INDUCT: A Logical Framework for Induction Over Natural Numbers and Lists Built in **SEQUEL**

Andrew Alexander Adams

The University of Leeds
School of Computer Studies

Submitted: September 1994

Submitted in accordance with the requirements for the degree of MSc.

Abstract

SEQUEL [Tar93a] is a new language built on top of Lisp and designed for implementing theorem provers and proof assistants. It includes a type theory for Lisp (XTT — eXtended Type Theory) and functions for sequent calculus and rewriting systems.

nqthm [BM79], a theorem prover for untyped Lisp, is a powerful tool for proving properties of functions. Producing a framework written in **SEQUEL**, which implements a similar system for the typed Lisp of **SEQUEL** could add an important feature to the type-checker.

The project presented here involves a description of such a framework, and the implementation details of programming such a system in **SEQUEL**'s sequent calculus notation.

The framework, called **INDUCT**, is based on the theory laid out in Boyer and Moore's 1979 Book 'A Computational Logic'. The current versions of **nqthm** are more powerful in some ways and less in others — relying on the user giving more information, allowing deeper search through a narrower search space.

INDUCT is designed to make available to the user various levels of interaction in the proof of theorems, and to make use of the validity of the **SEQUEL** type-checker to narrow its search space and shorten its proofs. It also includes some refinements which have been suggested since the original text.

The project has been limited to producing a framework dealing with Natural Numbers and Lists. Extension of the framework to other Lisp types and then to user-defined **SEQUEL** types should be possible, since little of the framework is dependent upon the nature of the types, merely upon the theoretical requirements common to **INDUCT** and **nqthm**. Areas which would require more than rudimentary changes to achieve these ends will be highlighted and suggestions made as to how these changes could be implemented.

Finally, it will be shown how **INDUCT** can be used as a substitute for **nqthm** in the areas it does cover, and how the **SEQUEL** *proof tool* and the interaction with the framework can enable a user to produce valid proofs.

Contents

Chapter 1

Introduction

SEQUEL is a new language developed by Mark Tarver at Leeds. Much of the philosophy of **SEQUEL** derives from the 'proof as type-checking' principle put forward by him in [Tar93a].

This principle may be taken a step further, in an obvious fashion: if proof is equivalent to type-checking, then since program verification is merely proof, the process of verification can also be reduced to type-checking. The type-checking thus required would of course be far more involved than for example testing a program to ensure that particular functions are only given integers as inputs. For instance we might wish to show that the tail recursive version of a function is equivalent to the simple recursive form. Verification of such conditions requires inductive proof.

The theory and implementation of inductive proof for a subset of Lisp has had much work done on it over the last twenty-odd years, the most well-known being the theory described in [BM79], which was the basis of the theorem prover **nqthm**. However, **nqthm** took a team of researchers a number of years to produce. Since **SEQUEL** has been designed for the implementation of theorem proving environments, implementing a framework to perform inductive proofs over the Lisp expressions produced by the **SEQUEL** compiler would seem much simpler than the implementation of **nqthm**. With the succinctness of code and the tools for theorem prover implementation provided by **SEQUEL**, it seemed feasible to undertake the production of a basic system as an MSc project.

The problems with such an undertaking are the translation of the logic represented by Boyer and Moore in [BM79] to a sequent calculus notation as used by **SEQUEL**, and the coding of the constructive nature of this logic into **SEQUEL**'s fairly static structure.

The resulting framework is only the first step in constructing a version of **SEQUEL** including the advanced features implied. It is however a very necessary first step, which has been done with a view to further work in this area.

Chapter 2

An Overview of SEQUEL

SEQUEL is a new functional language designed and implemented by Mark Tarver to enable the rapid prototyping of Proof Assistants (PAs) and Automated Theorem Provers (ATPs) by reducing the amount of code required for such systems. It is built in and on top of Lisp, and splits neatly into two sections: the core language (which allows access to all Lisp functions) and the sequent calculus extensions.

2.1 The Core Language

The top level of **SEQUEL**, as with all functional programming languages, is similar to the command level of a text-input operating system. The user types in a command or *expression*, which the system reads, evaluates to produce a result, and prints that result. This is called the read-eval-print loop. The simplest expressions are ones which do not change in the evaluation, such as integers (1, 2, 3, ...) and symbols (a, b, xyz, ...). These expressions are called self-evaluating expressions because they evaluate to themselves. A more complex expression consists of the application of a function to a set of arguments, for example (+ 1 3), which evaluates to 4, the result of applying the function + to the arguments 1 and 3. The arguments of a function call need not be self-evaluating expressions, but can themselves be function calls. BNF grammars for various portions of **SEQUEL**, including the top-level, can be found in Appendix A. Core **SEQUEL** has a syntax derived from Prolog. *W, x, y, z* and any symbol ending with *?* are variables, and lists are constructed using [ ] and |. Thus [1] is the list containing the single element 1, [1 2 3] is the list containing three elements 1, 2 and 3 in that order and:

$$[1\ 2 \mid [3]] = [1\ 2\ 3] = [1 \mid [2 \mid [3]]].$$

SEQUEL is a rewrite language, in which functions are specified by means of a list of rewrite rules, of the form $i_1\ i_2\ \ldots\ i_n \rightarrow o$, where all the variables in the formal result o must also appear in the formal arguments $i_1, i_2, \ldots i_n$. A rule is applicable if the formal arguments can be matched with the actual arguments. Matching these arguments requires the same structure for both, given that variables in the formal arguments will match with any structure in the actual arguments, although any variable which appears more than once within $i_1, i_2, \ldots i_n$ will only match with the same structure in both positions. If two or more rules could both be applied (such rules being referred to as being *in conflict*) then the rule appearing first in the function definition is applied. This is called a priority ordering. So, a test for zero can be written thus:

```
0 -> true
x -> false
```

where the second rule could be applied to the input `0`, except that the first rule will always have fired on this input.

As well as the requirements for the formal arguments to match with the actual arguments, extra conditions may be placed on the firing of a rule. These conditions, called *guards*, are evaluated in the order that they appear. Evaluation of guards and tests for a match between formal and actual arguments are carried out in *lazy* fashion, where guards and tests for matching are evaluated one at a time, and the result of each evaluation checked before evaluating the next in the sequence. The alternative route is called *eager* evaluation, where all the evaluations are performed first, and then the results are checked to see if they match the requirements. So, with lazy evaluation, for a rule

```
x (foo x) [y z | w] -> (bar w x y z)
```

the structure of the second actual argument will only be checked if `(foo x)` returns *t*. If any of the guards returns the value *nil* then the rule does not fire, and the next rule in the definition of the function will be applied. The **SEQUEL** code below shows how to define a modulus function using guards.

```
(defun modulus
x y (< x y) -> x
x y -> (modulus (- x y) y))
```

Backtracking is a useful feature of many functional and logic programming languages, but it is often implemented in a confusing and complicated manner. **SEQUEL**'s backtracking is embodied in the principle of *semantic backtracking*, which is straightforward to use, and allows easy understanding of the flow of control within a function. A *failure object* is defined, and semantic backtracking is indicated by using a left hand arrow instead of a right hand arrow for the rule. If the actual output of a rule using a left hand arrow is the failure object then execution passes down the list of rules as if the rule had not fired. The easiest way to return the failure object is to use the function *fail*. Thus we might have:

```
(define nqueen
  x -> (nqueen2 x [] 1))

(define nqueen2
  n? pos? _ (equal (length pos?) n?) -> pos?
  n? pos? x (> x n?) -> (fail)
  n? pos? x
```

```
      (not (member x pos?))
      (nq-consistent x (reverse pos?) 1)
    <- (nqueen2 n? [pos? <> [x]] 1)
  n? pos? x -> (nqueen2 n? pos? (1+ x)))

(define nq-consistent
  _ [] _ -> t
  x [pos? | poss?] n? (equal (abs (- x pos?)) n?) -> nil
  x [_ | poss?] n? -> (nq-consistent x poss? (1+ n?)))
```

which calculates a solution to the problem of placing n queens on an $n \times n$ chessboard so that none of them can take another in one move. (The underscore character in the input patterns represents a place-holder, which introduces a new anonymous variable each time it occurs in a rule. It is used to improve clarity and gains some slight improvement in the speed of compilation to Lisp code.)

The other way to produce the failure object is to use the function *fail-if*. Instead of *fail* being a possible result of an output, *fail-if* may be used to test the result of an output, so that in place of $i_1\ i_2\ \ldots i_n \leftarrow o$, we have $i_1\ i_2\ \ldots i_n \leftarrow$ (*fail-if* λ*-expression* o), where the λ-expression is an anonymous function of a single argument. If the result of evaluating the λ-expression is *nil* then backtracking occurs, otherwise the result of evaluating o is returned. *Fail* is used in the result function, when it is possible to detect failure during its execution. *Fail-if* is used when failure can only be detected once the result function has been evaluated.

Semantic backtracking is a clean and simple system for controlling of backtracking, since its use is only a syntactic shorthand: all uses of $\leftarrow$ can be replaced by rules using only the $\rightarrow$ via a static algorithm. **SEQUEL** code using semantic backtracking is usually shorter and clearer than the equivalent **SEQUEL** code without backtracking. For instance, say we wanted to try a heuristic method of solving a problem, but revert to a brute force method if the heuristics failed. The backtracking code for doing this (taken from [Tar93b]) is

```
(define solve-problem
  problem? <- (solve-by-heuristics problem?)
  problem? -> (solve-by-brute-force problem?))
```

The equivalent code without backtracking is

```
(define solve-problem
  problem?
      (equal (solve-by-heuristics problem?)
             (eval *failure-object*))
    -> (solve-by-heuristics problem?)
  problem? -> (solve-by-brute-force problem?))
```

2.2 Sequent Calculus Extensions to SEQUEL

2.2.1 Sequent Calculus

Sequent calculus[1] is a system for reasoning with sequents, where a sequent is an expression of the form $\Gamma \vdash \Delta$ (with Γ and Δ both being lists of propositions), which can be read as *'If all propositions in Γ are true then at least one proposition in Δ is true'*. The $\vdash$ symbol is called a turnstile and can be read as 'implies' although its technical meaning is that what follows it can be deduced from what precedes it. The Γ is called the *antecedent*, and the Δ is called the *consequent*. The empty sequent ($\vdash$) signifies a true (but trivial) theorem.

A proof in a sequent calculus is an object of the form:

$$\dfrac{\dfrac{\begin{matrix}\vdash\\ \vdots\end{matrix}}{\Gamma_1 \vdash \Delta_1} \quad \cdots \quad \dfrac{\begin{matrix}\vdash\\ \vdots\end{matrix}}{\Gamma_n \vdash \Delta_n}}{\dfrac{\vdots}{\Gamma \vdash \Delta}}$$

Production of a proof that a sequent $\Gamma \vdash \Delta$ is a theorem may be performed by forward or backward chaining, i.e. one may start with only the trivial theorem ($\vdash$) and a set of rules, and work down (forward chaining), eventually producing $\Gamma \vdash \Delta$, or start with $\Gamma \vdash \Delta$ and work upward (backward chaining).

The general schema for a sequent calculus rule is:

$$\frac{(G_{11}, \ldots G_{1i}, \Gamma_1 \vdash D_{11}, \ldots D_{1j}, \Delta_1) \cdots (G_{h1}, \ldots G_{hk}, \Gamma_h \vdash D_{h1}, \ldots D_{hl}, \Delta_h)}{G_{01}, \ldots G_{0m}, \Gamma_0 \vdash D_{01}, \ldots D_{0n}, \Delta_0},$$

where the G_{xy} and the D_{xy} are all propositions and the Γ_z and Δ_z are sets of propositions.

Backward chaining can be thought of as using such a rule thus: the sequent below the line can be *discharged* (proved to be a theorem) if the sequent(s) above the line can be discharged. A sequent has been discharged when a rule with only the empty sequent above the line has been applied.

A specific example of a sequent calculus rule is the 'or-left rule' for propositional calculus:

$$\frac{t_1, \Gamma \vdash \Delta \qquad t_2, \Gamma \vdash \Delta}{(t_1 \vee t_2), \Gamma \vdash \Delta}.$$

So, suppose we have a sequent $\neg B, (A \vee B) \vdash A$, then instantiating t_1 as A, t_2 as B, Γ as $(\neg B)$ and Δ as (A), we have the following:

$$\frac{A, \neg B \vdash A \qquad B, \neg B \vdash A}{(A \vee B), \neg B \vdash A}.$$

Meaning that we can prove that $(A \vee B), \neg B \vdash A$ is a theorem providing we can prove that both $A, \neg B \vdash A$ and $B, \neg B \vdash A$ are theorems.

Extra conditions on the application of sequent calculus rules can also be specified — these are called side-conditions; for example:

$$\frac{f(a), \Gamma \vdash \Delta}{\exists x.f(x), \Gamma \vdash \Delta} \text{ where a does not appear in } \Gamma \text{ or } \Delta.$$

[1]Referring to general sequentzen systems (see [Dil90]), not simply the sequent calculus implementation of First Order Predicate Calculus.

2.2.2 Use of Sequent Calculus in SEQUEL

Returning to the *proof as type-checking* principle underlying **SEQUEL**, a single notation is required for describing proof procedures and type checking procedures. There are a number of clearly described and easy to understand notations for specifying proof procedures, for example the sequent calculus described above. Type-checking procedures are usually written in languages developed for compiler-compilers (for example the Unix program YACC), and are therefore neither easily expanded for other use nor generally well understood. So it would appear to be much easier to develop a notation for specifying type-checking procedures by expanding a notation for specifying proof procedures. The end result is that **SEQUEL** uses sequent calculus notation for both purposes: the specification of types and the specification of theorem provers.

To improve the readability of **SEQUEL**'s sequent calculus, and to allow for more efficient compilation to Lisp code, via Horn-clauses, **SEQUEL** uses single-conclusion sequent calculus, so in the sequent $\Gamma \vdash \Delta$, Δ contains only a single proposition. This loss of generality does not effect modeling **nqthm**. Production of a proof in **SEQUEL** is done by backward chaining, so when using **SEQUEL** to produce proofs, a sequent is specified which is then manipulated via the rules in the proof procedure to produce the empty sequent at the top of all branches.

The function *theory* is used to define both logical systems and types, with the keyword ":interactive yes" for logics and (optionally) ":interactive no" for types. Examples of interactive theories for logics and non-interactive theories for types are given later.

2.2.3 The SEQUEL Type System

SEQUEL has a strong, static type system, XTT, which contains type information for over 300 Common Lisp functions. Use of this type system is optional, and the type-checking is declarative and fully transparent to the user through the type tracing mechanism. Type-checking within **SEQUEL** is essentially the automatic production of a proof that an object inhabits a certain type. There are two sorts of type within **SEQUEL**: *primitive* types and *general* types.

To define a primitive type, we define a unary predicate P. We can then define a primitive type P_t, by stating that x is of type P_t iff $(P\ x)$ is true. For instance the sub-type *bignum* of integers (specified as all integers bigger than 100) can be defined as a primitive type by defining a function *bignumber* thus:

```
(define bignumber
  x (integerp x) -> (> x 100))
```

and then by declaring *bignum* to be a primitive type defined by the function *bignumber*:

```
(primitive bignum bignumber)
```

Whilst this is an easy exercise for simple types such as *bignum*, the amount of coding required for more complex types would be significant. **SEQUEL** therefore allows the specification of types by sequent calculus rules specified using non-interactive theory definitions. The specification in sequent calculus of the type binary, implemented as a list of 1s and 0s, would be:

$$\frac{\Gamma \vdash X\colon \mathit{Binary}}{\Gamma \vdash [1|X]\colon \mathit{Binary}} \quad \frac{\Gamma \vdash X\colon \mathit{Binary}}{\Gamma \vdash [0|X]\colon \mathit{Binary}} \quad \frac{\vdash}{\Gamma \vdash []\colon \mathit{Binary}}$$

The corresponding **SEQUEL** code would be:

```
(theory binary
  <A> |- x * binary
  iff
  <A> |- [1 | x] * binary; rule 1

  <A> |- x * binary
  iff
  <A> |- [0 | x] * binary; rule 2

  thus
  <A> |- [] * binary); rule 3
```

'Thus' indicates a possible route for proving the sequent below the line (in rule 3, since there is no sequent above the line the 'thus' indicates successful discharge of a sequent) while the 'iff' (in rules 1 and 2) indicates that the only possible way of proving the sequent below the line is by proving the sequent(s) above the line. The type-checker will work through an input [0 1 1 0] in the following way:

[0 1 1 0] * binary

iff; rule 2

[1 1 0] * binary

iff; rule 1

[1 0] * binary

iff; rule 1

[0] * binary

iff; rule 2

[] * binary

thus; rule 3

QED.

This automatic proof production is performed by the **SEQUEL** type-checker for direct inputs such as `[0 1 1 0] * binary`, and for functions such as *binor*:

```
(define binor
  {binary binary -> binary}
  [] [] -> []
  [1 | x] [_ | y] -> [1 | (binor x y)]
  [_ | x] [1 | y] -> [1 | (binor x y)]
  [_ | x] [_ | y] -> [0 | (binor x y)])
```

2.2.4 Sequent Calculus Specification of Theorem Provers

SEQUEL was developed in order to allow the high-level specification of theorem proving systems. Theorem provers are used to reason about knowledge encoded in a logic. A structure for describing a logic is therefore required. This structure is comprised of:

A Syntax: The form of a proposition.

A Semantics: The meaning of a proposition.

A Proof Procedure: A specification of how to construct a proof of a theorem.

A Set of Pragmatics: A description of how to use the proof procedure in an efficient manner.

The semantics of a logic depends on the user assigning meaning to the propositions being reasoned about, and thus **SEQUEL** is not concerned with the semantics of a logic. The other parts of a logic — the syntax, proof procedure and pragmatics — form the representation of a logic in **SEQUEL**— called a *logical framework.*

To represent a logic in **SEQUEL**'s sequent calculus, we first define some types: *wff, type* and *t-expr*. *Wff* is short for well-formed formula. A *type* is exactly what is suggested by its name — the name of a type or a formula describing a type. *T-expr* is short for typed expression — a list of three elements, *(wff? * type?)* where *wff?* is of type *wff* and *type?* is of type *type*. In **SEQUEL**'s sequent calculus, a *t-expr* is a proposition, so in the sequent $\Gamma \vdash \Delta, \Gamma$ is a list of *t-exprs* and Δ is a single *t-expr*.

A *sequent* in a logical framework is a list with three elements: a list of *t-exprs* (the antecedent or hypotheses), a turnstile (represented in ascii text as |–), followed by another *t-expr* (the consequent or goal). So, if we have the *sequent*

$$[[[w_1 \; * \; t_1] \ldots [w_n \; * t_n]] \vdash [w_g \; * t_g]],$$

this is to be read as *'assuming that* w_1 *is of type* t_1 *and ... and that* w_n *is of type* t_n*, show that* w_g *is of type* t_g*'*, showing again the logics-as-types principle. For simple logics the types $t_g, t_1 \ldots t_n$ might all be just the same symbol, *thm* for example. For more complex logics such as a constructive type theory, the types would have a more complex syntax and manipulation of the *types* in the *t-exprs* could form as much or more of the proof procedure as manipulation of the *wffs*.

