately rather than when the justification function is applied. Hence there is very little delay for the primitive inferences.

Lazy derived rules can be defined in terms of primitives just as the non-lazy versions are. Thus, the code for the derived rules does not need to be changed when moving to a lazy framework. However, more delay can be introduced by placing the applications of the primitive rules inside the justification function of the result. The initial computation may then consist of only the essential validity checks. The drawback of introducing this delay is that the initial computation is extra.

The lazy framework is to some extent a time/space trade-off. Delay is achieved at the expense of the extra memory required to store the justification functions. With this in mind, three modes of operation are proposed. In addition to the lazy mode, there is an eager mode which evaluates the justification functions immediately. This prevents delay but has the advantage that it can be used when memory is becoming exhausted. There is also a draft mode which discards the justification function altogether. This is also cheap in terms of memory but a real theorem can never be generated. This mode may be useful for initial proof development. In draft mode the system is operating much like a partially-expansive theorem prover though its operation is less trustworthy because users' code may be involved.

7.1.3 Optimising Equational Reasoning

Equational reasoning forms a substantial part of many proofs in the HOL system. This is usually in the form of rewriting, but more specialised procedures are also used.

It is of some consequence then that equational reasoning is particularly amenable to optimisation. Equational reasoning in HOL is performed by functions known as conversions. These take a term and return a theorem stating its equivalence to some other term. Most conversions are generated from some basic conversions using functions for sequencing, alternating, and repeating. One of the most important basic conversions is the one that applies a rewrite rule. There are also functions for applying a conversion at subterms of a term. From these and the other functions, rewriting engines can be written with various term traversal strategies.

This modular approach to building proof procedures for equational reasoning has the advantage that the procedures are clear and easy to write. A possible disadvantage is that the modularity may hide and encourage inefficiency. Inefficiencies such as reconstructing unchanged subterms and repeated traversal have already been mentioned in this summary. Interestingly, it is also the modularity that offers a neat solution to the inefficiency. The basic conversions can be implemented in a number of ways that optimise the performance without requiring changes to the derived conversions. These techniques include the use of ML exceptions and the use of ML data types. The latter requires that conversions return the data type as their result rather than a theorem. This forces the introduction of functions to convert between the result type of a conversion and theorems. However, once this is done the system and users' code becomes independent of the underlying implementation of conversions.

The delay of justification of a theorem can be combined with the ML data type approach to permit non-local optimisations, namely the elimination of the justification corresponding to repeated traversals. An extension is proposed that may optimise when subterms are discarded. The data structures used for the optimisations resemble HOL terms, but they only keep track of the changes brought about by the conversions. The structures introduce additional overheads into the system but these do not appear to cause significant degradation of performance and are usually outweighed by the benefits of reducing the number of primitive inferences.

The original basic conversions apply congruence rules to access subterms. However, the use of an ML data type with delay allows other rules such as substitution to be used without disturbing the modularity of the code. Experiments have been performed using a substitution rule, to determine whether it offers superior performance to congruence rules. There is not an enormous difference in the results for HOL88, but the congruence rules win.

7.1.4 Decision Procedures

The choice of decision algorithms in a fully-expansive theorem prover involves some special considerations. In a partially-expansive system, speed and space usage are the overriding concerns. Any algorithm that decides correctly whether a formula is true or false is acceptable. In a fully-expansive system the procedure must not only obtain a true/false answer but must also generate a proof of it by applying primitive inference rules. In particular, metatheoretic reasoning cannot be used. This significantly changes the issues since the fastest algorithms for partially-expansive

systems may make use of techniques for which it is difficult to obtain a rigorous proof. Often a simpler approach requires less primitive inferences. Since the cost of applying primitive inference rules may far outweigh other aspects of the procedure, a simple algorithm may be much faster than a sophisticated one.

A decision procedure for a subset of linear natural arithmetic is used to illustrate a number of optimisation techniques. A variable elimination method is used. It is straightforward to augment this algorithm with applications of primitive inference rules. The more popular SUP-INF method was rejected because it is rather complex and operates over the rational numbers, so a rigorous interpretation into the naturals would be required. However, an implementation of the SUP-INF method that does not use primitive inferences can be used to compute witnesses for existentially-quantified formulas.

The chosen procedure operates in two stages. First the negation of the formula to be proved is normalised. This produces a disjunction of conjunctions of arithmetic inequalities. The aim is to falsify each conjunction by eliminating variables until only an inequality between numeric constants remains.

The normalisation stage of the procedure is equational reasoning. The general optimisations for equational reasoning are exploited. The ordering of operations is chosen carefully in an attempt to minimise primitive inferences. Compound rewrite rules are used wherever possible since it costs more to apply several rewrites to perform an operation than just one. Memoisation is used for the evaluation by proof of products of numeric constants, and may also offer a solution to the problem of repeatedly processing subterms that have been duplicated by normalisation.

The variable elimination stage of the decision procedure forms pairs of inequalities for which the variable to be eliminated appears on opposite sides of the relation. A specialised version of lazy theorems is used. This allows the functions to be written so that they only apply primitive inferences for those pairs of inequalities that actually contribute to the derivation of false. The structure used in this version of lazy theorems is an abstract representation of normalised inequalities. Computation on these structures is more efficient than on terms. This leads to a generalised data type of lazy theorems parameterised over the representation structure. A proposal is made to use this data type to enable decision procedures to be combined without losing the performance benefits of abstract representations.

The linear arithmetic procedure has been implemented and a number of results are given. It is an order of magnitude faster than an earlier procedure for HOL with similar behaviour. Results are also given for a symbolic compiler. This proof procedure compiles programs represented by their semantics in HOL. It illustrates the benefits that can be obtained from delaying justification when a fast program is available to perform the initial stage of computation.

7.2 Conclusions for Implementors

The techniques proposed in this thesis have been tested by implementation and benchmarking. The conclusions from these exercises are summarised in this sec-

tion. There are two main topics: laziness and equational reasoning. There is some interaction between the two.

The use of lazy theorems does not provide as much delay for general reasoning as might have been hoped but it appears to be an overall optimisation (Section 3.10). Even when the lazy version of HOL is used in an eager mode there is very little degradation. For equational reasoning the delay obtained can be much greater but there may then be a small degradation in overall performance (Section 5.2.7). Laziness also enables some non-local optimisations to be made in equational reasoning (Section 4.6). The conclusion, therefore, is that laziness is worthwhile. Using a lazy mode of operation may not be a good idea when memory is short but since the system can readily be changed into an eager mode of operation this should present no problems (Section 3.7).

By far the most significant optimisation to equational reasoning is the avoidance of theorem generation for unchanged subterms. This can be achieved in a number of ways. Using exceptions (Section 4.4) has the advantage that it does not change the type of conversions (though it does change the semantics slightly) but programming with exceptions can be messy. Using an ML data type (Section 4.5) is cleaner for the user but requires significant changes to the whole HOL system because conversions no longer produce theorems. Since changing the system is a one-off exercise whereas difficulties with exceptions will keep occurring, the ML data type approach is to be preferred. This is reinforced by the possibility of further optimisation when the data type is combined with laziness in lazy structures (Section 4.6).

Use of lazy structures enables at least two additional optimisations: elimination of unnecessary theorem-generation for repeated term traversals (Section 4.6.1), and discarding inferences when destructive rewrite rules are used (Section 4.7). The former provides additional optimisation but this appears to be small in comparison with the unchanged-subterm optimisation. The same is basically true for the latter but the extent of optimisation depends on the number of destructive rewrite rules being used and, more importantly, whether any operations are being performed on the discarded subterm prior to the application of one of these rules. The overheads involved may make the technique a deoptimisation unless there is significant discarding of inferences. The technique also introduces a dynamic dependency into the lazy structures (Section 4.7.2). Without this, lazy structures can easily be integrated with more general laziness by making the argument to the `Lazy_eq_thm` constructor be a lazy theorem rather than an unwrapped justification function of type `void→thm`. With the dynamic dependency this function has to take a term as an argument so it cannot be replaced directly by a lazy theorem. Taken along with the questionable benefits of giving destructive rewrites special treatment, this leads to the conclusion that the technique is not worthwhile. However, it is worth keeping in mind for specialised optimisations. For example, destructive rewrites seem to be used a lot in symbolic compilation, though it is not clear that this leads to significant discarding of inferences (Section 5.3).