The types *proof-object* and *proof* are also defined. An object of type *proof* is a representation of what remains to be proved before the original goal has been proved to be a theorem. An object of type *proof-object* is merely a list of *sequents.* To be of type *proof,* an object must be of type *proof-object*, and have been returned by a function with the result type *proof.* The system functions for submitting a theorem to a logical framework produce the initial object of type *proof*, and type-checked *tactics*[2] preserve this through the succeeding proof attempt.

So, we have an internal representation of an ongoing proof, namely a list of sequents to be discharged. Next, a means of discharging these sequents is required. Two basic operations are

[2]Functions which encode the pragmatics of producing a proof in the logic, which have been type-checked to have result type *proof.*

available for the manipulation of sequents: logical rewrite rules and refinement rules. A logical rewrite operates on a *t-expr*, replacing one proposition within the current *sequent* with another. Refinement works on the current *sequent*, replacing it with zero or more new *sequents*.

Logical rewrite rules are defined using the function *defrew* and are of type

(t-expr $\rightarrow$ t-expr) or *(t-expr α $\rightarrow$ t-expr)*.

where the extra argument (α) of the second form is to allow the passing of extra information, used to improve control over the use of the rule. They are defined using the same priority-rewrite language as ordinary **SEQUEL** functions. Thus a simple example of a logical rewrite rule for a propositional logic framework would be to rewrite the *t-expr*

$((a \Leftrightarrow b) : thm)$ to $(((a \Rightarrow b) \,\&\, (b \Rightarrow a)) : thm)$.

This would be:

```
(defrew d-imp
  {t-expr -> t-expr}
  [[x <=> y] * thm] -> [[[x => y] & [y => x]] * thm])
```

Logical rewrites defined thus are called with the function *rewrite*. This has the type

(integer (t-expr $\rightarrow$ t-expr) proof $\rightarrow$ proof)[3]

where the *integer* indicates which *t-expr*[4] in the top sequent is to be rewritten, the *(t-expr $\rightarrow$ t-expr)* is the name of the logical rewrite to be applied and *proof* is the current proof-object. If a logical rewrite rule is applied to a *t-expr* which is not of the correct form (i.e. none of the rules defining the logical rewrite fire) then the call of *rewrite* merely acts as the identity function. This allows speculative application of logical rewrite rules. Thus if the head of *proof?* is the sequent $\vdash (A \Leftrightarrow B){:}thm$ then applying *(rewrite 0 d-imp proof?)* returns *proof?* with the head sequent $\vdash ((A \Rightarrow B) \wedge (B \Rightarrow A)){:}thm$, but if the head of *proof?* is the sequent $\vdash (A \Rightarrow B){:}thm$ then calling *(rewrite 0 d-imp proof?)* will return *proof?* unchanged.

Refinements are defined using the interactive theory type. Basically a refinement is a sequent calculus rule. In addition to the straightforward coding of sequent calculus rules, additional capabilities for naming refinement rules, specifying parameters for use in refinement rules, and placing side-conditions on the use of refinement rules are included.

Thus, to encode the sequent calculus rule

$$\frac{f(a), \Gamma \vdash \Delta}{\exists x.f(x), \Gamma \vdash \Delta} \text{ where a does not appear in } \Gamma \text{ or } \Delta,$$

as a named refinement function *exists*, the following **SEQUEL** code could be used in an interactive theory definition:

[3]or *(integer α (t-expr α $\rightarrow$ t-expr) proof $\rightarrow$ proof)* where the appropriate rewrite requires an extra argument

[4]0 indicates the goal *t-expr* and the assumptions are numbered from the head of the list — if the *integer* is greater than the number of assumptions then *rewrite* will act as the identity function.

```
:name exists
:parameter (a? term)
:side-condition (free-in a? <A> delta?)
(f? a?) * thm, <A> |- delta?  * thm
thus
((some x) (f? x)) * thm, <A> |- delta? * thm
```

Unnamed refinement rules are called by the function *refine* with type

(symbol integer proof → proof);

where the *symbol* is the name of the interactive theory, the *integer* is the rule within that theory to be applied, and the *proof* is the current proof-object. As with logical rewrites, refinements act as an identity function if given unusable input. The type of the parameter *a?* is defined to be *term* by the `:parameter (a?  term)` line in the theory, where the definition of the type *term* will have been included as part of the definition of the syntax of the framework. On calling *(exists term? proof?)*, where *term?* is an expression of type *term* and *proof?* is of type *proof*, an attempt would be made to unify *(((some x) (f? x)) * thm)* with each hypothesis in the current (head) sequent of *proof?*. If an attempt at unification succeeds, then the side-condition will be checked. If the side condition evaluates to *nil* then the rest of the hypotheses will be likewise tried. If the side-condition evaluates to *t* then the rule will be applied. The order of hypotheses in the current sequent can therefore effect the action of a refinement rule. The hypotheses in the output sequent(s) will be in the order they appear in the sequent(s) before the thus. Thus whichever hypothesis in the input sequent unifies with *(((some x) (f? x) * thm)* will be removed from the list, and the rest of the list bound to <A>. the output sequent will have *((f? term?) * thm)* first, with <A> as the rest of the hypothesis list. The goal *t-expr* will remain unchanged.

Once the basic operations allowed in the logic are defined with *theory* and *defrew*, a framework can be used as a tool for proving theorems. This is only the first step. If logical rewrites and refinements were all that were available then proof within any system thus defined would be painfully slow, and only really of use for demonstrating how to perform step-by-step proofs in the logic. To define a useful PA or ATP, it is necessary to define the *pragmatics* of the logic: a definition of which rules to apply in which order to successfully prove a theorem. Following previous usage [PHH87], Tarver has called such pragmatics *tactics* — basically a **SEQUEL** function that has type

(α β ... proof → proof).

Tactics may be defined using *define* or *deftactic*, the latter indicating that the tactic is to be made available to the user of the framework. During development of a system, hierarchies of tactics will emerge, with the first tactics using only the logical rewrites and refinements, but with subsequent generations of tactics calling those tactics already defined. Thus a pyramid structure develops with the end result being either a fully automated theorem prover, or a proof assistant with a small set of recommended operations available. It should be noted here that while a proof-object may be decomposed to test how or whether to apply logical operations, re-construction of a proof-object as the result of a rewrite rule in a tactic definition may invalidate the tactic. (An exact description of how this may be allowed is given in [Tar93b], but is beyond the scope of this document.)

2.2.5 Proving Theorems

Once all the parts of a framework have been loaded into a **SEQUEL** session, the *proof tool* is invoked. The *proof tool* is the **SEQUEL** theorem proving environment. In complete installations an X-windows based GUI is available to aid use, but operation is basically the same with or without the GUI. After issuing the command *(prooftool)*, the user is requested to enter a sequence of *t-exprs* (the hypotheses), terminated by typing `ok`, followed by a final *t-expr* (the goal). Thus to give a framework for propositional calculus the sequent

$$(\neg\ (A\ \wedge\ B))\colon thm\ \vdash\ ((\neg\ A)\ \vee\ (\neg\ B)))\colon thm,$$

the following extract from a **SEQUEL** session would be appropriate:[5]

```
(15+) (prooftool)

SEQUEL Proof Tool


ASSUMPTIONS:-


1> ((~ (A & B)) * thm)
2> ok


CONCLUSION:-


?- (((~ A) V (~ B)) * thm)
```

The resulting output would then be:

```
==================================================
Step 1 [1]


?- (((~ a) v (~ b)) * thm)


1. ((~ (a & b)) * thm)


FRAMEWORK>>
```

To the prompt of **FRAMEWORK>>**,[6] the user would input, via the screen or via the GUI as appropriate, the tactics or logical rewrite rules they wished to invoke. A sample session for a

[5]The `(15+)` is the **SEQUEL** prompt. The 15 indicates that this is the sixteenth command given during the session, while the + indicates that type-checking of the top-level is switched on.

[6]The actual prompt will be whatever the framework was named by the programmer, such as **INDUCT>>** or **TABII>>**.

proof within a propositional calculus framework is in Appendix C, together with the equivalent sequent calculus.

In addition to the tactics programmed into the framework, there are a number of system options available. These are:

Abort: Drop out of the *proof tool* and return 'no' to indicate that the proof has failed.

Back n: Backtrack n steps in the proof.

Lemma: When given a *t-expr l*, lemma obeys the following sequent rule:

$$\frac{\Delta \vdash l \qquad l, \Delta \vdash \Gamma}{\Delta \vdash \Gamma}$$

Read: Brings up a pop-up window to allow direct input to the forming tactic command (when using the GUI).

Rotate n m: Switch the nth and mth sequents in the current proof.

Swap n m: Exchange the nth and mth assumptions in the current sequent.

Thin n: Delete the nth assumption from the current sequent.

Undo: Re-input a command from scratch ignoring the current partial input (when using the GUI).

Xtt: Invokes Xtt on the current sequent, solving it if any of the assumptions will unify with the goal.

2.2.6 Derived Rules

Once a theorem has been proved, **SEQUEL** provides a mechanism for adding an appropriate tactic to the framework to avoid having to go through the same stages again. The following example is taken from [Tar93b].

Say we have proved that $p,\ q\ \vdash\ (p\ \wedge\ q)$. If, immediately after having proved this, we issue the command *(mk-dr and-intro nil)* then and-intro will be the rule:

$$\frac{\vdash}{p,\ q,\ \Delta \vdash (p \wedge q)}$$

If we issue the command *(mk-dr and-intro t)* then and-intro will be the rule:

$$\frac{\Delta \vdash p \qquad \Delta \vdash q}{\Delta \vdash (p \wedge q)}$$

The latter is the rule we wish to create here, where each of the assumptions from the proof becomes a new subgoal.

When producing such derived rules, **SEQUEL** generalises all expressions other than those declared to be constants (here the symbol for "logical and" would have been declared as a constant).

Chapter 3

An Overview of nqthm

In 1979, the book A Computational Logic ([BM79]) was published, describing a theory for proving properties of pure Lisp programs (i.e. Lisp programs not involving assignment statements or destructive functions) by induction. Since then, much further work has been done on this topic by Boyer and Moore and others. There are now a number of theorem provers based on this theory, including **nqthm**, which is the current implementation by Boyer and Moore. **INDUCT** was based mainly on the theory set out in [BM79], with some extensions from [Ste88] and elsewhere. This chapter will present an overview of the theory from [BM79].

3.1 Syntax and System Functions

Due to its basis as a system for proving properties of Lisp programs, **nqthm** has an unusual syntax for a theorem prover. The theorems that are proved are simply Lisp terms that have been evaluated as always equal to true, whatever the value of any variables they might contain. Expressions are all function calls or abbreviations for commonly used expressions, using Lisp's Prefix notation. So, *0* is an abbreviation for the constant *(zero)*, *1* is an abbreviation for the expression *(add1 (zero))*, *t* is an abbreviation of *(true)*, etc. Other examples of expressions are *(plus x 3)*, *(cons x nil)* and *(or x (and (equal 1 y) t))*.

We start with a system in which the only objects defined are the boolean values *t* and *f*, and the only functions are *equal* and *if*:

$$(equal\ x\ y) = \begin{cases} t & \text{if } x = y \\ f & \text{if } x \neq y \end{cases} \qquad (if\ x\ y\ z) = \begin{cases} y & \text{if } x = t \\ z & \text{if } x = f \end{cases}$$

Next, the logical connectives *and, or, not* and *implies* are all defined in terms of *if*. For example, *or* is defined as: *(or x y) =(if x t y)*.

We also need objects other than *t* and *f* to work with. We define the *shell* representing natural numbers in this way:

- The bottom object is *0*. This is stated to be a number.
- *Add1* is the function used to construct new instances of numbers, and is called a constructor function. Thus *(add1 x)* is a number for all values of x. If x is not a number, x is treated

as *0*, so for example *(add1 t)* is evaluated to *1*.

- Numberp is a recogniser for integers, returning *t* for objects which are numbers and *f* for objects which are not numbers.
- *Sub1* is the inverse operation of *add1*, and is hence called the destructor function for numbers. If a non-number or *0* is passed to *sub1* the result is the default value of *0*.
- *Count* is defined as:
  $$(count\ x) = (if\ (numberp\ x)\ x\ 0)$$
- *Lessp* is defined as:
  $$(lessp\ x\ y) = (if\ (equal\ y\ 0)\ f\ (if\ (equal\ x\ 0)\ t\ (lessp\ (sub1\ x)\ (sub1\ y))))$$
- A number of axioms are also added to the system. Rather than list them here, these axioms will be stated when their use is explored.

A theorem stating that $\forall x{:}\mathbf{N}(x+1) > x$ is input as a top-level theorem (or *lemma*) to be proved:

$$(implies\ (numberp\ x)\ (lessp\ x\ (add1\ x))),$$

with the assumed meaning

$$(implies\ (numberp\ x)\ (lessp\ x\ (add1\ x))) = t.$$

The internal representation of the current conjecture is a disjunction of literals — called, as usual, a clause. The conjecture *(if a b c)* is represented internally as the two clauses {*(not a)*, *b*} and {*a*, *c*}, both of which must be proved true for the original if-expression to be proved true. A clause has been proved when any of the literals in it are reduced to *t*. A literal that has been reduced to *f* is removed from the clause. If any of the clauses produced in trying to prove a conjecture are reduced to the empty clause, then the proof attempt has failed. (Failure of a proof attempt does not mean that the conjecture is false, since we may have performed an over-generalisation during the proof attempt.) An embedded if-expression within a literal will be distributed out, until the literal can be split. Thus, if we are trying to prove *(foo (if x y z))*, this is equivalent to trying to prove *(if x (foo y) (foo z))*. So, instead of the clause {*(foo (if x y z))*}, we will have the two clauses {*(not x)*, *(foo y)*} and {*x*, *(foo z)*}. Similarly to **nqthm** and [BM79], in this chapter a statement of the current conjecture to be proved will be in the form *(implies (not p) q)* instead of the clause {*p*, *q*}, to improve readability. An explanation of why this internal notation is desirable can be found on [BM79, pp88-89].

Rewriting of an expression is done using an appropriate system axiom. Such axioms are of the form *expr* or *(equal* $expr_1$ $expr_2$*)*. These axioms may involve *variables* (any symbol not defined as a constant), in which case the axiom will apply if the expression being rewritten pattern matches with the axiom. Axioms of the form *expr* will be used to rewrite an expression to *t*, while axioms of the form *(equal* $expr_1$ $expr_2$*)* will be used to rewrite an expression matching $expr_1$ to a dereferenced version of $expr_2$. Once theorems (also called *lemmas*) have been proved they may also be used as rules for rewriting expression. Axioms or theorems such as the example above (*(implies (numberp x) (lessp x (add1 x)))*) which are of the form *(implies hyps thm)* are used in a slightly different way, as will be explained later.

At this point it is useful to define the terms *explicit value* and *explicit value template*, which will be used in a number of places further on. All bottom objects are explicit values, as are *t* and *f*. Explicit values can also be formed using constructor functions applied to explicit values, provided the arguments of the constructor function satisfy the shell restrictions. So, *t* and *(add1 0)* are explicit values, but *(add1 t)* is not since *add1* has a shell restriction of *numberp* which *t* does not satisfy. An explicit value template is a non-variable expression formed using only shell constructor functions, bottom objects and variables. Thus *(add1 x)* is an explicit value template.

3.1.1 Function Definition

There are two sorts of function that can be defined — abbreviations such as

(square x) = (times x x)

and recursively defined functions (referred to from this point on as recursive functions).

Any function defined must conform to the definition principle, which requires explanation of some terminology before being stated.

An important definition is that of a *well-founded relation*: a boolean-valued binary function, say $\prec$, with domain $\mathbf{S} \times \mathbf{S}$, where $\mathbf{S} = \{s_1, s_2, \ldots\}$, where $\prec$ is a well-founded relation provided there is no infinite sequence:

$$\ldots s_i \prec s_j \prec s_k.$$

Boyer and Moore do not prove that a particular function is a well-founded relation, but state that *lessp* is such a relation for numbers. The need for well-founded relations will become clear in the section about induction (§3.3.4).

A term is *fn*-free if *fn* does not occur as a function symbol anywhere in it. A term e governs an occurrence of a term s in a term b if s is a sub-term of p in a sub-term *(if e p q)* of b, or if s is a sub-term of q in a sub-term *(if (not e) p q)* of b.

The Definition Principle:[1]

Given a function definition of the form: *(fn* $x_1 \ldots x_n$*)* = **body**, we require:

- *fn* to be a new function symbol of n arguments.
- $x_1 \ldots x_n$ to be distinct variables.
- **body** to contain no variables other than $x_1 \ldots x_n$.
- that if **body** includes any calls of *fn*, then there exists a well-founded relation r and a measure function m of n arguments, such that for each occurrence of a sub-term of **body** of the form *(fn* $y_1 \ldots y_n$*)*, governed by the *fn*-free terms $g_1 \ldots g_n$, it is a theorem that:

  (implies (and $g_1 \ldots g_n$*) (r (m* $y_1 \ldots y_n$*) (m* $x_1 \ldots x_n$*)))*

The first three are of course simple to check, as is the fourth for abbreviations such as *square* above. The only involved part is the final restriction. The reason for this restriction is to enable the automatic production of valid inductions.

[1]Taken from [BM79, pages 44,45]

Induction in this system is of a generalised form developed from *structural induction.* To illustrate this, we first define the function *plus* over the natural numbers

(plus x y) = (if (zerop x) (fix y) (add1 (plus (sub1 x) y)))

where *fix* is defined as

(fix x) = (if (numberp x) x 0)

and *zerop* is defined as

(zerop x) = (or (equal x 0) (not (numberp x)))

Since the only recursive call in the definition of *plus* is *(plus (sub1 x) y)*, which is governed by the expression *(not (zerop x))*, the definition principle requires that we have a measure function *m* and a well-founded relation *r* such that

(implies (not (zerop x)) (r (m (sub1 x) y) (m x y))).

So, given that it has been stated that *lessp* is a well-founded relation, if we define *m* as:

(m x y) = (count x)

then we have to prove

(implies (and (not (equal x 0)) (numberp x)) (lessp (count (sub1 x)) (count x)))

This is one of the axioms introduced by the definition of numbers, and is called an *induction lemma.* For each new shell, induction lemmas of the form:

(implies (and (not (equal x btm)) (of-type-p x)) (lessp (count (d_i x)) (count x)))

will be introduced, where *btm* is the bottom object of the shell (*0* for numbers), *of-type-p* is the shell identifier (*numberp* for numbers) and the d_i are the destructor function(s) (*sub1* for numbers).

The principle of structural induction states that, if we are trying to prove a conjecture *(P x)* (where *P* is a higher-order template representing **any** conjecture involving *x* and may involve other variables) we can prove *(P x)* provided we can prove:

(and (implies (B x) (P x))
    (implies (and (not (B x)) (P (d x)))
        (P x)))

where *B* is a higher-order template, and we have proved that

(implies (not (B x)) (r (m (d x)) (m x)))

where *r* is a well-founded relation and *m* and *d* are unary functions.

The definition of *plus* will be accepted by **nqthm** and the following induction will be stored as sound for use with conjectures involving calls of *plus*:

(and (implies (equal x 0) (P x))
    (implies (and (not (equal x 0)) (P (sub1 x))) (P x))).

Thus if we are trying to prove the conjecture

(implies (not (zerop y)) (lessp x (plus x y)))

then we can prove this by proving the conjecture

```
(and (implies (equal x 0) (implies (not (zerop y)) (lessp x (plus x y))))
     (implies (and (not (equal x 0))
                        (implies (not (zerop y)) (lessp (sub1 x) (plus (sub1 x) y))))
                   (implies (not (zerop y)) (lessp x (plus x y))))).
```

This only shows the case for induction over single variables, which may be easily generalised to account for more variables. A proof that such inductions are sound (subject to certain conditions) can be found on [BM79, pp.188–189].

3.2 An Example Proof

So, we have the above definition of *plus* accepted by **nqthm**. To illustrate the workings of **nqthm** a proof of the commutativity of *plus* as defined above will be shown, followed by a more detailed examination of the separate parts of the proof procedure, using examples from this proof and others. In **nqthm**, commutativity of *plus* is stated as

(equal (plus a b) (plus b a)).

The first part of the proof procedure to be usefully applied to this conjecture is induction. There are two valid induction schemes suggested by the terms in this lemma, one by *(plus a b)* the other by *(plus b a)*. These schemes are instantiations of the schemes above with x replaced by a or b. The schemes are otherwise identical, and there are no reasonable criteria for choosing between them other than randomly. (In fact, they are equally valid, the only difference being the order they occur in the expression.) So, we choose the induction scheme which *inducts upon b*. Thus, in order to prove our lemma, we need only prove the two sub-goals

Sub-Goal 1:

(implies (zerop b) (equal (plus a b) (plus b a)))

and

Sub-Goal 2:

```
(implies (and (not (zerop b)) (equal (plus a (sub1 b)) (plus (sub1 b) a)))
         (equal (plus a b) (plus b a))))
```

(The original conjecture has been replaced by a new one which is the conjunction of these two sub-goals — the sub-goals are the base and induction cases of an induction argument using the structural induction suggested by the definition of *plus*.)

Concentrating first on sub-goal 1: rewriting *(zerop b)* to its definition above, and unfolding *(plus b a)* (and the resulting occurrence of *(fix a)* in the unfolded expression) we get

(implies (or (equal b 0) (not (numberp b)))
(equal (plus a b)
(if (or (numberp b) (equal b 0))
(if (numberp a) a 0)
(add1 (plus (sub1 b) a))))).

The *(or (equal b 0) (not (numberp b)))* expression in the conclusion of the implication is rewritten to *t* since it is the hypothesis, giving us

(implies (or (equal b 0) (not (numberp b)))
(equal (plus a b) (if (numberp a) a 0))).

As described above, the internal representation of this conjecture will be as a set of clauses. The distribution of the if-expression together with manipulation of the conjecture as a set of clauses gives the following four new conjectures:

Sub-Goal 1.1:

(implies (and (not (numberp b)) (numberp a))
(equal (plus a b) a))

Sub-Goal 1.2:

(implies (and (not (numberp b)) (not (numberp a)))
(equal (plus a b) 0))

Sub-Goal 1.3:

(implies (and (equal b 0) (numberp a))
(equal (plus a b) a))

and

Sub-Goal 1.4:

(implies (and (equal b 0) (not (numberp a)))
(equal (plus a b) 0))

Sub-goal 1.2 can be simplified to *t*, as can sub-goal 1.4, merely by unfolding the occurrences of *plus* and simplifying the expressions. Sub-goals 1.1 and 1.3 are more interesting. So, concentrating on sub-goal 1.1 first, the starting point is another induction, this time inducting on *a*, giving us (after simplification) the two conjectures

Sub-Goal 1.1.1:

(implies (and (zerop a) (not (numberp b)) (numberp a))
(equal (plus a b) a))

and

Sub-Goal 1.1.2:

(implies (and (not (zerop a)) (not (numberp b)) (numberp a)
(equal (plus (sub1 a) b) (sub1 a)))
(equal (plus a b) a))

Looking at sub-goal 1.1.1, we can see that *(and (zerop a) (numberp a))* reduces to *(equal a 0)*, and making use of this to rewrite all occurrences of *a* in the conclusion of the conjecture to *0*, we get

(implies (and (not (numberp b)) (equal a 0)) (equal (plus 0 b) 0)))

which, by unfolding *(plus 0 b)* and then the resulting occurrence of *(fix b)*, reduces to *t*. We now turn to sub-goal 1.1.2, which gives, after unfolding *(zerop a)* and *(plus a b)*

(implies (and (numberp a) (not (equal a 0))
(not (numberp b)) (equal (plus (sub1 a) b) (sub1 a)))
(equal (add1 (plus (sub1 a) b)) a)).

Given the hypothesis *(equal (plus (sub1 a) b) (sub1 a))* and the term *(plus (sub1 a) b)* in the conclusion of the conjecture, we replace *(plus (sub1 a) b))* with *(sub1 a)* in the conclusion, and discard the assumption, giving the conjecture

(implies (and (numberp a) (not (equal a 0)) (not (numberp b)))
(equal (add1 (sub1 a)) a)).

Another of the lemmas added when numbers were defined is

(implies (and (numberp x) (not (equal x 0))) (equal (add1 (sub1 x)) x)).

So, given that *(numberp a)* and *(not (equal a 0))* are hypotheses of the conjecture, we can rewrite *(add1 (sub1 a))* in the conclusion to *a*, hence the above conjecture reduces to *t*. We have therefore proved sub-goals 1.1 and 1.2, and hence have solved sub-goal 1. So, we turn to

Sub-Goal 2:

(implies (and (not (zerop b)) (equal (plus a (sub1 b)) (plus (sub1 b) a)))
(equal (plus a b) (plus b a))))

Which is simplified (unfolding calls of *zerop* and *plus*) to

(implies (and (not (equal b 0)) (numberp b))
(equal (plus a b) (add1 (plus a (sub1 b))))).

(Notice the use of the inductive hypothesis *(equal (plus a (sub1 b)) (plus (sub1 b) a))* to replace *(plus a (sub1 b))* by *(plus a (sub1 b) a)* in the conclusion. The equality is then removed from the list of assumptions, for reasons explained in §3.3.8.)

We now have an expression involving a destructor function (*sub1*). When numbers were defined a lemma *(implies (and (numberp x) (not (equal x 0))) (equal (add1 (sub1 x)) x))*

was added, as has been already stated. As well as using this to rewrite *(add1 (sub1 x))* to *x* provided the assumptions are valid, this can be used to replace occurrences of the destructor function *sub1* with the constructor function *add1*. If we have a conjecture *(P x)* which involves a call of *(sub1 x)*, we can prove *(P x)* if we can prove

(and (implies (zerop x) (P x))
(implies (and (not (zerop x)) (numberp z) (equal x (add1 z))) (P x))).

This is called the *elimination of destructor functions* and §3.3.6 will show why this is desirable and valid. We apply this to sub-goal 2 to replace occurrences of *(sub1 b)* with *c* and occurrences of *b* with *(add1 c)*. Since sub-goal 2 includes *(numberp b)* and *(not (equal b 0))* in the assumption of the implication, the *(implies (zerop x) (P x))* part of the substitution immediately reduces to *t*, and we are left with (after some simplification)

(implies (and (numberp (add1 c)) (not (equal (add1 c) 0)))
(equal (add1 (plus a c))
(plus a (add1 c)))).

We now come across two more of the lemmas added with the definition of a new shell, that

(not (equal (const $x_1 \ldots x_n$*) btm))* and *(of-type-p (const x)).*

In the case of numbers this means that

(not (equal (add1 x)) 0) and *(numberp (add1 x))*

are added, so here the assumptions *(not (equal (add1 c) 0))* and *(numberp (add1 c))* are reduced to *t* and discarded, leaving

(equal (add1 (plus c a)) (plus (add1 c) a)).

We perform an induction on *c*, with the induction scheme derived from the definition of *plus*, giving the two new conjectures

Sub-Goal 2.1:

(implies (zerop c) (equal (add1 (plus c a)) (plus (add1 c) a)))

and

Sub-Goal 2.2:

(implies (and (numberp c) (not (zerop a))
(equal (add1 (plus (sub1 a) c)) (plus (sub1 a) (add1 c))))
(equal (add1 (plus a c)) (plus a (add1 c)))).

Since the assumption *(zerop x)* means that *(plus x y)* unfolds to *(fix y)*, and *(zerop (add1 x))* evaluates to *f*, opening up both calls of *plus* in sub-goal 2.1 leaves us with the conjecture

(implies (zerop c) (equal (add1 (fix a)) (add1 (plus (sub1 (add1 c)) a)))).

(sub1 (add1 c)) reduces to 0 under both assumptions *(not (numberp c))* and *(equal c 0)* from the hypothesis, so opening up the second call of *plus*, we get

(equal (add1 (fix a)) (add1 (fix a))),

which obviously reduces to *t*.

Opening up both calls of *plus* in the conclusion of sub-goal 2.2 and simplifying we get

```
(implies (and (numberp c) (not (zerop b))
                (equal (add1 (plus (sub1 a) c)) (plus (sub1 a) (add1 c))))
           (equal (add1 (add1 (plus (sub1 a) c)))
                  (add1 (plus (sub1 a) (add1 c)))))
```

Using the equality hypothesis to replace *(add1 (plus (sub1 a) c))* in the conclusion with *(plus (sub1 a) (add1 c))* gives

```
(implies ...
           (equal (add1 (plus (sub1 a) (add1 c)))
                  (add1 (plus (sub1 a) (add1 c))))).
```

which obviously reduces immediately to *t*, finishing the proof of the last remaining sub-goal and therefore proving the original conjecture.

3.3 The Proof Procedure in Depth

The various phases of the proof procedure demonstrated above (induction, simplification, unfolding, elimination and generalisation) are applied in what Boyer and Moore refer to as a waterfall model. When a conjecture is presented for proof, it is passed through simplification to a fixpoint, followed by unfolding. If the unfolding heuristic makes changes to the conjecture then it is returned to the 'top of the waterfall' again, otherwise it is passed to the induction phase (if the conjecture is proved at any point it passes out of the waterfall, of course). Once a conjecture has been through an induction, the elimination and generalisation (including cross-fertilisation) phases are added to the waterfall so that the order becomes 'simplification, unfolding, elimination, generalisation, induction'. If a conjecture is passed to the induction phase and no appropriate inductions are found then the proof attempt has failed.

3.3.1 Types of Goal

Above we have the proof of the lemma *(equal (plus x y) (plus y x))*. To prove this lemma the command

(prove-lemma com-plus (rewrite) (equal (plus x y) (plus y x)))

was issued. *Prove-lemma* is, obviously, the function used to prove a lemma, **com-plus** is the name you wish to give to the lemma and *(rewrite)* is a list of the ways the lemma is to be used. During proof, **nqthm** provides a running commentary on what it is doing. This commentary was not included above, but as an example, the lemma

(implies (and (numberp x) (not (equal x 0))) (equal (add1 (sub1 x)) x))

is added by the definition of numbers with the name **sub1-elim**, so when it is used by **nqthm** to eliminate *(sub1 a)* in the above proof **nqthm** printed

```
This simplifies, opening up zerop and plus, to the new conjecture:


      (implies (and (not (equal a 0))
                    (numberp a)
                    (equal (plus (sub1 a) b)
                           (plus b (sub1 a))))
               (equal (add1 (plus b (sub1 a)))
                      (plus b a))).


Applying the lemma sub1-elim, replace a by (add1 c) to eliminate
(sub1 a). We  employ the shell restriction lemma noted when sub1
was introduced to restrict the new variable.  This produces the
new conjecture:


      (implies (and (numberp c)
                    (not (equal (add1 c) 0))
                    (equal (plus c b) (plus b c)))
               (equal (add1 (plus b c))
                      (plus b (add1 c)))),
```

The *(rewrite)* argument is a list of symbols indicating how the lemma should be used if it is proved . In this case the lemma is to be used only as a rewrite rule. The forms of lemma are

Rewrite: A rewrite rule has the form *expr*, *(equal* $expr_1$ $expr_2$*)*, *(implies hyps expr)* or *(implies hyps (equal* $expr_1$ $expr_2$*))*. In the case where there is a set of hypotheses, the lemma will be implemented as a conditional rewrite rule (so that the rewrite will only be applied if the hypotheses can be proved first). Where the actual rewrite is of the form *(equal* $expr_1$ $expr_2$*)* then this will be implemented as rewriting any expression which pattern-matches with $expr_1$ to the dereferenced $expr_2$. Where the rewrite is of the form *expr* then any expression which pattern-matches with *expr* will be rewritten to t.

Induction: Induction lemmas are of the form

$$(implies\ hyps\ (r\ (m\ x_1 \ldots x_n)\ (m\ y_1 \ldots y_n)))$$

where

- r is a well-founded relation
- $y_1 \ldots y_n$ are all distinct variables.

These lemmas are used by **nqthm** to help prove that a recursive function definition satisfies the definition principle.

Elimination: Elimination lemmas are of the form

$$(implies\ hyps\ (equal\ lhs\ x))$$

where

- x is a variable
- there is at least one proper sub-term *(d* $v_1 \ldots v_n$*)* of *lhs* with $v_1 \ldots v_n$ all distinct variables and the only variables appearing within the elimination lemma
- x only appears in *lhs* within such *(d* $v_1 \ldots v_n$*)* sub-terms.

Generalisation: On occasion, the current conjecture may be easier to solve by replacing a term that appears more than once by a variable. The new conjecture is a more general version of the original, but may be no longer true. One reason it may no longer be true is that some specific property of the term being generalised has been lost in the generalisation, for instance generalising the term *(sort x)* to *y* will often invalidate a conjecture, but including the assumption *(ordered y)* will usually prevent this. To avoid useful properties of the term generalised being lost, generalisation lemmas encode properties in such a way that they may be re-introduced as extra hypotheses after generalisation. A generalisation lemma will be of the form *(P (fn* $s_1 \ldots s_n$*))* where P is a schema, not simply a function. If we generalise a term

(fn $s_1 \ldots s_n$*)* in *(R (fn* $s_1 \ldots s_n$*))*

to the new variable *z*, then we add the assumption *(P (fn z))*, so that we instead try to prove

(implies (P (fn z)) (R z)).

3.3.2 Rewriting

Most system axioms, as well as proved lemmas, are used as rewrite rules in addition to their use in other ways, although occasionally a lemma written with a specific syntax in order to be useful as an induction, generalisation or elimination lemma is not in a form that is useful as a rewrite rule, so an equivalent lemma may be proved separately to encode the information as a useful rewrite rule. For instance the expressions

(implies (not (zerop x)) (lessp (count (sub1 x)) (count x)))
and *(implies (not (zerop x)) (lessp (sub1 x) x))*

are equivalent since for numbers *(count x)* = *x*. The first is useful as an induction lemma, but the second is more useful as a rewrite rule. As mentioned above, there are four formats for a lemma to be used as a rewrite rule:

- *expr*
- *(equal* $expr_1$ $expr_2$*)*
- *(implies hyps expr)*
- *(implies hyps (equal* $expr_1$ $expr_2$*))*.

In effect, there are two sorts of *rule* — one in which an expression can be rewritten to *t*, and another which contains two expression which can be rewritten to each other. Either sort of rule can include restrictions on when it is applicable (i.e. a set of hypotheses (*hyps*) may have to be proved before it can be used).

3.3.2.1 Use of Rewrite Rules

If a rewrite rule is of the form *expr*, then the only decision to be made is whether or not to rewrite an instance ($expr'$) of it to *t*. Since the aim in proving lemmas is to reduce the conjecture expression to *t*, and since *t* will always be a simpler expression than $expr'$, the rule will always be used.

If the rule is of the form *(equal* $expr_1$ $expr_2$*)*, then there are a number of options as to its use. It may be allowed to only rewrite instances of it to *t*, or it could be used as an equivalence rule for instances of its sub-expressions $expr_1$ and $expr_2$ allowing either or both of the rules $expr_1 \rightarrow expr_2$ and $expr_2 \rightarrow expr_1$ to be used. The use of the rule to rewrite instances of *(equal* $expr_1$ $expr_2$*)* to *t* is subsumed by using only one of the other rules, since the rule rewriting *(equal x x)* to *t* will then be applicable. **nqthm** allows the user control of how the rule is used by only using $expr_1 \rightarrow expr_2$. It is therefore up to the user to state the rules in the most useful form, so for instance the lemma stating the non-utility of more than one application of a sorting function should be proved as

(equal (sort (sort x)) (sort x)) not as *(equal (sort x) (sort (sort x)))*.

If we have a literal *(not l)* in a clause, this will be used to rewrite sub-expressions *l* of other literals to *t*, while a literal *l* is used to rewrite *l* to *f*. If we have a literal *(not (equal* l_1 l_2*))*, this may be used to rewrite a sub-expression l_1 to l_2 or l_2 to l_1.

With rules such as *(equal (plus a b) (plus b a))*, there is of course the danger of continually rewriting a conjecture and never reaching a fixpoint, but cycling through two or more versions of the rule. So, say we are trying to prove *(foo (plus x (add1 y)))*, having proved the commutativity of *plus* already. We can perform the rewrite on this in an infinite loop

$$\begin{aligned}
\textit{(foo (plus x (add1 y)))} &\rightarrow \textit{(foo (plus (add1 y) x))}\\
&\rightarrow \textit{(foo (plus x (add1 y)))}\\
&\rightarrow \textit{(foo (plus (add1 y) x))}\\
&\ \vdots
\end{aligned}$$

To avoid this loop, **nqthm** uses a lexical measure to determine whether a *symmetric* rule such as the commutativity of *plus* should be used. Thus *(plus x (add1 y))* will be rewritten to *(plus (add1 y) x)*, but the reverse operation rewriting *(plus (add1 y) x)* back to *(plus x (add1 y))* will not occur, since the string "(plus(add1y)x)" is lexically less than the string "(plusx(add1y))".

Finally, all function definitions which are merely abbreviations are also used as unconditional rewrite rules. Thus, whenever the term *(square x)* is encountered, it will be rewritten to *(times x x)*.