Two ways of generating a theorem from a lazy structure have been considered: application of congruence rules (Section 4.6.4), and substitution (Section 4.6.6). The results suggest that the application of congruence rules is faster but this is dependent

on the nature of those rules and on the substitution rule. In HOL88, these rules are all taken as primitive when strictly only the substitution rule should be. A different formulation might lead to the opposite conclusion.

Returning to lazy theorems, results for polymorphic lazy theorems indicate that a significant saving is made by keeping the structure of the lazy theorem outside the reference cell (Section 5.4). The downside to this is that there is then less certainty that the structure obtained on request actually corresponds to the structure of the theorem that will eventually be generated by the justification function.

7.3 Discussion of Laziness

The lazy techniques described in this thesis have three advantages:

- The justification stage of theorem generation is delayed, giving users greater control over its timing. Interactive response time is improved.
- The justification stage is avoided for theorems that are never used.
- Non-local optimisations can be made.

Lazy theorems are not fundamental to providing delay or optimisation. The same effects can be achieved in other ways. However, lazy theorems provide a uniform framework in which to use such techniques. Without a framework, a real theorem has to be generated between each of the ad hoc ‘optimisations’. This causes many of the benefits to be lost: The non-local optimisations do not build up when proof procedures are combined, and it is only when small delays are chained together into large delays that users gain the freedom to choose when to perform the justification stage of a proof.

In the worst case the initial stage and justification stage in Lazy HOL will each take as long as the full computation in HOL88. The total computation time in Lazy HOL will then be at least twice that for HOL88 (there may be additional costs due to, for example, extra garbage collection). However, the results given in Table 3.1 suggest that in practice the extra computation involved in getting an initial result is small and may even be compensated for by the optimisation provided by the lazy techniques. Laziness is not being exploited to any great extent in the benchmark. Nor have the derived rules been modified to make use of laziness. However, to do so is unlikely to improve the figures for HOL88 since the heavily-used derived rules are implemented as primitives, but it could make it more feasible to take them out of the critical code.

For certain kinds of proof, especially those that are normally considered too computationally-intensive in a fully-expansive theorem prover, lazy techniques can provide a significant reduction in the initial computation time. For example, Table 5.1 demonstrates that in Lazy HOL results can be obtained around three times faster for an arithmetic decision procedure, with only a small increase in the overall computation time. Much of the delay is due to the separation of search and justification facilitated by lazy theorems. To delay computation, all parts of the proof must

be lazy, including the equational reasoning. This is an advantage of lazy conversions over the use of exceptions.

Laziness does not breach the security of a fully-expansive theorem prover. An invalid formula cannot become a theorem because of the laziness. The worst that can happen is for users to be fooled for a while into thinking they have successfully proved a conjecture when in fact the justification function will not be able to generate a real theorem corresponding to the lazy theorem. Obviously, this would be annoying, but failure of lazy theorems produced by built-in functions should be no more common than errors in a partially-expansive theorem prover. The quality of user-defined functions is more questionable. New users should be discouraged from writing ‘optimised’ functions to protect them from the frustration of justification failures.

Lazy techniques make a time/space trade-off. The theorem prover becomes faster to use at the cost of higher memory usage. This is a good exploitation of large memories. However, when the amount of real memory becomes inadequate, swapping overheads seriously degrade the real-time performance. In this situation, a combination of the draft and eager modes of operation described in Section 3.7 could be used.

One of the most promising applications of laziness is in the use of fully-expansive theorem provers for the mechanisation of programming language semantics. This may involve symbolic compilation or execution of logical terms representing programs [Cam91]. The numbers of primitive inferences involved in these processes can be huge. Laziness offers the ability to perform the compilation and execution rapidly using an ML or Lisp program while retaining security by checking the result later using primitive inferences. This is particularly valuable because a program may be compiled and executed many times before it is perfected, yet a theorem is only required for the final run.