3.3.2.2 Conditional Rewrite Rules

Many axioms and lemmas are of the form *(implies hyps rule)*, for instance **sub1-elim** is

(implies (not (zerop x)) (equal (add1 (sub1 x)) x)).

Use of these rules is more complicated than use of the equivalent lemmas without conditions, for instance the lemma **com-plus** *(equal (plus a b) (plus b a))*.

We can of course use these rules in a backwards chaining method so that if we are trying to prove *expr′*, an instance of *expr*, and we have a lemma *(implies hyps expr)*, then we can merely replace the conjecture *expr′* with the new conjecture *hyps′* (the appropriate instance of *hyps*). However, this can quite easily lead to infinite backchaining, even through the use of only one lemma, as shown by Boyer and Moore's example:

> If we have proved that *(implies (lessp x (1- y)) (lessp x y))*, and have *(lessp x y)* as the current conjecture then we might perform the infinite loop:
>
> *(lessp x y)* → *(lessp x (1- y))*
> → *(lessp x (1- (1- y)))*
> → *(lessp x (1- (1- (1- y))))*
> ⋰

To fully explain the usage of conditional rewrite lemmas we must return to the clausal representation used by **nqthm** internally. Suppose we are trying to prove *(implies hyp g)*, where both *hyp* and *g* are single logical terms (i.e. they are not a sequence of terms joined with *and* or *or*). Suppose we have already proved the lemmas *(implies h g)* and *(implies j h)* where *h* and *j* are single terms. The clausal representation of *(implies hyp g)* will be {*(not hyp), g*}. Thus to prove *(implies hyp g)* it is enough to prove *(implies hyp h)*, the clause {*(not hyp), h*}. This is the first attempt to use a conditional rewrite rule, so we start from scratch with a list containing the single literal, the negation of *h*, and go on to rewrite the clause {*(not hyp), h*}. Unconditional rewrites are used if appropriate, but in this case we assume we come to the lemma *(implies j h)* without changing the clause. So, we now decide that we can prove *(implies hyp h)* if we can prove *(implies hyp j)*, i.e. we move from a clause {*(not hyp), h*} to a clause {*(not hyp), j*}. Before doing this, we need to check that we are not in danger of looping. We have a list with *(not h)* on it. If *j* is on the list, we can assume *j* is true, and the clause has been proved. If *(not j)* is on the list, then we have already tried to prove *j*, and returned to it, so we are looping. We also decide we are looping and give up trying to use the the conditional lemmas if *j* is an *elaboration* of the *atom* of one of terms on the list. (The *atom* of a term *(not p)* is *p*, otherwise the atom of a term *q* is *q*.) *A* is an elaboration of *B* if:

- A is identical to B or
- the complexity of A is greater than or equal to the complexity of B and A is *worse than* B.

A is *worse than B* if

- *A* is a variable that is a proper sub-term of *B* or
- neither *A* nor *B* is a variable and either
  - *A* and *B* have different top-level function symbols, with a sub-term *C* of *A* being *worse than* or identical to *B* or
  - *A* and *B* have the same top-level function symbol, with some argument of *A* being *worse than* the corresponding argument of *B*, no argument of *B* being a variable or

explicit value without the corresponding argument of A also being a variable or explicit value, with no argument of A being worse than the corresponding argument of B.

There is one more problem with conditional rewrite rules — there are useful rules that have variables in the hypotheses which do not appear in the rule, for instance the transitivity of *lessp*:

(implies (and (lessp x y) (lessp y z)) (lessp x z)),

where y only appears in the hypotheses. For rules such as this a fairly weak heuristic is used to decide how to apply them. Say we have the literals *(P x y t)* and *(R x y)*, and have proved

(implies (and (P x y z) (S z)) (R x y))

then we will bind z to t, having already established *(P x y t)*, we then try to establish *(S t)*. Thus a free variable in a hypothesis of a conditional rewrite rule will be bound only to a term which will allow immediate satisfaction of one of the hypotheses in which it occurs. So if we had also had the literal *(S u)* in the conjecture, we would have then tried to establish *(P x y u)* to allow the use of the rewrite *(R x y)* $\rightarrow t$. The weakness of this heuristic is that **nqthm** will not find a binding for the free variable that might be obvious to a human, but that does not meet the exact criterion above. The benefit of this heuristic over, say, attempting all possible bindings for the free variable to expressions appearing in the current conjecture, is the gain in speed. If all possible bindings were attempted each time rewriting occurred, for all conditional rewrites with free variables, then **nqthm** would lose a lot of time trying inappropriate bindings. The weakness of this heuristic will usually not prevent **nqthm** proving a lemma, but it will be a longer, more involved process (using further inductions etc.) than the more 'direct' approach of using a previously proved lemma with free variables.

3.3.3 Typing

Since **nqthm** uses a syntax and parser based on Lisp, expressions such as *(add1 t)*, which would be ill-formed in a strongly typed language, are perfectly admissible. However, type information can be very useful in proving theorems about Lisp functions in this context, and we have already seen that Boyer and Moore introduced type specifiers and type restrictions into the shell definition. This type information is also used implicitly during the rewriting procedures. So far, we have introduced the system axioms and added the shell of natural numbers only.

We define the type $\underline{\text{f}}$ to be the set $\{f\}$, the type $\underline{\text{t}}$ to be the set $\{t\}$, the type $\underline{\text{numberp}}$ to be the set of all objects for which the function *numberp* returns t, and the type $\underline{\text{others}}$ to be the set containing all objects not in these other sets. A *type set* is a set of one or more types, for example $\{\underline{\text{t}}, \underline{\text{f}}\}$, which is the type set usually referred to as boolean — for an expression to have this type set means that the expression should only evaluate to t or f, whatever the value of any variables in the expression. For example the expression *(numberp x)* has type set $\{\underline{\text{t}}, \underline{\text{f}}\}$. The type set universe is the set of all types, at this point universe $= \{\underline{\text{t}}, \underline{\text{f}}, \underline{\text{numberp}}, \underline{\text{others}}\}$.

Calculating the minimal type set for an expression is not possible in general (such an algorithm would be equivalent to a sound and complete proof procedure), so the type set computation returns a superset of the minimal type set.

The use of type sets is to reduce the search space for **nqthm**. For instance, the expression *(numberp (fn x))* can sometimes be reduced to *t* or *f* depending on whether the type set of *(fn x)* contains only numberp or does not contain numberp.

Whenever a new function is defined, its type set is calculated and a lemma added to the system specifying this — for instance the function *plus* always returns a number (this is why the base case of the recursion returns *(fix y)* instead of just y), so when *plus* is accepted as a recursive function the lemma *(numberp (plus x y))* is added. These lemmas are used as *rewrite* and *generalisation* lemmas. For a fuller treatment of this topic, including a description of the type set computation algorithm, see [BM79, Ch. VI].

3.3.4 Induction

The ability to handle proof by induction is the core of Boyer and Moore's system — all the other parts of the proof procedure are merely there to support this method of proof. Almost all proofs in **nqthm** rely at some point on an induction, if only in that a lemma used in the proof was proved by induction. To explain the system of induction in **nqthm** we must first return to the definition principle.

3.3.4.1 The Definition Principle

Recall that to accept a recursive function its definition must satisfy some fairly trivial (to demonstrate) syntactical considerations, and also the restriction that for a definition of *fn:*

$$(fn\ x_1\ \ldots\ x_n) = \mathbf{body},$$

we must show that:

> if **body** includes any calls of *fn*, then there exists a well-founded relation r and a measure function m of n arguments, such that for each occurrence of a sub-term of **body** of the form *(fn* $y_1 \ldots y_n$*)*, governed by the *fn*-free terms $g_1 \ldots g_n$, it is a theorem that *(implies (and* $g_1 \ldots g_n$*) (r (m* $y_1 \ldots y_n$*) (m* $x_1 \ldots x_n$*)))*.

To prove this, the first requirement is to calculate the *machine* of the new recursive function: for a new recursive function *fn*, a list of the recursive calls present in the body of the function, together with a list of the *fn*-free terms governing each call. Say we have the function *ack* defined as Peter's version of Ackerman's function:

```
(ack x y) = (if (zerop x) (add1 y)
                 (if (zerop y) (ack (sub1 x) 1)
                    (ack (sub1 x) (ack x (sub1 y))))).
```

This has machine

Hypotheses	Recursive Arguments
(not (zerop x)) *(zerop y)*	*(sub1 x), 1*
(not (zerop x)) *(not (zerop y))*	*(sub1 x), (ack x (sub1 y))* *x, (sub1 y)*

As has been mentioned above, **nqthm** satisfies the definition principle condition by using previously proved *induction lemmas.* induction lemmas are of the form

(implies hyps (r (m $x_1 \ldots x_n$*) (m* $y_1 \ldots y_n$*)))*

where

- r is a well-founded relation
- $y_1 \ldots y_n$ are all distinct variables.

Note that for a lemma to be useful as an induction lemma it must have the same measure function m in both arguments of the measure function r. The only well-founded relation present in the initial configuration of **nqthm** is lessp. A lemma of the form *(implies h (lessp a b)),* where a is a functional expression returning a number can be used as an induction lemma by translating it into the form *(implies h (lessp (count a) (count b))).*

In the following discussion reference is repeatedly made to the component parts of induction lemmas. To facilitate this, we define the parts of an induction lemma

(implies hyps (r (m $x_1 \ldots x_n$*) (m* $y_1 \ldots y_n$*)))*

as follows

hyps is the induction hypothesis,

r is the relation,

m is the measure,

$x_1 \ldots x_n$ are the x-arguments,

$y_1 \ldots y_n$ are the y-arguments and

(r (m $x_1 \ldots x_n$*) (m* $y_1 \ldots y_n$*)))* is the induction conclusion.

If we have a new function definition with arity n, nqthm forms all the (non-empty) subsets of the arguments. So, for *ack* above we have the sets of arguments:

$\{x\}$, $\{y\}$ and $\{x, y\}$

Each set is then tested with all the induction lemmas whose measure has the same number of arguments as there are members of the set (in the initial configuration of **nqthm** we have only one lemma which has a unary measure (*count*)). This testing follows the following procedure:

Say we have the new unary recursive function *fn* which has machine:

Hypotheses	Recursive Argument
(not (B x)), (not (B (bar x)))	(foo (bar x)

and that we have two induction lemmas:

1.*(implies (not (B y)) (r (m (foo y)) (m y)))* and

2.*(implies (not (B z)) (r (m (bar z)) (m z))).*

We have a single recursive argument and two induction lemmas with unary measure functions. We examine our recursive argument and try and pattern match it with the x-argument in an induction lemma. Thus we try and pattern match *(foo (bar x))* with *(foo y)* and *(bar z),* which gives us a match only for lemma 1 above (with substitution [(bar x)/y]). We therefore have the relation *(r (m (foo (bar x))) (m (bar x)))* provided we can prove that the terms governing the recursive call entail the dereferenced terms in the hypothesis of induction lemma 1. We do not test this yet since we have further to go. The dereferenced y-argument of lemma 1 is now *(bar x)*, which is not the formal argument (recall that we are trying to prove that there is a relation/measure pair for which the recursive argument is less than the formal argument). We therefore try and pattern match the *(bar x)* expression with the x-argument in an induction lemma, this time finding that we can match it with induction lemma 2 (with substitution [x/z]). This gives us the relation *(r (m (bar x)) (m x))*, where the y-argument of the dereferenced lemma is now equal to the formal argument x. So, we have a chain of relations linking the recursive argument *(foo (bar x))* to the formal argument x. We now test each link in the chain to see if it holds, by checking that the governing terms from the machine entail the dereferenced hypotheses in the lemmas. Here this means that we must check the two conjectures

(implies (and (not (B x)) (not (B (bar x)))) (not (B (bar x)))) and

(implies (and (not (B x)) (not (B (bar x)))) (not (B x))).

In this case these conjectures are trivial to prove (sometimes more complicated theorem proving is required to show that the relation holds but often it will be of this trivial form). Since we have established the two relations *(r (m (foo (bar x))) (m (bar x)))* and *(r (m (bar x)) (m x))* we can state that the relation *(r (m (foo (bar x))) (m x))* holds provided r is transitive. In the usual case, r will be *lessp*, which is transitive, and Boyer and Moore state on [BM79, p.182] that from any well-founded relation we can construct a transitive version by taking its transitive closure. The conditions on this relation are *(not (B x))* and *(not (B (bar x)))* (in this case there are no governing terms in the machine that are not required — in general this may not be the case). Here we have only used two links in a transitive chain. In general we may use more than this. The measure function m and well-founded relation r must be the same at every stage so that we can state by transitivity that

(r (m (R x)) (m x))

where *(R x)* is the template for the recursive argument. All possible sequences that can be found in the above manner are tried. The restriction that the y-arguments of an induction lemma must all be variables prevents an infinite chain being formed — at least one of the x-arguments will be reduced to a structurally simpler form at each step of the chain.

In addition to lemmas (and transitive chains of lemmas) which imply that the measure of the x-argument is 'strictly less than' the measure of the y-argument, we may also have induction lemmas that imply a 'less than or equal to' relation between the measured arguments. Such lemmas typically include a hypothesis

(not (equal (m $x_1 \ldots x_n$*) (m* $y_1 \ldots y_n$*)))*.

We may form such lemmas from each induction lemma which implies a strictly less than relationship by adding the extra hypothesis required, for example the 'strictly less than' induction lemma

(implies (not (bar x)) (r (m (foo x)) (m x)))

may form the basis of a 'less than or equal to' induction lemma

(implies (and (not (equal (m (foo x)) (m x))) (not (bar x))) (r (m (foo x)) (m x))).

When we have an induction lemma whose x-arguments match the subset of the arguments of the recursive call, but the checking of the hypotheses fails we form such a subsidiary lemma and check that.

At the end of this process, we will have a list of lists of induction lemmas (or transitive chains of lemmas) for each (non-empty) sub-set of the arguments, and for each recursive call. Each list will either imply a 'strictly less than' relationship or a 'less than or equal to' relationship between the arguments of the recursive calls and the formal arguments of the function. First we analyse these lists to see if we can form a direct measure function that gives a 'strictly less than' measure on all the recursive calls for some subset of the arguments. Any relation/measure/set of arguments which can be found to work for all recursive calls is stored on its own and not used for the next stage, which is to attempt to produce a *lexicographic* relation — an ordering based on that used to order words in a dictionary (hence the name). If we have a measure m_1 and a relation r_1 for which some subset $z_1 \ldots z_l$ of the formal arguments has the property that on at least one recursive call (say with arguments $a_1 \ldots a_l$)

$$(r_1\ (m_1\ a_1 \ldots a_l)\ (m_1\ z_1 \ldots z_l))$$

and on others (say with arguments $b_1 \ldots b_l$) we have

$$(m_1\ b_1 \ldots b_l) = (m_1\ z_1 \ldots z_l)),$$

and we have another relation r_2 and measure m_2, for some subset $y_1 \ldots y_k$ of the formal arguments, such that for each of the recursive calls for which the relation/measure pair r_1/m_1 is merely non-increasing, we have

$$(r_2\ (m_2\ b_1 \ldots b_k)\ (m_2\ y_1 \ldots y_k)),$$

then we can form the well-founded (lexicographic) relation on the pair

$$< (m_1\ z_1\ \ldots\ z_l), (m_2\ y_1\ \ldots\ y_k) >$$

the measured subset of the arguments of such a relation being the union of the sets $\{z_1 \ldots z_l\}$ and $\{y_1 \ldots y_k\}$. This strategy can of course be extended so that r_1 or r_2 may themselves have been formed as lexicographic measures, say $< r_3, r_4 >$. Thus r_3 need only be decreasing/non-increasing on those calls not dealt with by the r_1 relation, while r_4 needs only be decreasing on those calls not dealt with by either r_1 or r_3.

Finally, we can now look at our example *ack*. We have only the induction lemma provided by the definition of natural numbers, which is unary, so we try this function on each recursive call and for each of the two arguments x and y. Laying the results out in a table following the recursive calls of the machine above we get:

For x	For y	Recursive Arguments
$<$	$-$	*(sub1 x), 1*
$<$	$-$	*(sub1 x), (ack x (sub1 y))*
$\leq$	$<$	x, *(sub1 y)*

Thus if we form the lexicographic measure formed from *lessp* and *lessp*[2] and use the measure *count* on x for all the recursive calls, and on y for the last call only, we have the measured subset $\{x, y\}$ which can be used to satisfy the definition principle for *ack*. The highly complex nature of the analysis going on is to ensure that all possible relation/measure[3] combinations for all possible measured subsets are found. The reasons for doing this are discussed in the next section.

3.3.4.2 Production of Induction Schemes

We have, therefore, a set of hypotheses which justify a measure/relation pair for which a measured subset of the arguments decrease on each call, justifying the recursive function definition under the definition principle. In the case of *ack* we therefore have an amended machine

Hypotheses	Recursive Arguments
(not (zerop x))	*(sub1 x), 1*
(not (zerop x))	*(sub1 x), (ack x (sub1 y))*
(not (zerop y))	*x, (sub1 y)*

with the measured subset $\{x, y\}$. (Note that there are two sets of recursive arguments both of which are governed by the same hypotheses.) The hypotheses in this table are the ones from the induction lemma(s) used to show that the measure decreases according to a relation. This is to prevent hypotheses which are required for the function to operate properly clogging up the induction scheme where they are not really needed. In this case, for instance the hypothesis *(zerop y)* governs the first recursive call in the machine, but is discarded for the induction scheme, since it is not required for the satisfaction of the definition principle. This cleaning up by discarding unneeded assumptions avoids problems later on when we try to form a new induction scheme which replaces two of those suggested by different recursive functions in a conjecture. Since *(zerop y)* is not a required hypothesis, and may interfere with this process, we discard it. (See page 34 for the details.)

From this amended machine, we can produce the base (or degenerate) case of the induction as well. To do this we take each separate set of hypotheses and take their conjunction, then negate that conjunction, and finally take the disjunction of these negated conjunctions. After some manipulation via the rules governing the propositional logical connectives, we are left with a single expression. If this expression is a disjunction, it is equivalent to a separate base case for each disjoint term. For *ack*, we initially get the expression

(or (not (not (zerop x))) (not (and (not (zerop x))) (not (zerop y)))))).

The *(not (not (zerop x)))* term is rewritten to *(zerop x)* immediately, and application of De Morgan's rules then produce

(or (zerop x) (zerop x) (zerop y)),

which of course allows one of the *(zerop x)* terms to be discarded.

[2]The more usual terminology is that the lexicographic relation is induced by *lessp* and *lessp*, but the term induced is too close to induction so is not used to avoid confusion.

[3]For those relations and measures used in induction lemmas previously proved.

So, we have a measure of some subset of the formal arguments that reduces (according to a particular well-founded relation) on each recursive call, and from this we will produce an induction scheme:

(and (implies (or (zerop x) (zerop y)) (P x y))
    (implies (and (not (zerop x)) (P (sub1 x) 1)) (P x y))
    (implies (and (not (zerop x)) (not (zerop y))
            (P (sub1 x) (ack x (sub1 y))) (P x (sub1 y)))
        (P x y)))

This form of induction is the core procedure involved in **nqthm**. A proof that such inductions are sound (under certain instantiation criteria demonstrated below), can be found on [BM79, pp.188–189]. For the remainder of the chapter, induction schemes will be specified in sequent calculus form, to increase clarity. The induction scheme above is:

$$\frac{\begin{array}{c}\{\Gamma,\ (\mathit{or}\ (\mathit{zerop}\ x)\ (\mathit{zerop}\ y)) \vdash (P\ x\ y)\} \\ \Gamma,\ (\mathit{not}\ (\mathit{zerop}\ x)),\ (P\ (\mathit{sub1}\ x)\ 1) \vdash (P\ x\ y) \\ \Gamma,\ (\mathit{not}\ (\mathit{zerop}\ x)),\ (\mathit{not}\ (\mathit{zerop}\ y)),\ (P\ (\mathit{sub1}\ x)\ (\mathit{ack}\ x\ (\mathit{sub1}\ y))),\ (P\ x\ (\mathit{sub1}\ y)) \vdash (P\ x\ y)\end{array}}{\Gamma \vdash (P\ \mathbf{x}\ y).}$$

(P **x** *y)* can be any expression containing the sub-expression *(plus x y)*,[4] where x is a variable (bold variables in the consequent of the sequent below the line indicate the measured variables[5] — these must be variables for the scheme to be applied to a particular expression). In the sequents above the line:

$$(P\ A\ B) = \begin{cases} (P\ \mathbf{x}\ y)[A/x, B/y] & \text{if } y \text{ is a variable,} \\ (P\ \mathbf{x}\ y)[A/x] & \text{if } y \text{ is not a variable.} \end{cases}$$

The bracketed sequent above the line is the base case. In the rest of this chapter the base case sequent will be omitted since it can be deduced from the induction cases and it does not effect the heuristics being discussed.

3.3.4.3 Production of Appropriate Induction Schemes

Before performing an induction, we have to decide on the form of that induction. Production of valid induction schemes is fairly simple, the complexity coming from the need for useful induction schemes — ones which are likely to prove the conjecture. Our example proof of the commutativity of *plus* contained a number of inductions, none of them involving the more complicated selection heuristics. We will therefore concentrate on different conjectures to illustrate these heuristics. Firstly consider

(implies (and (lessp d e) (lessp e f)) (lessp d f))

The recursive function calls present in this conjecture are:

(lessp d e), (lessp e f) and *(lessp d f)*.

The definition of *lessp* suggests two induction schemes

[4]This is a heuristic restriction to increase the chances of proving the conjecture.

[5]See definition on next page.

$$\frac{\Gamma,\ (not\ (zerop\ x)),\ (P\ (sub1\ x)\ (sub1\ y)) \vdash (P\ x\ y)}{\Gamma \vdash (P\ \mathbf{x}\ y).}$$

$$\frac{\Gamma,\ (not\ (zerop\ y)),\ (P\ (sub1\ x)\ (sub1\ y)) \vdash (P\ x\ y)}{\Gamma \vdash (P\ x\ \mathbf{y}).}$$

The induction schemes suggested by the recursive terms in the conjecture are therefore

$$1\quad \frac{\Gamma,\ (not\ (zerop\ d)),\ (P\ (sub1\ d)\ (sub1\ e)) \vdash (P\ d\ e)}{\Gamma \vdash (P\ \mathbf{d}\ e).}$$

$$2\quad \frac{\Gamma,\ (not\ (zerop\ e)),\ (P\ (sub1\ d)\ (sub1\ e)) \vdash (P\ d\ e)}{\Gamma \vdash (P\ d\ \mathbf{e})}$$

$$3\quad \frac{\Gamma,\ (not\ (zerop\ e)),\ (P\ (sub1\ e)\ (sub1\ f)) \vdash (P\ e\ f)}{\Gamma \vdash (P\ \mathbf{e}\ f).}$$

$$4\quad \frac{\Gamma,\ (not\ (zerop\ f)),\ (P\ (sub1\ e)\ (sub1\ f)) \vdash (P\ e\ f)}{\Gamma \vdash (P\ e\ \mathbf{f})}$$

$$5\quad \frac{\Gamma,\ (not\ (zerop\ d)),\ (P\ (sub1\ d)\ (sub1\ f)) \vdash (P\ d\ f)}{\Gamma \vdash (P\ \mathbf{d}\ f)}$$

$$6\quad \frac{\Gamma,\ (not\ (zerop\ f)),\ (P\ (sub1\ d)\ (sub1\ f)) \vdash (P\ d\ f)}{\Gamma \vdash (P\ d\,\mathbf{f}).}$$

At this stage, we need to define a number of terms pertaining to induction schemes. The variables present in a conjecture are separated into the *changing* variables and the *unchanging* variables — a variable is a member of the *changing* variables if any of the inductive hypotheses include a non-identity substitution. An *unchanging* variable is one which has an identity substitution in all the inductive hypotheses. Sometimes the distinction will be made between a *measured* changing variable and an *unmeasured* changing variable — depending on whether a changing variable is member of the measured subset justifying the scheme or not. Thus in scheme 1 above, d and e are changing variables and there are no unchanging variables. Were we to actually use scheme 1, *fn* would not be substituted for in the scheme, but that is because it does not appear in the expressions suggesting the induction, rather than appearing as a variable which the scheme indicates it is useful to leave unchanged.

Suppose we were to choose one of these schemes at random (there seem to be few other criteria for choosing one), say 3. Ignoring all but the most critical parts of the induction process, this gives us an inductive hypothesis involving the terms

(lessp d (sub1 e)), (lessp (sub1 e) (sub1 f)) and *(lessp d (sub1 f)),*

and a conclusions involving the terms

(lessp d e), (lessp e f) and *(lessp d f).*

Unfolding the terms in the conclusion gives us a conclusion involving

(lessp (sub1 d) (sub1 e)), (lessp (sub1 e) (sub1 f)) and *(lessp (sub1 d) (sub1 f)).*

The term *(lessp (sub1 e) (sub1 f))* appears in both the conclusion and the inductive hypothesis, but the terms *(lessp (sub1 d) (sub1 e))* and *(lessp (sub1 d) (sub1 f))* do not, nor do the expressions before unfolding appear either. The key step in a proof by induction is the use of the inductive hypothesis to *prove* the conclusion, so we therefore aim to have an induction scheme which gives us a conjecture of the form:

$$(implies\ (P\ (d_1\ x_1)\ \ldots\ (d_n\ x_n))\ (R\ (P\ x_1\ \ldots\ x_n)))$$

where we can unfold *(P* x_1 ... x_n*)* to *(P* (d_1 x_1) ... (d_n x_n)*))*, Leaving us with only *(R t)* to prove. This is termed having a reflection of the hypothesis in the conclusion. (A whole philosophy of proof planning for proof by induction has been built up on this idea following [Aub79].)

We therefore wish to somehow create an induction which involves

(lessp (sub1 d) (sub1 e)), *(lessp (sub1 e) (sub1 f))* and *(lessp (sub1 d) (sub1 f))*

in the inductive hypothesis. We can do this by *merging* the schemes shown above. We can merge induction scheme S into scheme P, where the inductive hypotheses for the cases are the substitutions: $s_{i=1...m_s}^{k=1...n_s}(x_{ik})$ and $p_{j=1...m_t}^{l=1...n_t}(y_{jl})$, x_l being the variable substituted for, if:

- the changing variables of P and the changing variables of S have a non-empty intersection,
- the unchanging variables of P have an empty intersection with the changing variables of S,
- the unchanging variables of S have an empty intersection with the changing variables of P,
- we can merge the sets of substitutions such that

  $$(\forall k\ \exists l\ \exists i\ \exists j\ ((s_i^k = p_j^l)\ \wedge\ (x_{ik} = y_{jl})))\ \wedge (\forall k\ \forall l\ \forall j\ \forall k\ ((x_{ik} = y_{jl}) \rightarrow\ (s_i^k = p_j^l)))$$

  where the same value of j or l cannot be used for more than one value of k or i.

The induction scheme resulting from such a merging operation includes the hypotheses and substitutions from each case. The changing and unchanging variables for the new scheme are the unions of the appropriate sets of variables from the original schemes. Identical schemes will of course merge without change. We see here the reasoning behind keeping the set of hypotheses governing each substitution as small as possible. Once two schemes are merged, a hypothesis which was appropriate for a scheme suggested by one recursive function call may be an undesirable property of the term it operates on in terms of the recursive function suggesting the other merged scheme.

If different substitutions for the same variable occur in a case in both schemes, there are three possibilities: the variable is in the measured subset for both schemes, in which case the cases cannot be merged; the variable is in the measured subset for only one of the schemes, in which case the cases may be merged and the substitution for a measured variable will be used for the new scheme; or the variable may not be in the measured subset for either of the two schemes, in which case a random choice is made as to which substitution is used for the new scheme.

In our example, the two schemes 1 and 5 will merge to form the new scheme

$$\frac{\Gamma,\ (not\ (zerop\ d)),\ (P\ (sub1\ d)\ (sub1\ e)\ (sub1\ f)) \vdash (P\ d\ e\ f)}{\Gamma \vdash (P\ \mathbf{d}\ e\ f).}$$

If we start with n induction schemes, each scheme is checked with the other *(n - 1)* schemes for possible mergers. Any scheme which will merge with any of the others is discarded after all possible merged schemes have been calculated. The schemes which could not merge with any others and the new schemes produced from mergers are then passed through this procedure again — we proceed until we reach a fixpoint set of induction schemes. In our example, this results in the single scheme

$$\frac{\begin{array}{c}\Gamma,\ (\mathit{not}\ (\mathit{zerop}\ d)),\ (\mathit{not}\ (\mathit{zerop}\ e)),\\ (\mathit{not}\ (\mathit{zerop}\ f)),\ (P\ (\mathit{sub1}\ d)\ (\mathit{sub1}\ e)\ (\mathit{sub1}\ f))\end{array} \vdash (P\ d\ e\ f)}{\Gamma \vdash (P\ \mathbf{d}\ \mathbf{e}\ \mathbf{f}).}$$

Boyer and Moore comment that when using **nqthm** to slowly develop a theoretical background for proving a theorem (such as proving the lemmas needed for proving the uniqueness of prime factors in number theory), the possible induction schemes will often merge into only one scheme (or into a set of schemes from which their most powerful selection heuristic will choose only one as being highly suitable).

Next we turn to the conjecture

(implies (not (zerop a)) (lessp (half a) a)),

where *half* is defined as

(half x) = (if (zerop x) 0 (if (zerop (sub1 x)) 0 (add1 (half (sub1 (sub1 x))))))

This definition is allowed under the definition principle by using the transitivity of *lessp* and proving that *(sub1 (sub1 x))* (< *(sub1 x)*) < *x*. (See [BM79, pp.181–182] for details.) The induction scheme suggested by its definition is

$$\frac{\Gamma,\ (\mathit{not}\ (\mathit{zerop}\ x)),\ (\mathit{not}\ (\mathit{zerop}\ (\mathit{sub1}\ x))),\ (P\ (\mathit{sub1}\ (\mathit{sub1}\ x))) \vdash (P\ x)}{\Gamma \vdash (P\ \mathbf{x})}$$

so the induction schemes suggested by recursive calls in the conjecture are

$$1\quad \frac{\Gamma,\ (\mathit{not}\ (\mathit{zerop}\ (\mathit{half}\ a))),\ (P\ (\mathit{sub1}\ (\mathit{half}\ a))\ (\mathit{sub1}\ a)) \vdash (P\ (\mathit{half}\ a))}{\Gamma \vdash (P\ a\ (\mathbf{half}\ \mathbf{a}))}$$

$$2\quad \frac{\Gamma,\ (\mathit{not}\ (\mathit{zerop}\ a)),\ (P\ (\mathit{sub1}\ (\mathit{half}\ a))\ (\mathit{sub1}\ a)) \vdash (P\ (\mathit{half}\ a)\ a)}{\Gamma \vdash (P\ (\mathit{half}\ a)\ \mathbf{a})}$$

$$3\quad \frac{\Gamma,\ (\mathit{not}\ (\mathit{zerop}\ a)),\ (\mathit{not}\ (\mathit{zerop}\ (\mathit{sub1}\ a))),\ (P\ (\mathit{sub1}\ (\mathit{sub1}\ a))) \vdash (P\ a)}{\Gamma \vdash (P\ \mathbf{a}).}$$

Schemes 1 and 2 are not completely valid, however, since they include substitutions for non-variables. We may discard a substitution provided it is not one of the measured terms, so we can discard the substitution of *(sub1 (half a))* for *(half a)* in 2, but not in 1. Since 1 involves a non-variable term in a measured position, the whole scheme must be discarded, leaving us with scheme 3 unchanged and the new scheme 2

$$\frac{\Gamma,\ (\mathit{not}\ (\mathit{zerop}\ a)),\ (P\ (\mathit{sub1}\ a)) \vdash (P\ a)}{\Gamma \vdash (P\ \mathbf{a}).}$$

(We can discard the unmeasured substitution from the scheme and still maintain validity because only the measured terms are required. The unmeasured term substitutions are merely present to increase the reflection effect noted above as being desirable.) Examining the two schemes, we can see that 3 appears to be a *repeated form* of the first, in that the substitution from 2, applied twice, is equivalent to the substitution of 3, and the hypotheses of 3 are the hypothesis of 2 and that hypothesis with the substitution from 2 applied:

$(a[\mathit{(sub1\ a)}/a])[\mathit{(sub1\ a)}/a] = \mathit{(sub1\ (sub1\ a))}$ and

$$(not\ (zerop\ a))[(sub1\ a)/a] = (not\ (zerop\ (sub1\ a))).$$

In cases such as this we say that the second scheme has *subsumed* the first (this is Boyer and Moore's terminology), and is therefore the only scheme we are left with. The term subsumed can be slightly confusing, since the normal meaning of subsumption is that a more general term (conjecture, clause etc.) subsumes a particular instance of it. Here, the more constrained, less general term subsumes the more general term. From experience gained proving theorems by induction, Boyer and Moore decided that better reflections were produced by using the scheme involving the repeated scheme once rather than trying to use the non-repeated scheme.

Once we have gathered the induction schemes, decided which are valid, and performed all the merging and subsumption that is possible, we may still be left with more than one induction scheme. There are four heuristics remaining for choosing between these. Firstly we compare the schemes, to see if they interfere with each other. Schemes with no overlap will probably be applicable sequentially, so whichever is applied first, the other will be a candidate for further inductions if necessary. Overlapping schemes, however, may have problems, so the *flaw* detection heuristic is applied first (a scheme is flawed if its application would invalidate the application of another candidate at a later point in the proof). Any unflawed scheme is preferable over a flawed scheme. A scheme S is said to be *flawed* with respect to a scheme T if a changing variable, say v, of S is a changing or unchanging variable of T. If v is a changing variable of T, then S must substitute differently for some common variable (not necessarily v), otherwise the schemes S and T would have merged. To demonstrate why a changing variable in one scheme being an unchanging variable in another causes problems, we turn to the proof that plus is associative for an example. The conjecture stating the associativity of *plus* is

$$(equal\ (plus\ a\ (plus\ b\ c))\ (plus\ (plus\ a\ b)\ c)).$$

After validation, merging and subsumption, we are left with the two schemes

$$1 \quad \frac{\Gamma,\ (not\ (zerop\ a)),\ (P\ (sub1\ a)\ b\ c) \vdash (P\ a\ b\ c)}{\Gamma \vdash (P\ \mathbf{a}\ b\ c)}$$

$$2 \quad \frac{\Gamma,\ (not\ (zerop\ b)),\ (P\ (sub1\ b)\ c) \vdash (P\ b\ c)}{\Gamma \vdash (P\ \mathbf{b}\ c).}$$

Scheme 2 is flawed with respect to scheme 1 since the unchanging variable b in scheme 1 is a changing variable in scheme 2. Say we were to use scheme 2, we would then have an inductive hypothesis involving the terms *(plus a (sub1 b))* and *(plus (sub1 b) c)* and the terms *(plus a b)* and *(plus b c)* in the conclusion. The term *(plus b c)* will unfold to a term *(plus (sub1 b) c)*, as we would wish for reflection. However, the term *(plus a b)* will always include b not *(sub1 b)*, so no matter how many times we unfold it we will never achieve a reflection of *(plus a (sub1 b))*. On the other hand, if we were to use scheme 1, we would have no similar problem because a does not appear in any term which unfolds to a recursive call mentioning a directly, terms involving a unfold to terms involving the *(sub1 a)* that we require for reflection.

If we are still left with more than one unflawed candidate induction, or if all the candidate inductions are flawed, we must choose between them. A *score* for each scheme is maintained from the beginning of the process as follows. The initial score of a scheme is the ratio of the number of changing variables over the number of formal arguments. Thus for the scheme suggested by a term *(plus a b)* in the conjecture is $\frac{1}{2}$. If two schemes merge, then the resulting new scheme has the score of both added together, and if scheme S subsumes scheme T the score for scheme T is added to the score for S. The scheme with the highest score is chosen from the remaining set of all flawed or all unflawed schemes. If two or more schemes share the highest score, then the function which includes the most substitutions which are not *primitive-recursive* (unchanging or substituted for by only shell destructor calls on itself — thus *(sub1 x)/x, (sub1 (sub1 x))/x* and *x/x* are all primitive-recursive substitutions, but *(ack x (sub1 y))/y* is not). Finally, if we still have more than one scheme to choose from then the choice is made randomly — usually these schemes will be suggested by symmetric conjectures so there is little difference between the schemes.

3.3.5 Unfolding

The next phase of the proof procedure we will explore is that of unfolding recursive function calls. If this were done without control, it would lead to an infinite loop of unfolding an expression, rewriting the resulting expression and then unfolding the recursive call(s) again ....

We therefore need a heuristic which prevents such infinite progression of unfolding, but allows unfoldings which might be useful (since not unfolding a recursive function call at an appropriate point can prevent a proof being found). There are two heuristics we use to decide whether or not to unfold a particular recursive call.

The first of these is that under some circumstances we can predict that repeated unfolding will result in an expression not involving any *descendant calls* of the function unfolded — a descendant call is one which is introduced by the unfolding. Thus if we have the expression *(plus a (plus b c))* and unfold the outermost *plus*, we get

(if (zerop a) (plus b c) (add1 (plus (sub1 a) (plus b c))))

which includes three occurrences of the function *plus*. Only the occurrence

(plus (sub1 a) (plus b c))

is a descendant call, however, since the other two calls (both of *(plus b c)*) are simply sub-terms of an argument of the call unfolded. The indication we have that unfolding will eliminate descendant calls is that a measured subset (as calculated when the function was defined and determined to obey the definition principle) consists of only explicit values (as defined earlier in the chapter).