Another application is in the interface to external systems. For example, proof tasks (subgoals) could, given a suitable interface, be passed to an external theorem prover (such as the Boyer-Moore Theorem Prover [BM88a]) or a proof planner (such as CLAM [BvHHS91]). For a theorem to be generated by primitive inferences the external system must return not only a result but also a proof of the result. In a lazy system the result can initially be trusted to give the structural part of a lazy theorem while the proof returned is used as the justification function.

7.4 Prospects for Further Work

Although many of the techniques presented in this thesis are general-purpose (or at least wide-ranging) and largely transparent, some, like those described in Section 5.2, are quite specialised. A possible avenue for future research is automatic generation of proof procedures from specifications. For example, given some specification of the normalisation stage of the arithmetic decision procedure it would generate the ML code. Then, even though the optimisations cannot be embedded in the basic functions for equational reasoning, they would be embodied instead in the knowledge

of the proof-procedure generator. For example, it could decide whether to traverse terms top-down or bottom-up at various stages, and also construct the program so as to minimise the number of traversals required.

Whether a proof procedure is optimal may depend on the context in which it is being used. Recall, for example, the problem of duplication in the normalisation procedure (Section 5.2.4). Which expansions to use for Boolean equalities depends on whether the term is going to be put into disjunctive or conjunctive normal form. Hence, a proof-procedure generator would have to take a global view in order to obtain optimal efficiency. It is not always clear that one approach is better than another. There may be two or more conflicting issues. Optimal performance may, therefore, be dependent on the formula and on the execution times of various system functions. Thus, it seems unlikely that a generator will be able to produce a procedure that is optimal for all possible arguments.

So, constructing an optimising proof-procedure generator will not be easy. However, given the difficulty of writing proof procedures in HOL without considering efficiency, such a tool would be of benefit even if it did not optimise effectively. Higher-level tools such as these may hold the key to the future of the HOL system as an environment for supporting formal reasoning in computer science and related disciplines.

The lazy structures used to optimise equational reasoning (Section 4.6.1) are specific to the term structure and the built-in equivalence relation of the HOL system. However, the optimisations are more widely applicable to situations in which structural rules are being applied. It would be interesting to see whether the idea can be adapted to other equivalence relations such as those that arise in embedded languages like CCS [Nes92]. Congruence rules are required in addition to the equivalence relation. In the case of an embedded language these rules are with respect to defined operators rather than the raw syntax of HOL (combinations and abstractions). The applicability of the rules may also be dependent on side-conditions. Could the abstract type of equations proposed in Section 4.8 be parameterised over a congruence relation? The functions `RATOR_CONV`, `RAND_CONV`, and `ABS_CONV` would have to be replaced by a list of the congruence rules.

The linear arithmetic decision procedure presented in this thesis is a significant step forward. For the first time, HOL users are free from the burden of having to prove trivial lemmas of natural number arithmetic, an activity which used to take tens of minutes by hand for each lemma. The procedure proves many of the lemmas commonly arising in verification proofs in just a few seconds. It should be straightforward to implement similar procedures for the integers and the reals. It may be possible to produce a generic procedure that can be instantiated with the axioms of the arithmetic theory in question (actually derived theorems in HOL), though having to handle negative values and dense/non-dense orderings differently might mean that too many special cases have to be considered for it to be worthwhile.

A more interesting prospect is the idea of combining a decision procedure for arithmetic with ones for the theory of lists, other structured types, uninterpreted function symbols, and possibly set theory. Alone, the arithmetic procedure is useful for proving lemmas but rarely is it powerful enough to prove users' theorems or even

subgoals generated during an interactive proof. Combined with decision procedures for other theories it would be able to handle a broader class of formulas that arise more commonly as subgoals. From the point-of-view of this thesis, the interesting question is to what extent the optimisations proposed can be maintained when decision procedures are combined. This is discussed at greater length in Section 5.5.

As also mentioned in Section 5.5, the prospects of maintaining the optimisations when integrating decision procedures with heuristic theorem-proving techniques seem bleak. However, Bundy has suggested that proof plans may provide a means of integration that does not destroy the separate identities of the procedures [Bun91]. To what extent this would be efficient in HOL probably boils down to the question of how efficient are the proofs generated by the planner, or how efficient the tactics they call can be made.