For example, if we have the expression *(plus (add1 (add1 0)) a)*, then since $\{x\}$ is a measured subset of the formal arguments of *plus*, $\{x,\ y\}$, we can unfold this (three times), which gives us, after rewriting, *(add1 (add1 a))*, which does not contain any descendant calls of *plus*.

This heuristic is only occasionally useful, however. We need a more powerful heuristic to decide whether unfolding a particular call is useful. We do this by unfolding the expression, and applying the simplification rules of rewriting to the expression, using the full power of the rewriting algorithms (but only on this expression). The resulting expression (say *val*), is then compared with the original function call (say *(fn* s_1 ... s_n*)*) in order to decide which it is better to keep.

The first criterion is that if *val* contains no occurrences of *fn*, then we keep it, since we have simplified away any mention of the recursive function *fn*. There is no way of telling if occurrences of *fn* in *val* are descendant calls or not, so this may occasionally fail when it should succeed, but one of the further heuristics should capture this case and allow the unfolding.

Otherwise, suppose we have m occurrences of *fn* in *val*:

$$(fn\ s_{1,1}\ldots\ s_{n,1})\ \ldots\ (fns_{m,m}\ldots\ s_{n,m})$$

If each recursive call satisfies one of the following criteria then we allow the unfolding:

- each of the arguments $s_{1,i}\ldots\ s_{n,i}$ already occurs in the current conjecture, or
- more of the $s_{1,i}\ldots\ s_{n,i}$ are explicit values than $s_1\ \ldots\ s_n$, or
- the symbolic complexity of some measured subset of $s_{1,i}\ldots\ s_{n,i}$ is less than the symbolic complexity of the same subset of $s_1\ \ldots\ s_n$. An adequate measure of the symbolic complexity of an expression is the number of occurrences of function symbols, with the restriction that the symbolic complexity of *(if x y z)* is the larger of the symbolic complexity of *y* and *z*.

3.3.6 Elimination of Destructor Functions

As was demonstrated in the proof of the commutativity of *plus*, **nqthm** uses a set of lemmas (the *elimination* lemmas) to replace a conjecture in which a certain function symbol occurs with a new conjecture (often exactly equivalent, sometimes more general) in which there are fewer occurrences of that function symbol, but more of another. These lemmas are of the form:

(implies hyps (equal lhs x))

where

- x is a variable,
- there is at least one proper sub-term *(d* $v_1\ldots v_n$*)* of *lhs* where $v_1\ldots v_n$ are all distinct variables and are the only variables which appear in the lemma,
- x only appears in *lhs* within such *(d* $v_1\ldots v_n$*)* sub-terms.

The *d* functions are the destructor functions, and any function present in an elimination lemma as one of these functions is marked as a destructor function (**nqthm** makes use of this information as part of the generalisation heuristic — see §3.3.7). For each new shell defined, an elimination lemma of the form

(implies (and (r x) (not (equal x btm))) (equal (const (d_1 *x) ...(*d_n *x)) x))*

is added. So, as we have seen, the definition of natural numbers involved the addition of the axiom **sub1-elim**:

(implies (and (numberp x) (not (equal x 0))) (equal (add1 (sub1 x)) x))

which is used to remove occurrences of a term *(sub1 y)*, where y is a variable, and replace it in the current conjecture with the term z, where z is a new variable not already occurring in the conjecture, so that where y occurred there is now an occurrence of *(add1 z)*. The syntactic requirements above, and the way the lemma is used (see below) guarantee that the new conjecture is at least a generalisation of the previous conjecture so that *(new-conjecture ⇒ old-conjecture)*. A sub-class of elimination lemmas (which include the shell axiom elimination lemmas) has the property that *(new-conjecture ⇔ old-conjecture)*, meaning that the conjecture after elimination is exactly equivalent to the original conjecture. [BM79, pp.139–141] shows the properties an elimination lemma must have to be such a validity preserving lemma. It is left to the user to decide whether to only prove validity preserving elimination lemmas, or to use more general elimination lemmas. These more general lemmas can sometimes cause a proof to fail since the new conjecture after elimination of a destructor might not be true, even though the original lemma was true.

So, we have an elimination lemma

(implies hyps (equal lhs x))

containing the destructor term *(d x)*. To use it we take the current conjecture, say P, and try to prove the new conjectures

(implies (not hyps) P) and

(implies (and hyps (equal lhs x)) P)

the second of which is generalised by replacing all the appropriate destructor terms by new variables (also adding in extra hypotheses as per the generalisation heuristics), and then using the equality relation *(equal lhs′ x)* (where *lhs′* is the generalised version of *lhs*) to replace all occurrences of *x* in *P*.

For a concrete example we return to the proof that *plus* is commutative. Use of the lemma **sub1-elim** as an elimination lemma (using *(zerop x)* as shorthand for *(or (not (numberp x)) (equal x 0))*), we get the following:

```
(and (implies (zerop x) (P x))
     (implies (and (not (zerop x))
                   (numberp c)
                   (equal x (add1 c)))
              (P x))).
```

At the point in the proof of **com-plus** that this is used we have the conjecture:

```
(implies (and (not (equal a 0)) (numberp a)
              (equal (plus (sub1 a) b) (plus b (sub1 a))))
         (equal (add1 (plus b (sub1 a))) (plus b a))).
```

So, we have the two conjectures which are equivalent to the original:

```
(implies (and (zerop a) (not (equal a 0)) (numberp a)
              (equal (plus (sub1 a) b) (plus b (sub1 a))))
         (equal (add1 (plus b (sub1 a))) (plus b a))).
```

and

```
(implies (and (not (zerop a)) (numberp c) (equal a (add1 c))
              (not (equal a 0)) (numberp a)
              (equal (plus (sub1 a) b)
                     (plus b (sub1 a))))
         (equal (add1 (plus b (sub1 a))) (plus b a))).
```

The first of these has the assumptions *(zerop a)*, *(not (equal a 0))* and *(numberp a)*, which together are incompatible, so this conjecture is automatically true. In the second, we use the equality *(equal a (add1 c))* to replace every occurrence of *a* in the other hypotheses and the conclusion with *(add1 c)*, and then remove this assumption (see §3.3.8 for the general rules about such operations) giving us:

```
(implies (and (not (zerop (add1 c))) (numberp c)
              (not (equal (add1 c) 0)) (numberp (add1 c))
              (equal (plus (sub1 (add1 c)) b)
                     (plus b (sub1 (add1 c)))))
         (equal (add1 (plus b (sub1 (add1 c)))) (plus b (add1 c)))).
```

The assumption *(numberp (add1 c))* is of course immediately proved true by type set manipulation (there is also a system axiom stating this), while the assumption *(not (equal (add1 c) 0))* is immediately proved true by another system axiom; the assumption *(not (zerop (add1 c)))*, being merely a combination of these two lemmas is also discarded, leaving us with:

```
(implies (and (numberp c) (equal (plus c b) (plus b c)))
         (equal (add1 (plus b c)) (plus b (add1 c))))
```

If we look back to the original conjecture before the elimination lemma was used:

```
(implies (and (not (equal a 0)) (numberp a)
              (equal (plus (sub1 a) b) (plus b (sub1 a))))
         (equal (add1 (plus b (sub1 a))) (plus b a))).
```

we can see that effectively *(sub1 a)* has been replaced with *c* and *a* has been replaced with *(add1 c)*, giving us the usual method of mathematical induction, assuming the conjecture for an arbitrary value (*c*), and showing that the conjecture therefore holds for the successor value *(add1 c)*. The conjectures produced by elimination of destructor functions are frequently much easier to prove than the original conjectures, especially if further inductions are required. Moreover, it is fairly common that we are able to perform an elimination following induction. Since this is the case, it might seem sensible to avoid the complex two tier process, and perform constructive inductions instead, while still allowing elimination. However, not all 'destructor' functions have an exact inverse constructor function — in fact any function which is a *many-one* mapping will not have an appropriate inverse.

Finally, we must take care not to involve ourselves in an infinite loop by eliminating a destructor function, which is then re-introduced when we unfold a recursive function, which we proceed to eliminate again .... For example if we have the expression *(lessp (1- x) x)*, and eliminate *(1- x)* replacing it with *y*, we get *(lessp y (1+ y))* which unfolds to *(lessp (1- y) y)*, which is equivalent to where we started. To avoid such loops elimination is never performed on a term involving a variable introduced by a previous elimination, unless we have performed an induction since the previous elimination.

3.3.7 Generalisation

The generalisation heuristic of **nqthm** is a very weak one, and except in very straightforward cases, experienced **nqthm** users will stop the proof when a generalisation occurs in order to analyse the current conjecture and formulate a more appropriate generalised conjecture than **nqthm** is likely to find on its own, prove such a conjecture as a rewrite lemma, and finally re-run the original proof (source [Moo93]).

There are two parts to the generalisation procedure — the identification of terms to generalise, and the actual generalisation, the latter part taking into account previously proved generalisation lemmas. The second part of this heuristic is also used in the elimination of destructor terms, at the point where the destructor terms are generalised to new variables.

There has been much other work done on this subject, which is one of the most difficult but interesting areas of research into proof by induction and [Vad93] presents a good overview of the work. Here, we will concentrate only on the heuristic implemented in **nqthm**.

As an example, we will turn to part of the proof that *plus* (as already defined) is distributable over *times* defined as

(times x y) = (if (zerop x) 0 (plus y (times (sub1 x) y)))

The complete proof of this is fairly long, so only the generalisation will be shown here. During this proof we come to the conjecture

(implies (and (numberp x) (numberp u))
　　　　(equal (plus z (add1 (plus u x))) (add1 (plus z (plus u x))))).

A human looking at this would almost certainly see that this is simply a specific example of the more general conjecture that the successor function (*add1*) applied to the second argument of *plus* can be distributed outside the *plus*. The reason this is not immediately apparent to **nqthm** is because of the way *plus* was defined — as recursing on the first argument.

This conjecture is generalised to

(implies (and (numberp w) (numberp x) (numberp u))
　　　　(equal (plus z (add1 w)) (add1 (plus z w)))).

3.3.7.1 Identification of Terms to Generalise

Again, this heuristic requires consideration of the internal representation of a conjecture in clausal form, rather than as a single expression. An expression *e* is suitable for generalisation if

- it is not a variable, an explicit value or explicit value template,

- its top-level function is not *equal* or a *destructor*[6] function and
- e occurs in two or more literals within the clause, or e occurs within x and y in a literal of the form *(equal x y)* or of the form *(not (equal x y))*.

So, in the above example, we have the clause

$$\{(not\ (numberp\ x)),\ (not\ (number\ u)),$$
$$(equal\ (plus\ z\ (add1\ (plus\ u\ x)))\ (add1\ (plus\ z\ (plus\ u\ x))))\}.$$

The only expression which fits the above criteria is *(plus u x)*, which occurs in both arguments of the equality literal.

3.3.7.2 Generalising Terms

Having identified the terms we wish to generalise (either via the above heuristic or using elimination lemmas), we need to ensure that useful properties of the generalised terms are not lost. In the example above, we are generalising an expression *(plus u x)* to the new variable w. When *plus* was defined, its type set was computed to be {$\underline{\text{numberp}}$}. Since this is a singleton type set we add this type restriction as an extra assumption *(numberp w)*. If the type set had included more than one type, this restriction would not have been added — Boyer and Moore state that adding such type restrictions produces more problems than it solves.

As well as such type restriction we may have proved a generalisation lemma highlighting a property of an expression which may be important. Such generalisation lemmas will be of the form *(r (fn* $x_1 \ldots x_n$*))*, where r is a schema. If a term *(fn* $s_1 \ldots s_n$*)* is identified as suitable for generalisation, then *(r (fn* $s_1 \ldots s_n$*)* is added as an assumption, together with any appropriate type lemmas. For instance, if we have defined the binary function remainder to return the remainder of dividing the second argument by the first, then the following generalisation lemma may be proved

$$(equal\ (lessp\ (remainder\ x\ y)\ y)\ (not\ (zerop\ y))).$$

If we generalise an expression *(remainder a b)* to z, we then add the assumption

$$(equal\ (lessp\ z\ b)\ (not\ (zerop\ b))).$$

This means that if, at a later stage in the proof, we need to prove that *(lessp z b)*, we can prove it indirectly provided we can show that *(not (zerop b))*, the latter being far easier than the former. This is due to the fact that we know z was really a remainder term from a division by b, so z is less than b provided b is a non-zero number. Since there is a useful elimination lemma replacing *remainder* with *plus* and *difference*, the generalisation of *remainder* may occur quite often. Thus, proving generalisation lemmas (such as the above one) for terms which may be eliminated as destructors, is often useful.

Once this generalisation has been performed, we get the new conjecture in our ongoing example

[6]Destructor functions include both those introduced by shell definitions and any function which is 'eliminated' by a previously proven elimination lemma.

(implies (and (numberp w) (numberp x) (numberp u))
    (equal (plus z (add1 w)) (add1 (plus z w)))).

The assumptions *(numberp x)* and *(numberp u)* are obviously now redundant, and can be discarded. It is quite common for this to happen fairly regularly during a proof, particularly after generalisation or elimination of destructor terms, but it is not always obvious whether or not an assumption can be discarded, nor is the process risk-free. (Technically, the removal of a hypothesis is another form of generalisation, and can cause the production of a non-valid conjecture from an initially valid one.) Therefore there is a set of heuristics dealing with what assumptions can be discarded and when it is useful to do so.

3.3.8 Discarding Irrelevant Hypotheses

There are two main sorts of assumption that we wish to eliminate — literals which are divorced from the rest of the clause and some equality assumptions.

The first of these is the case we have above, in the example used to demonstrate generalisation. We were left with the conjecture (in clausal form)

{*(not (numberp w)), (not (numberp x)), (not (numberp u)),*
(equal (plus z (add1 w)) (add1 (plus z w)))}.

The literals *(not (numberp x))* and *(not (numberp u))* are divorced from the rest of the conjecture since both include only variables that only appear in that one literal. If they have not been proved true by the simplification algorithms (rewriting and unfolding), then we assume they are untrue or unprovable and discard them. While the conjecture that results is technically more general than the original there is actually little risk of over-generalising the conjecture during this process.

The other is the case of an equality as one of the assumptions — i.e. the negation of an equality as a literal in the clause. If we are trying to prove an equality relation, then an induction will lead to an equality hypothesis, as in the case of the commutativity of *plus*, where following an induction and simplification, we get the conjecture

(implies (and (not (equal a 0)) (numberp a)
        (equal (plus (sub1 a) b) (plus b (sub1 a))))
    (equal (add1 (plus (sub1 a) b)) (plus b a))).

which in clausal form is

{*(equal a 0), (not (equal (plus (sub1 a) b) (plus b (sub1 a)))),*
(equal (add1 (plus (sub1 a) b)) (plus b a))}.

We could of course leave things as they are, which is not all that useful, or we could use the equality hypothesis but keep the literal containing it. The reason we discard the equality hypothesis after use is to 'clean up' the conjecture so that if a further induction is required, the equality hypothesis does not cloud the issue of finding the correct induction scheme. Since choosing the wrong induction scheme can cause a proof to fail, it is less risky to generalise the conjecture by discarding an equality hypothesis after use than it is to keep a redundant literal. The name given to this process by Boyer and Moore is called cross-fertilisation (see [BM79, Ch. XI]). Suppose we have a conjecture *(implies (equal s' h') (equal s (fn h')))*. We can legitimately substitute s' for h' or substitute h' for s' in all the other literals, but Boyer and Moore found that the most useful heuristic was to substitute s' for h' in each of the other literals, except for equality literals, where the substitution is made only in the second argument of the equality, so even if s contains h' as a sub-term we would still end up with the new conjecture

(equal s (fn s')).

In our example from the commutativity of *plus*, therefore, we use the equality

(equal (plus (sub1 a) b) (plus b (sub1 a))))

to substitute *(plus (sub1 a) b)* for *(plus b (sub1 a))* in *(equal (plus b a) (add1 (plus (sub1 a) b))))*.

3.4 Conclusion

The theory presented here comprises most of the work from [BM79]. Much further work has been done in this area by Boyer and Moore and others (particularly by a group at Edinburgh University working with the Oyster-Clam system based on the same theory, see [Ste88] and [B$^+$89] for example). There are a number of theorem provers based on the original theory presented here, including **nqthm**-1992, the current release of Boyer and Moore's implementation. These systems tend to deal with an untyped object language. As well as allowing for the extension of the ideas presented here into areas such as constructive type theory by integrating **INDUCT** with other **SEQUEL** frameworks (such as that for Thompson's **TT0**, see [Tho91]), an implementation of the theory presented here as a **SEQUEL** framework could explore the advantages of inheriting a strong type system, or even allow the verification or transformation of functions to become an integrated part of such a type system. In the next chapter the groundwork for such a framework is presented. Some of the heuristics detailed above are rationalised or reduced in scope, while others are no longer required due to the type system the framework **INDUCT** inherits from **SEQUEL**.

Chapter 4

An Overview of INDUCT

This chapter deals with the framework developed in **SEQUEL** to perform proof by induction. The theoretical basis for most of this, taken from [BM79], is described in the previous chapter. Some refinements of the basic theory presented by Boyer and Moore have been included (most notably Stevens' rationalisation of the merging algorithm and flawing heuristics from [Ste88]), and not all of the functionality of the system presented in [BM79] has been implemented due to time constraints. The framework is called **INDUCT**, its most notable difference from **nqthm** being the strong typing which it inherits from **SEQUEL**.

4.1 Using INDUCT

Unlike most frameworks, **INDUCT** is not accessed by explicitly calling the *proof tool*. Due to its incremental nature[1], **INDUCT** is called by a function *start-induct* which calls the *proof tool* and sets up an appropriate environment within it. This environment consists of a (mostly) hypothesis-free sequent calculus system. To prove a theorem (which is held in the consequent of the sequent), various tactics are invoked to reduce the expression to *t*. In general, there will only be a single sequent in the current proof, although some of the interactive proof settings produce extra sequents and/or antecedents.

4.1.1 Interactive Settings

SEQUEL frameworks are designed to be interactive. Although **INDUCT** is currently less interactive than most **SEQUEL** frameworks, more interaction is possible than with **nqthm**.

The main use for interaction will probably come with generalisation, although interaction with induction is also available. As stated in §3.3.7, the generalisation heuristic which **nqthm** uses is not all that useful. Experienced users of **nqthm** will frequently break out of a proof when a generalisation occurs, prove an appropriate lemma separately, and then start from scratch with the original problem. Obviously this wastes a lot of time. With **INDUCT**, by setting the global variable **generalisation-interactive** to *t*, the set of suggested generalisations is printed whenever

[1]Each lemma adds to the rule base in a way not provided for by **SEQUEL**'s derived rule method (§2.2.6).

a generalisation is possible and the user is given a number of choices.