Bibliography

- [ACHA90] S. F. Allen, R. L. Constable, D. J. Howe, and W. E. Aitken. The semantics of reflected proof. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, pages 95–105. IEEE, June 1990.
- [Age92] S. Agerholm. Mechanizing program verification in HOL. Master’s thesis, Computer Science Department, Aarhus University, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark, April 1992. Technical Report DAIMI IR – 111.
- [AHL91] Abstract Hardware Limited, Brunel University, Uxbridge, Middx., U.K. *The LAMBDA System – Complete Reference Set, v. 4.1*, 1991.
- [Ben89] D. Benanav. Recognizing unnecessary inference. In N. S. Sridharan, editor, *Proceedings of the 11th Int. Joint Conf. on Artif. Intell.*, pages 366–371. Morgan Kaufmann, August 1989.
- [BGG⁺92] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In Stavridou et al. [SMB92], pages 129–156.
- [BH80] W. W. Bledsoe and L. M. Hines. Variable elimination and chaining in a resolution-based prover for inequalities. In *Proceedings of the 5th Conference on Automated Deduction*, volume 87 of *Lecture Notes in Computer Science*, pages 70–87. Springer-Verlag, 1980.
- [Ble75] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *Advance Papers of the 4th Int. Joint Conf. on Artif. Intell.*, Tbilisi, Georgia, U.S.S.R., pages 15–21, September 1975.
- [BM79] R. S. Boyer and J S. Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.
- [BM81] R. S. Boyer and J S. Moore. Metafunctions: Proving them correct and using them efficiently as new proof procedures. In R. S. Boyer and J S. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

- [BM88a] R. S. Boyer and J S. Moore. *A Computational Logic Handbook*, volume 23 of *Perspectives in Computing*. Academic Press, San Diego, 1988.
- [BM88b] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. In J. E. Hayes, D. Michie, and J. Richards, editors, *Machine Intelligence 11*, chapter 5, pages 83–124. Oxford University Press, 1988.
- [Bou92] R. J. Boulton. Boyer-Moore automation for the HOL system. In Claesen and Gordon [CG92], pages 133–142.
- [Bro80] F. M. Brown. An investigation into the goals of research in automatic theorem proving as related to mathematical reasoning. *Artificial Intelligence*, 14(3):221–242, 1980.
- [Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.
- [Bun83] A. Bundy. *The Computer Modelling of Mathematical Reasoning*. Academic Press, 1983.
- [Bun91] A. Bundy. The use of proof plans for normalization. In R. S. Boyer, editor, *Essays in Honor of Woody Bledsoe*, pages 149–166. Kluwer, 1991. Also: Research Paper 513, Department of Artificial Intelligence, University of Edinburgh, November 1990.
- [BvHHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, September 1991.
- [C⁺86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice Hall, Englewood Cliffs, New Jersey, 1986.
- [Cam91] J. Camilleri. Symbolic compilation and execution of programs by proof: A case study in HOL. Technical Report 240, University of Cambridge Computer Laboratory, December 1991.
- [CG92] L. J. M. Claesen and M. J. C. Gordon, editors. *Higher Order Logic Theorem Proving and its Applications: Proceedings of the IFIP TC10/WG10.2 Workshop*, volume A-20 of *IFIP Transactions*, Leuven, Belgium, September 1992. North-Holland/Elsevier.
- [Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

- [CKM⁺91] D. Craigen, S. Kromodimoeljo, I. Meisels, W. Pase, and M. Saaltink. EVES: An overview. In S. Prehn and W. J. Toetenel, editors, *VDM'91 Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 389–405. Springer-Verlag, 1991.
- [Coo72] D. C. Cooper. Theorem proving in arithmetic without multiplication. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, chapter 5, pages 91–99. Edinburgh University Press, 1972.
- [CS90] D. Craigen and K. Summerskill, editors. *Formal Methods for Trustworthy Computer Systems (FM89)*, Workshops in Computing. Springer-Verlag, 1990.
- [Cur92] P. Curzon. Deriving correctness properties of compiled code. In Claesen and Gordon [CG92], pages 327–346.
- [DS79] M. Davis and J. T. Schwartz. Metamathematical extensibility for theorem verifiers and proof-checkers. *Computers and Mathematics with Applications*, 5:217–230, 1979.
- [GG91] S. J. Garland and J. V. Guttag. A guide to LP, the Larch prover. Report 82, Digital Equipment Corporation Systems Research Center, December 1991.
- [GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [GMM⁺78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in LCF. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 119–130, Tucson, Arizona, January 1978.
- [GMW79] M. J. Gordon, A. J. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [Goo84] D. Good. Mechanical proofs about computer programs. Technical Report 41, Institute for Computing Science, University of Texas at Austin, 1984.
- [Gor86] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In G. Milne and P. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Conference on VLSI*, pages 153–177. North-Holland, 1986. Also available as Technical Report 77, University of Cambridge Computer Laboratory.