1. Allow the proof to continue with all the suggested generalisations being performed.
2. Allow the proof to continue, but choose a (possibly empty) subset of the generalisations to be performed.
3. Stop the proof at that point and enter the interactive proof process.

Having stopped the proof, the user can then prove an appropriate rewrite rule as a secondary lemma and return to the original proof at the point it was stopped. This avoids the problem of having to restart the original proof, which may have been substantially complete. The user may also choose to concentrate proof effort on a sub-expression within the current conjecture by using the *abstract* tactic.

A similar system exists for choice of induction schemes, provided **induction-interactive** is set to *t*. **INDUCT** calculates the possible induction schemes as usual, but instead of applying the selection heuristics, gives the user the choice of performing one of the induction schemes or performing some other action before continuing with the proof. Appendix B contains a sample **INDUCT** session showing the interactive settings.

4.1.2 Tactics

The on-line documentation for the tactics available within **INDUCT** is shown below:

proof-procedure:

This is the heart of the theorem prover, controlling simplification, unfolding of recursive functions, generalisation, elimination of destructor terms and induction. The proof procedure is automatically called by prove-lemma, but if proof of a lemma is paused during interaction, calling this tactic resumes operation on the current goal lemma.

prove-lemma:

This tactic requires two arguments. First is a boolean valued expression which is the goal to be proved true, and second is an indicator of the type of goal. Valid types are rewrite rules (including conditional rewrites), induction lemmas, generalisation lemmas, elimination lemmas and 'none'. Elimination, induction and generalisation lemmas are all used as rewrite rules as well as any other appropriate usage. Goals indicated as 'none' are merely passed through the proof procedure.

abstract:

This tactic is used during interactive proof. It allows a boolean valued sub-expression of the current goal to be abstracted out and worked on with the proof procedure separately from the rest of the goal.

abstraction-complete:

This tactic is used during interactive proof. It replaces the current goal within its parent expression after an abstraction.

abort-lemma:

Removes the current lemma from the stack.

environment:

Allows saving and restoration of the current environment of proven rules etc. Options are 'save filename comments' or 'restore filename nil', where filename is a string and comments is a list of strings which will be placed at the beginning of the environment file. The environment file should not be manually edited.

induct-define:

Given the name of a type-checked **SEQUEL** function, this function will attempt to show that the function is well-founded and if so will add it to the list of legally acceptable **INDUCT** recursive functions.

print-functions:

Will print out a list of the legally acceptable **INDUCT** functions — 'undefined', recursive and non-recursive, in that order.

4.1.2.1 Proof-procedure

This tactic will usually only be called by the user following interaction. It proceeds with the *proof tool* in the normal way following, for example, an extra lemma proved in place of a generalisation.

4.1.2.2 Prove-lemma

This is the tactic which starts the ball rolling with the framework. It requires two arguments: a conjecture to be proved, and a method of using that conjecture once it has been proved. As described in §3.3.7, 3.3.4 and 3.3.6, generalisation, induction and elimination lemmas all have specific forms and are used as rewrite rules in addition to their other functions. If the expression is proved true, the lemma will be used appropriately. In addition, the lemma type 'none' can be used to prove a theorem without adding it to the system.

4.1.2.3 Abstract and Abstraction-complete

It is sometimes helpful to be able to concentrate on one sub-expression instead of on the whole goal. If the sub-expression can be evaluated to true then the user could use prove-lemma to prove this and add the appropriate rewrite rule. However, if the user wishes to concentrate on a sub-expression which is not inherently true, but which can be simplified, then they can use the abstract tactic to work on a sub-expression with the proof-procedure, and then return the resulting expression to its place in the parent expression using abstraction-complete.

The sub-expression to be abstracted should have a result type of *bool*. **INDUCT** will allow expressions with a variable type (e.g. α) to be abstracted but will issue a warning.

The sub-expression is indicated by a list of numbers representing the argument position at each level. Thus

(1) indicates *(foo x)* in *(if (foo x) y z)* and

(1 4 2) indicates *(plus y z)* in *(if (bar t (plus 1 z) nil (plus 3 (plus y z))) nil t)*.

If the abstracted sub-expression evaluates to t at any point, it will automatically be returned to the parent-expression. Abstractions may be carried out recursively. Due to the mechanism of abstraction and the multiple conjectures possible in the stack, the **SEQUEL** primitive function rotate should not be used as this invalidates **INDUCT** — since **INDUCT** stores information about the sequents in the stack in global variables containing a list of values (one for each sequent on the stack), use of the **SEQUEL** primitive *rotate* which changes the order of the sequents on the stack causes the information in the global variables to be incorrect. It is not possible to prevent a user from using the **SEQUEL** primitive tactics, so there is a warning in the documentation for users to avoid using certain of them.

4.1.2.4 Environment

Without a means of saving the set of lemmas proved in a session, **INDUCT** would not be very useful, since each session would start from only the system axioms. The environment tactic allows the user to save the current image and restore a previous one. the files produced for storing the image should not be edited in any way. As well as the name of the file and a switch for save/restore, the environment command accepts a list of strings which will be added as comments to a saved image on the first few lines.

There is one piece of information that is not saved — the **SEQUEL** definitions of functions accepted under the definition principle. These should be added to the file 'induct.defs.rec' in the **INDUCT** framework directory, ensuring they are loaded into each session with the rest of the framework.

4.1.2.5 Induct-define

Once a function definition has been entered into the **SEQUEL** top-level, it must be passed through the definition principle in order to be accepted by **INDUCT** for use in theorems. Induct-define performs this task when given the name of the **SEQUEL** function to be so analysed.

'Undefined functions' should be defined to the **SEQUEL** top-level using induct-defined-undefined, and non-recursive definitions by using induct-define-nr.

4.1.2.6 Print-functions

A list is printed out of the functions 'known' to the system. Only functions that are listed as recursive, non-recursive or undefined functions will be allowed when entering lemmas to be proved.

4.2 Implementation and Theory

A **SEQUEL** proof procedure uses a sequent calculus notation, while [BM79] mostly presents a system based on rewriting, and **nqthm** actually uses a clausal notation. Due to time constraints and the complexity of the ideas involved in coding an inductive framework for **SEQUEL**, the simplest possible internal representation of a conjecture was chosen for **INDUCT**. **INDUCT** therefore uses a hypothesis-free sequent-calculus notation, with a single *type* (*true*) for the *wffs* in the consequent *t-expr*. Internally, the conjecture we are attempting to prove is represented as an *if*-expression — that is, all logical connectives are rewritten to their equivalent *if*-expression at the earliest opportunity. To ease interaction, however, *if*-expressions are translated to their equivalent logical form before being printed. This has produced some less powerful versions of the rewriting techniques used by Boyer and Moore, but has allowed for the concentration of effort on the more in-depth areas of the proof procedure, particularly on the induction heuristics. Continuation of this work should not require a great deal of changes to the code which already exists (see §5 for more details).

SEQUEL is not designed for complete automation of the proof procedure, nor does it have fully developed capabilities to allow functions to use the *proof tool* without human involvement. The framework therefore has been designed to work from within the *proof tool*. A function *start-induct* is used to bring the *proof tool* up. Any lemmas which the user wishes to prove are given to the system as arguments of the function *prove-lemma*. Use of the **SEQUEL** primitive sequent calculus operations is not recommended as extra information is held separately from the current conjecture and this will not be amended during, say, a **SEQUEL** *rotate* operation. Again, a discussion of the problems this would cause in further development of the framework, and suggestions about how to overcome this are presented in the next chapter.

4.2.1 Control Mechanisms

The heuristics to control which process to apply to a conjecture next in the search for a proof is one of the most important parts of the search procedure. Following Boyer and Moore, **INDUCT** uses a waterfall approach: when a conjecture is first presented for proof, it is passed through the simplification procedure to a fixpoint, then the resulting new conjecture is passed to the unfolding procedure. If the unfolding procedure allows the unfolding of a recursive call within the conjecture then the resulting new conjecture is returned to the top of the waterfall again. If the unfolding procedure makes no change to the conjecture, then **INDUCT** will attempt to produce a valid induction. Once an induction has taken place, we add the extra phases of elimination, local generalisation (see §4.2.5.1), distribution of *if* (see §4.2.4) and generalisation after unfolding and before induction.

4.2.2 Type information

INDUCT inherits functions and types from **SEQUEL**. Before a recursive function can be used in a conjecture presented to **INDUCT** for proof, the function must have been defined both to **SEQUEL** and **INDUCT**. First, the function must be defined in **SEQUEL** and be type-checked under XTT. The signature of the function should only involve *lists* and *integers* in arguments where the argument is other than a place-holder. Ideally, only the types *integer*, *(list integer)* and *(list α)* should be used (where α may be any **SEQUEL** anonymous type). Once a function has been type-checked by **SEQUEL**, it must be passed through the **SEQUEL** function *metarewrite* and defined within an **INDUCT** session in order the check that the definition complies with the definition principle. The **SEQUEL** function *metarewrite* produces an auxiliary function used by the function *unfold* to allow the unfolding of recursive functions. The **SEQUEL** *unfold* is not exactly what is required by **INDUCT**, however — certain changes to the resulting expression must be made to keep the expression within the expected norms of **INDUCT**. These alterations include:

- translation of calls of the function *list* to the appropriate calls of *cons* — **INDUCT** does not include the *list* function in its construction of a list structure, expecting functions to have a rigid number of parameters;

- removing optimisations using local assignments (introduced by **SEQUEL** with *fastcode* optimisation — see [Tar93b]);

- translation of calls of *and* and *or* with more than two arguments to nested applications;

- translation of $c\{a/d\}^*r$ to the appropriate nests of the well-typed functions *head* and *tail.*

Basically, these reduce the **SEQUEL** system calls produced by metarewriting to the standard simple functions known to **INDUCT**.

Checking that a recursive function definition satisfies the definition principle is simplified in **INDUCT**, since any measure function which might be used in an induction lemma must also be typed, and this type information reduces the set of induction lemmas which might be used to justify a definition.

During induction, the number of base cases which need to be checked is often reduced, since degenerate cases do not have to be considered. For instance, the definition of *plus*

```
(define plus
  {integer integer -> integer}
  0 y -> y
  x y -> (1+ (plus (1- x) y)))
```

gives us the induction scheme

$$\frac{\Gamma,\ (\mathit{not}\ (\mathit{zerop}\ x)),\ (P\ (\mathit{1\text{-}}\ x)\ y) \vdash (P\ x\ y)}{\Gamma \vdash (P\ \mathbf{x}\ y)}$$

as it does in **nqthm**. In **INDUCT**, however, *zerop* is defined as

```
(define zerop
  x -> (equal x 0))
```

so in **INDUCT**, the single base case for the above induction scheme is

$$(\mathit{implies}\ (\mathit{equal}\ x\ 0)\ (P\ x\ y))$$

instead of having the extra base case:

$$(\mathit{implies}\ (\mathit{not}\ (\mathit{number}\ x))\ (P\ x\ y)).$$

as in **nqthm**.

4.2.3 Simplification

The simplification algorithm contains three parts — the use of unconditional rewrite rules, the use of conditional rewrite rules, and the use of governing terms (the test expressions from *if*-expressions) as rewrite rules for the governed terms. Unconditional rewrite rules are the simplest of these. The only problem here is with symmetric rewrite rules — rules such as *(equal (plus x y) (plus y x))* which can be used in an infinite loop. The solution implemented is similar to that produced by Boyer and Moore, although it fails in one respect. The terms in the conjecture to be rewritten are packed into a string (ignoring list construction), so that the expression *(foo (bar a?) y)* becomes the string “foobara?y”. Say we have a rewrite rule *(foo x y)* $\rightarrow$ *(foo y x)*, and have a conjecture including *(foo (bar a?) y)*. The rewrite rule be used in this case since (for a lexical string measure) “foobara?y” < “fooybara?”. The problem with this comes with such contrived expressions as *(fo oa? b?)* which is equivalent (in terms of the string that it is packed into) to *(foo a? b?)*. While it is possible to imagine (or to deliberately) create a situation where this prevented a proof being found the likelihood of it occurring in the usual course of events is very small.

The use of governing terms as rewrite rules for an expression is implemented side-by-side with the use of unconditional rewrite rules. The main problem associated with this is how to use equality rules. There are three ways a term *(equal* $expr_1$ $expr_2$*)* can be used as a rewriting rule: rewriting *(equal* $expr_1$ $expr_2$*)* and *(equal* $expr_2$ $expr_1$*)* to *t*; rewriting $expr_1$ to $expr_2$; or rewriting $expr_2$ to $expr_1$. (We do not need to worry here about symmetric rewriting since we are using a rule involving specific terms, not binding a generic rule which may then be used with different bindings). The use of a governing term to perform the rewriting of *(equal* $expr_1$ $expr_2$*)* and *(equal* $expr_2$ $expr_1$*)* to *t* is actually not required provided we rewrite $expr_1$ to $expr_2$ or vice-versa, since we will then have an expression *(equal a a)* where *a* is $expr_1$ or $expr_2$. So, we need a decision mechanism for deciding which way to use the equality. We must be careful here with certain sorts of term, specifically terms such as *(equal x (foo x))*, where we could continually rewrite *x* to *(foo x)* and never end, merely nesting further and further calls of *(foo (foo ...(foo x) ...))*. So, the first criterion we use is that if $expr_1$ is a sub-term of $expr_2$ then we use the governing term to perform the rewrite $expr_2 \rightarrow expr_1$ (and of course the reverse if $expr_2$ is a sub-term of $expr_1$). Secondly, if only one of the two terms is a variable that is not contained in the other term, then we rewrite the variable to the other term. (We will see later that due to the interaction between two other heuristics, this occurs quite often and produces useful results.) Thirdly, if $expr_1$ is an explicit value then we will perform the rewrite $expr_2 \rightarrow expr_1$ (and again the reverse is $expr_2$ is an explicit value — we will never find that both expressions are explicit values since only identical explicit values are equal,[2] so that the governing term would have been rewritten to *t* previously). Failing all of the above, we use the governing term to rewrite *(equal* $expr_1$ $expr_2$*)* and *(equal* $expr_2$ $expr_1$*)* to *t* only. (We must also note here that when we have used the *abstract* tactic, any governing terms are stored as sequent assumptions, and are included in the governing terms which we use as rewrite rules at this stage.)

[2]Free data structures are not permitted in **nqthm** or **INDUCT**.

The implementation in **INDUCT** of conditional rewrite rules is weak when compared to the equivalent process in **nqthm**. If we have a conjecture involving the expression *(foo x)*, and the conditional rewrite rule *(implies (bar x) (foo x))* then we rewrite *(foo x)* to an expression

(imp-lemma (foo x) (bar x) t)

and then use unconditional rewrite rules and propagation of governing terms on the hypothesis term *(bar x)*. If we are then left with *(imp-lemma (foo x) t t)*, we return *t*, whereas if we have *(imp-lemma (foo x) e t)*, where *e* is any expression other than *t*, then we return *(foo x)*. If the conditional rewrite rule was of the form *(implies hyps (equal e f))* then we will use this to rewrite *e* to *f* provided we can establish *hyps* in the method above (subject to the same rules governing symmetric unconditional rules above where applicable).

This method is significantly inferior to that used by Boyer and Moore for **nqthm** where chains of conditional rewrite rules may be used, each further rule being used to prove the hypotheses of previous rules (under some restrictions to prohibit infinite loops occurring).

There is one way in which **INDUCT** is superior to **nqthm**, which is in the use of free variables in the hypotheses of conditional rewrite rules. As stated in §3.3.2.2, **nqthm** attempts instantiations for free variables only where an appropriate term exists in the conjecture to which the full hypothesis term containing the free variable may be matched. Given the strong typing of **INDUCT**, we are able to identify all the terms in the current conjecture which are of an appropriate type and attempt to use them to instantiate the free variable. Once we have identified these terms, **INDUCT**'s fast but weak checking for conditional rewrite hypotheses means that these extra possibilities do not slow the framework too much in general, although proving too many conditional rewrite rules with free variables still produces a bottleneck in the proof procedure.

4.2.4 Distribution of *If*

To allow for easier identification of terms for generalisation and local generalisation, calls of *if* are *distributed* to the outside of the current conjecture. This relies on the fact that

(foo (if x y z)) = (if x (foo y) (foo z)).

It is performed after simplification and unfolding since these two processes will often remove calls of *if* before they are distributed, reducing the time taken to find a proof.

4.2.5 Generalisation

The **INDUCT** generalisation heuristics are equivalent to the **nqthm** heuristics, although the use of generalisation lemmas (already minor in **nqthm**) is downplayed even further since many of the lemmas in **nqthm** are type set specifiers which are superseded by the strong typing **INDUCT** inherits from **SEQUEL**. As we have already seen, interaction is also possible during the generalisation heuristic. This was included due to the assertion by Moore (in [Moo93]) that the **nqthm** generalisation heuristic was in general too weak to be very useful. The possibilities for further strengthening of the generalisation heuristic and/or of the interaction available are discussed in the final chapter.

Other than the explicit generalisation heuristic, there is the cross-fertilisation heuristic for the use of equality hypotheses to rewrite terms in the conclusion of an implication expression. The cross-fertilisation heuristic is not fully implemented in **INDUCT** due to the implementation not splitting up top-level conjunctions of sub-goals. A replacement (less powerful) heuristic has been added, called *local generalisation*, which covers the discarding of irrelevant hypotheses and which displays similar results to cross-fertilisation (particularly when the interactions of the local generalisation and generalisation heuristics are considered).

4.2.5.1 Local Generalisation

Local generalisation is the name given to the process of discarding irrelevant governing terms. Some care is required in identifying terms which may be discarded. In **nqthm**, any literal may be discarded from a clause while retaining soundness in the proof. In **INDUCT**, we do not have a syntax of literals, but have a single expression which will usually involve multiple levels of *if*-expressions nested only within the branches (calls of *if* within a test are distributed according to the rule

$$\textit{(if (if test } l_1\ r_1\textit{) } l_2\ r_2\textit{)} = \textit{(if test (if } l_1\ l_2\ r_2\textit{) (if } r_1\ l_2\ r_2\textit{)).}$$

If we have a conjecture

(if test left t) or *(if test t right)*,

we may generalise these to *left* or *right* respectively and still maintain soundness. (Note that the term governing *left* is *test* and the term governing *right* is *(not test)*.) In general any sub-expression of this form may also be locally generalised by discarding the *test*, except where the sub-expression is a negated one, i.e. if, within the top-level expression, there are an odd-number of occurrences of *(not x)*, where x contains the sub-expression to be locally generalised. To give a concrete example, say we have the conjecture

(not (if (equal x 0) (foo x) t)).