- [Gru92] J. Grundy. A window inference tool for refinement. In C. B. Jones, B. T. Denvir, and R. C. F. Shaw, editors, *Proceedings of the 5th Refinement Workshop*, pages 230–254, Lloyd’s Register, London, January 1992. Workshops in Computing, Springer-Verlag.
- [GWM⁺92] J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.-P. Jouannaud. Introducing OBJ. In *Applications of Algebraic Specifications using OBJ*. Cambridge University Press, 1992. Also: Technical Report SRI-CSL-92-03, Computer Science Laboratory, SRI International, Menlo Park CA 94025.
- [Har93] J. Harrison. A decision procedure for elementary real algebra. In Joyce and Seger [JS93].
- [HD92] F. K. Hanna and N. Daeche. Dependent types and formal synthesis. In C. A. R. Hoare and M. J. C. Gordon, editors, *Mechanized Reasoning and Hardware Design*, International Series in Computer Science, pages 121–135. Prentice Hall, 1992. First published in the *Philosophical Transactions of the Royal Society of London*, Series A, Volume 339, 1992.
- [HDH92] K. Hanna, N. Daeche, and G. Howells. Implementation of the Veritas design logic. In Stavridou et al. [SMB92], pages 77–94.
- [HO80] G. Huet and D. C. Oppen. Equations and rewrite rules: A survey. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*. Academic Press, New York, 1980. Also: Technical Report CSL-111, Computer Science Laboratory, SRI International, Menlo Park, California, January 1980.
- [Hod71] L. Hodes. Solving problems by formula manipulation. In *Advance Papers of the 2nd Int. Joint Conf. on Artif. Intell.*, London, pages 553–559. The British Computer Society, September 1971. Revised version in *Artificial Intelligence*, 3:165–174, 1972.
- [How88] D. J. Howe. Computational metatheory in Nuprl. In *Proceedings of the 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 238–257. Springer-Verlag, 1988.
- [HT93] J. Harrison and L. Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In Joyce and Seger [JS93].
- [Jac92] P. B. Jackson. Nuprl and its use in circuit design. In Stavridou et al. [SMB92], pages 311–336.
- [JAR89] *Journal of Automated Reasoning*, volume 5, number 4. Special issue on system verification, 1989.

- [JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. International Series in Computer Science. Prentice-Hall, 1993.
- [Joy89] J. J. Joyce. Totally verified systems: Linking verified software to verified hardware. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, volume 408 of *Lecture Notes in Computer Science*, pages 177–201, Ithaca, N.Y., July 1989. Springer-Verlag.
- [JS93] J. Joyce and C.-J. Seger, editors. *Proceedings of the 1993 International Meeting on Higher Order Logic Theorem Proving and its Applications (HUG'93)*, The University of British Columbia, Vancouver, Canada, August 1993. To be published as a volume of *Lecture Notes in Computer Science*. Springer-Verlag.
- [KC86] T. B. Knoblock and R. L. Constable. Formalized metareasoning in type theory. In *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pages 237–248. IEEE, 1986.
- [KP92] S. Kromodimoeljo and W. Pase. Final report for the investigation of proof techniques within the Eves verification technology. Final Report FR-92-5451-02, ORA Canada, 265 Carling Avenue, Suite 506, Ottawa, Ontario K1S 2E1, Canada, May 1992.
- [Käu88] T. Käufel. Reasoning about systems of linear inequalities. In *Proceedings of the 9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 563–572. Springer-Verlag, 1988.
- [KZ89] D. Kapur and H. Zhang. An overview of RRL (Rewrite Rule Laboratory). In N. Dershowitz, editor, *Proceedings of the Third International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 559–563, Chapel Hill, North Carolina, USA, April 1989. Springer-Verlag.
- [Mar85] P. Martin-Löf. Constructive mathematics and computer programming. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 167–184. Prentice Hall, 1985.
- [Mar91] L. Marcus. SDVS 10 users' manual. Technical Report ATR-91(6778)-10, The Aerospace Corporation, 1991.
- [McM93] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
- [Mel89] T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*.

- Springer-Verlag, 1989. Also available as University of Cambridge Computer Laboratory Technical Report 146.
- [Mic68] D. Michie. “Memo” functions and machine learning. *Nature*, 218:19–22, April 1968.
- [Mil72] R. Milner. Implementation and application of Scott’s logic for computable functions. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, 1972. ACM SIGPLAN Notices 7(1).
- [MoD91] Interim defence standard 00–55: The procurement of safety critical software in defence equipment, part 2: Guidance. Ministry of Defence, Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow, G2 8EX, U.K., April 1991.
- [MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [Mus89] D. R. Musser. Report on the HOL (Higher Order Logic) proof checker. Computer Science Department, Rensselaer Polytechnic Institute, 1989.
- [Nel80] G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980. Revised version: Technical Report CSL-81-10, Xerox PARC, June 1981.
- [Nes92] M. Nesi. A formalization of the process algebra CCS in higher order logic. Technical Report 278, University of Cambridge Computer Laboratory, December 1992.
- [Nip89] T. Nipkow. Equational reasoning in Isabelle. *Science of Computer Programming*, 12:123–149, 1989.
- [NO79] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.
- [Pau83] L. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.
- [Pau87] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.
- [Pau90] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.
- [Pau91] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

- [Pey87] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. International Series in Computer Science. Prentice Hall, 1987.
- [Pie90] W. Pierce. Toward mechanical methods for streamlining proofs. In M. E. Stickel, editor, *Proceedings of the 10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 351–365, Kaiserslautern, FRG, July 1990. Springer-Verlag.
- [Raj92] P. S. Rajan. Executing HOL specifications: Towards an evaluation semantics for classical higher order logic. In Claesen and Gordon [CG92], pages 527–536.
- [Rus90] J. Rushby. Formal methods and critical systems in the real world. In Craigen and Summerskill [CS90], pages 121–125.
- [Sha85] N. Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.
- [Sho77] R. E. Shostak. On the SUP-INF method for proving Presburger formulae. *Journal of the ACM*, 24(4):529–543, October 1977.
- [Sho78] R. Shostak. Deciding linear inequalities by computing loop residues. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 1978.
- [Sli91] K. Slind. An implementation of higher order logic. Master’s thesis, Department of Computer Science, University of Calgary, 2500 University Drive N.W., Calgary, Alberta, Canada T2N 1N4, January 1991. Research Report 91/419/03.
- [Sli92] K. Slind. Adding new rules to an LCF-style logic implementation. In Claesen and Gordon [CG92], pages 549–559.
- [SMB92] V. Stavridou, T. F. Melham, and R. T. Boute, editors. *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, Nijmegen, The Netherlands, June 1992. North-Holland/Elsevier.
- [SOR93] N. Shankar, S. Owre, and J. M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, Menlo Park CA 94025, March 1993. Beta Release.
- [Spi89] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, London, 1989.

- [Sym93] D. Syme. Reasoning with the formal definition of Standard ML in HOL. In Joyce and Seger [JS93].
- [VG93] M. VanInwegen and E. Gunter. HOL-ML. In Joyce and Seger [JS93].
- [vW91] J. von Wright. Mechanising the Temporal Logic of Actions in HOL. In M. Archer, J. J. Joyce, K. N. Levitt, and P. J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 155–159, Davis, California, August 1991. IEEE Computer Society Press.
- [vW94] J. von Wright. Representing higher-order logic proofs in HOL. Technical Report 323, University of Cambridge Computer Laboratory, January 1994.
- [Wey80] R. W. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. *Artificial Intelligence*, 13:133–170, 1980.
- [Won93] W. Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, July 1993.