If we distribute the *if* from within the *not*, we get the conjecture

(if (equal x 0) (not (foo x)) nil),

which no longer fits our requirement that we are dealing with a conjecture logically equivalent to an expression of the form *(implies test expr)*, which may be soundly generalised to simply *expr*. So, we only locally generalise certain sub-expressions. We still need to decide which to locally generalise — the reasoning behind such generalisation is the same as that for discarding irrelevant literals in **nqthm** (see §3.3.8), and similar criteria are used. Suppose we have the term *test* governing the term *expr* in an appropriate position for local generalisation within the current conjecture, and that *vars* is a list of all the variables occurring in the conjecture except those occurring only in *test*. We may locally generalise and discard *test* where any of the following hold:

- *test* is of the form *(equal e x)* or *(equal x e)*, where x is a variable which does not occur in *expr*;

- *test* is of the form *(not (equal x e))* or *(not (equal e x))*, where e an explicit value and x is a variable which does not occur in *expr* or

- *test* contains variables which are not in *vars* (i.e. there are variables in *test* which occur nowhere else in the current conjecture.

When we consider the interactions of the generalisation and the local generalisation procedures, we must be aware of the order in which they are applied, and of any intervening processes. To re-iterate the waterfall model, after we perform the first induction in a proof, we have the stages simplification, unfolding, elimination, local generalisation, distribution of *if*, generalisation and induction in the waterfall. Thus, once we have performed a generalisation, we will return to the top of the waterfall and perform simplification, unfolding and elimination before we reach local generalisation. Say we are trying to prove the commutativity of *plus* in **INDUCT**. Following an induction (and having simplified the base case to *t* and performed another induction) we will have the conjecture

(implies (equal (plus x y) (plus y x)) (equal (1+ (plus x y)) (plus y (1+ x)))).

The equality *(equal (plus x y) (plus y x))* does not fall into any of the categories for use as a governing term to rewrite one argument to another, so it has no effect as a governing term. Once we reach the generalisation procedure, we will find that *(plus x y)* is an appropriate term for generalisation, occurring in both a hypothesis and the conclusion of an implication. We therefore generalise *(plus x y)* to the new variable *z* giving us the conjecture

(implies (equal z (plus y x)) (equal (1+ z) (plus y (1+ x)))).

This conjecture is passed through the simplification process, at which point we use the equality hypothesis *(equal z (plus y x))* to rewrite all occurrences of *z* in the conclusion to *(plus y x)*. No further manipulation of the conjecture is performed until we reach the local generalisation procedure with

(implies (equal z (plus y x)) (equal (1+ (plus y x)) (plus y (1+ x)))).

The local generalisation procedure identifies the hypothesis *(equal z (plus y x))* as appropriate to discard since it contains a variable equated to another term. We therefore produce the conjecture

(equal (1+ (plus y x)) (plus y (1+ x))),

which is a cleanly stated sub-goal and appropriately stated for proof by induction. If we examine this process closely, we see that if we have an induction hypothesis *(equal s t)* and a conclusion *(equal (f s) t′)*, where t' is a similar expression to t, but is not identical to it,[3] then we will generalise s to a new variable, replace that new variable in the conclusion with t, and locally generalise to discard the induction hypothesis leaving only *(equal (f t) t′)*, which is what we require.

4.2.6 Induction

It is in the induction-related procedures that **INDUCT** differs most widely from **nqthm**. The implementation of the definition principle in **INDUCT** makes a wide use of the type information available, reducing the search space for justifications of a recursive function definition which would suggest induction schemes, but does not include searching for transitive chains of induction lemmas to justify functions. The collection of candidate induction schemes from a conjecture

[3]This situation is extremely common in inductive theorem proving and the rippling tactic [Aub79] and its extensions are based on this observation.

in **INDUCT** is equivalent to the same process in **nqthm** (including as it does the guarantee of soundness for those schemes), however, the candidate schemes are not separated for different branches of a top-level '*and*' in a conjecture. This weakness would be avoided by a fuller implementation of a sequent calculus system for **INDUCT** (see next chapter). The selection of induction schemes from among the candidates suffers from the lack of separation of different sub-goals, although this can be combated by using the interactive setting for induction and the *abstract* tactic, manually separating the subgoals of a conjecture and allowing **INDUCT** to work on them separately by induction (although this approach currently suffers from a lack of flaw identification in the interactive induction scheme selection process). The automatic induction selection criteria only include flaw identification and one simple tie-breaking rule for equally flawed schemes.

4.2.6.1 The Definition Principle

The heuristic controlling the search for ways to satisfy the definition principle is simplified since only measure functions with appropriate input types can be used to justify the function under examination — thus when trying to justify the definition of Peter's version of Ackerman's function, we need only try induction lemmas whose measure functions take numbers as input, and can safely ignore lemmas whose measure function is, for example, *length*. **INDUCT** includes a system for the automatic identification of lexicographic measures, which current versions of **nqthm** do not. However, **INDUCT** does not form transitive chains of measures so that, for instance the lemma

(implies (and (not (zerop x)) (not (zerop (1- x)))) (lessp (ident (1- (1- x))) (ident x)))

(where *ident* is the identity function for numbers) is required to allow the definition of *half*:

```
(define half
  {integer integer -> integer}
  0 -> 0
  1 -> 0
  x -> (1+ (half (1- (1- x)))))
```

In its initial configuration, **INDUCT** only has a single induction lemma available for use with integers. This lemma uses the well-founded relation *lessp* and the measure function *ident* (an identity function) stating that *(1- x)* is less than *x*. The function *ident* with signature *(integer* $\rightarrow$ *integer)* is required for the same reason **nqthm** requires the equivalent function *count* in the equivalent lemma — namely that induction lemmas include the same measure function to be the top-level function call of both arguments of the well-founded relation function call. *Ident* is used in the case of numbers, and *length* in the case of lists in **INDUCT**, rather than using the same function (*count*) for both, since measure functions in **INDUCT** are required to have appropriate non-disjunct types (i.e. the signature of the measure function should not have an input type of the form *(or* α β*)*).

Other than restricting the search pattern to induction lemmas with appropriate types for the measure functions, and given the lack of transitive chain formation, **INDUCT** performs similarly to the strategy described in [BM79] when justifying a recursive function definition. Formation of induction schemes suggested by a recursive function definition also adheres to the standard supplied by Boyer and Moore.

4.2.6.2 Performing Induction

As stated above, the main restrictions on the **INDUCT** framework during the induction process are that the current conjecture may contain separate sub-goals which do not share an appropriate induction, and that selection between equally flawed candidates is done via a very primitive system. Due to the simple syntax chosen for **INDUCT**, the conjecture

$$\textit{(and (lessp (f } x\ y\textit{) } y\textit{) (lessp } x \textit{ (g } x\ y\textit{)))}$$

is not split into the two sub-goals *(lessp (f x y) y)* and *(lessp x (g x y))*. If *(f x y)* and *(g x y)* suggest schemes which can be merged, then this will not cause any problems. If *(f x y)* suggests a scheme

$$\frac{\Gamma,\ (\textit{not}\ (\textit{zerop}\ y)),\ (P\ x\ (\textit{1-}\ y)) \vdash (P\ x\ y)}{\Gamma \vdash (P\ x\ \mathbf{y})}$$

and *(g x y)* suggests the scheme

$$\frac{\Gamma,\ (\textit{not}\ (\textit{zerop}\ x)),\ (P\ (\textit{1-}\ x)\ y) \vdash (P\ x\ y)}{\Gamma \vdash (P\ \mathbf{x}\ y)}$$

then the schemes will not merge, and each is flawed with respect to the other (since x is an induction variable in the second and an unchanging variable in the first, and similarly for y in the reverse fashion). Thus, whichever induction we choose will probably prove one half of the top-level *and*, but leave the other difficult or impossible to prove.

INDUCT's merging heuristics include two refinements of the original heuristic as stated in [BM79], one from [Ste88] (see below), and another which suggested itself during coding — In 3.3.4.3 we examined the merging of induction cases from two schemes, where there was a clash of substitutions. [BM79] suggests that a random choice is made as to which is chosen, since the merged scheme will be less than perfect for one of the expressions suggesting the differing schemes. Such a random choice would appear to be at odds with the very precise nature of the other methods of assessing suitability presented in [BM79]. On reflection, it would appear that if one of the original schemes was better overall than the other then the substitution from this better scheme should be chosen. An even more effective way to avoid this perceived problem is the one chosen for **INDUCT**, which is to produce both possible mergers of the induction schemes. Then, once all the induction selection heuristics have been applied, whichever scheme is more suitable will be chosen at this stage — only if the two schemes are still equally useful will the choice be made at random (although this is unlikely to happen as Boyer and Moore point out that only when some symmetry is at work between the expression suggesting the schemes do such equally 'good' schemes present themselves following all their selection heuristics).

The merging heuristic (following theory put forward by Stevens in [Ste88]) includes subsumption as a special case by allowing mergers between schemes using multiple instances of the same destructor function and merging them to their lowest common multiple of instances (for example a scheme with the substitution *(1- (1- x))/x* will merge with a scheme including the substitution *(1- (1- (1- x)))/x* to form a scheme with the substitution *(1- (1- (1- (1- (1- (1- x))))))/x*, and including the appropriate base cases).

Once the merging has taken place, **INDUCT** includes only a rudimentary selection process compared with **nqthm**. A flaw detector system, based on the principles presented in [BM79] and more fully explained in [Ste88] is implemented — the original implementation by Boyer and Moore of the theory in [BM79] apparently flawed schemes irrespective of the status of the 'flawing variable' as a member/non-member of the measured subset justifying the induction. Once the candidate induction schemes have been separated into flawed and unflawed schemes, only rudimentary selection criteria are applied — any unflawed scheme is preferred to any flawed scheme; if more than one unflawed scheme is available or if there are no unflawed schemes available and there is more than one flawed scheme, then the scheme 'covering' the most expressions is chosen. An induction scheme 'covers' an expression if that scheme was suggested by the expression, or if that scheme was formed by merging the scheme suggested by that expression with other schemes. If more than one scheme is still available then a random choice is made.

Chapter 5

Conclusions and Further Work

This chapter brings the thesis to a close by drawing conclusions about the validity of the approaches taken during the project and the use of **SEQUEL** as an enabling technology for work of this sort, and by examining the implications of these conclusions regarding further development of **INDUCT**. One of the reasons the current implementation uses an unsatisfactory theoretical basis is that emphasis was placed on efficiency too early in the development of the system. (The overheads of iteration using the **SEQUEL** system logical functions of refinement and logical rewriting are quite large when compared with using an auxiliary function to perform all the iterations as a single logical rewriting operation.) The following discussion of how the framework might be developed focuses on the theoretical basis rather than on the efficiency of such an implementation.

5.1 INDUCT vs nqthm

The original aim of the project was to produce something which could be integrated into the **SEQUEL** type-checker, using the theory presented by Boyer and Moore. Unfortunately, [BM79] is not a book which clearly and concisely presents that theory in a generic, easily understood form, and even those papers presenting a rational reconstruction of the heuristics do not always illuminate the subject as they could. Despite its obvious importance, indeed its requirement, as a technique for automatic program verification, (semi-)automatic induction remains an esoteric field, understood completely only by those who have been involved in building or developing such a system. The literature on the subject therefore tends to be couched in terms of the system developed rather than general logical formulations. Therefore, the development of **INDUCT** was diverted from an implementation in a typed sequent calculus system and became an implementation of a slightly rationalised version of the theory presented in [BM79]. On reflection, the original goals of the MSc were set far too high in terms of the resulting implementation. Instead of aiming to produce a system capable of automatic proof of complex conjectures, the time might have been better spent attempting to translate the theoretical basis of **nqthm** into one which might later be implemented in **SEQUEL**, with only minor forays into producing large amounts of working code. The project was hindered on both sides of the equation, since some aspects of

SEQUEL make it less than ideal for this particular sort of framework (see §5.4).

With hindsight, too much time was spent on understanding the Boyer and Moore theory by implementation of an analogue of **nqthm** in **SEQUEL**, rather than concentrating on translation of the early implementations into a more appropriate **SEQUEL** format. As a project, the MSc has produced something of a blind alley in terms of a useful system, although much of the code produced would be useful as a basis for implementing a more rationalised framework for induction.

5.2 A Rational Development of INDUCT

As can be seen from §2, **SEQUEL** takes a new approach to the problem of coding logical frameworks in a form amenable to automation of proof. The sequent calculus based on the proof as type-checking principle is very powerful for certain logical paradigms. **SEQUEL** is ideal, for instance, for implementation of constructive type theory (a basic framework implementing Thompson's $\mathbf{TT}_0$ was produced in only a weekend for example). There are problems with some other implementations, however, when the complexities of the **SEQUEL** syntax do not merge easily with the object syntax of a framework. The object language of Boyer and Moore is untyped Lisp, essentially just functional expressions. The assumed meaning of a conjecture is *conjecture* $= t$ for all values of the variables present. The natural implementation of such a framework is to use explicit typed universal quantification of variables over a conjecture such as stating the commutativity of the function *plus* as

$$\forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (\mathit{equal}\ (\mathit{plus}\ x\ y)\ (\mathit{plus}\ y\ x)){:}\,\mathit{true}$$

However, one of the core reasons that a system such as **nqthm** is useful for proving properties of highly complex programs is the increment hierarchy, where most conjectures are merely building blocks from which a final, useful, conjecture is proved. This approach is well-known to mathematicians, whose work often splits a proof down into numerous lemmas, upon which the short final proof depends, which lemmas are subsequently (or have been previously) proved.

Suppose we have a system including the standard rule for elimination of typed universal quantification in the consequent:

$$\frac{\Gamma,\ a{:}\,\alpha \vdash D[a/x]{:}\,\beta}{\Gamma \vdash \forall x{:}\,\alpha\ D{:}\,\beta} \quad \text{Where } a \text{ does not occur in } D \text{ or } \Gamma.$$

and in the antecedent:

$$\frac{\Gamma \vdash a{:}\,\alpha \qquad \Gamma,\ a{:}\,\alpha,\ D[a/x]{:}\,\beta,\ \forall x{:}\,\alpha\ D{:}\,\beta \vdash \Delta}{\Gamma,\ \forall x{:}\,\alpha\ D{:}\,\beta \vdash \Delta.}$$

Suppose we prove the distributivity of the successor function from the second argument over *plus*:

$$\forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (\mathit{equal}\ (\mathit{plus}\ x\ (\mathit{1+}\ y))\ (\mathit{1+}\ (\mathit{plus}\ x\ y))){:}\,\mathit{true}$$

If we next try to prove that *plus* is commutative:

$$\forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (\mathit{equal}\ (\mathit{plus}\ x\ y)\ (\mathit{plus}\ y\ x)){:}\,\mathit{true}$$

then during this proof we will have the conjecture:

$$a{:}\,\mathbf{N},\ b{:}\,\mathbf{N} \vdash (\mathit{equal}\ (\mathit{plus}\ a\ (\mathit{1+}\ b))\ (\mathit{1+}\ (\mathit{plus}\ a\ b))){:}\,\mathit{true}$$

which is simply an instantiated version of the distributivity property proved before. If we have made the distributivity law above part of the set of assumptions Γ, then we can have the proof of the commutativity of *plus*:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash}{\begin{array}{c}\Gamma,\ (equal\ (plus\ a\ (1{+}\ b))\ (1{+}\ (plus\ a\ b))){:}\,true,\\ \forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (equal\ (plus\ x\ (1{+}\ y))\ (1{+}\ (plus\ x\ y))){:}\,true,\ a{:}\,\mathbf{N},\ b{:}\,\mathbf{N}\\ \vdash (equal\ (plus\ a\ (1{+}\ b))\ (1{+}\ (plus\ a\ b))){:}\,true\end{array}}}{\begin{array}{c}\Gamma,\ \forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (equal\ (plus\ x\ (1{+}\ y))\ (1{+}\ (plus\ x\ y))){:}\,true,\ a{:}\,\mathbf{N},\ b{:}\,\mathbf{N}\\ \vdash (equal\ (plus\ a\ (1{+}\ b))\ (1{+}\ (plus\ a\ b))){:}\,true\end{array}}}{\vdots}}{\begin{array}{c}\Gamma,\ \forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (equal\ (plus\ x\ (1{+}\ y))\ (1{+}\ (plus\ x\ y))){:}\,true\\ \vdash \forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (equal\ (plus\ x\ y)\ (plus\ y\ x)){:}\,true\end{array}}$$

This is a valid proof (given appropriate steps in place of the $\vdots$), but in terms of a readable and automatic system, it is virtually unworkable. There are two main problems. The first comes from the difficulty of using an assumption of the form

$$\forall x{:}\,\mathbf{N}\ \forall y{:}\,\mathbf{N}\ (equal\ (plus\ x\ (1{+}\ y))\ (1{+}\ (plus\ x\ y))){:}\,true$$

to rewrite another *t-expr*

$$(f\ (plus\ x\ (1{+}\ y))){:}\,\beta$$

to the *t-expr*

$$(f\ (1{+}\ (plus\ x\ y))){:}\,\beta.$$

The second problem is that the current sequent would quickly become unwieldy and finally unusable for a human, and extremely inefficient for an ATP.

In its current formulation, **INDUCT** does not have the same problems, since conjectures are implicitly universally quantified and were designed to be included in a **SEQUEL** function which performed rewriting on a *wff* — thus a lemma of the form *(equal* $expr_1$ $expr_2$*)* would be used to rewrite $expr_1$ to $expr_2$, under certain restrictions to prevent infinite looping in the case of symmetric rules (i.e. commutativity rules). There is a separate encoding for using rewrite lemmas of the form

(implies hyps rewrite)

where there are conditions on the rule being applicable (such as one of the variables in the rule being non-zero).

Encoding such rules into a separate function which has type *(wff → wff)* also avoids the problem that **SEQUEL** is designed to manipulate *t-exprs*, not just *wffs*, the two forms of manipulation being refinement (which manipulates *t-exprs* within sequents) and logical rewriting (which manipulates single *t-exprs*). The recursive structure of wffs in **INDUCT** causes problems — we may have the wff *(equal (foo x) (bar x))* in the consequent *t-expr* of the current sequent. It is impossible to define a **SEQUEL** logical rewrite rule from the lemma

(equal (foo x) (foo1 x))

which will cover all possible positions of the expression *(foo x)* within a *t-expr* in the current sequent, even if we use higher-order pattern matching, since *(foo x)* may occur at an arbitrary depth within the *wff* of the *t-expr*. In fact, we may come across instances where we do not wish to apply the rewrite suggested by the equality hypothesis *(equal (foo x) (foo1 x))* to all occurrences of *(foo x)* within the consequent (we make no distinction here between, say, inductive hypotheses which involve equality relations, and lemmas introduced as assumptions) for instance if we wish to perform cross-fertilisation.

The answer to this is to implement the logical equivalent of a structured editor. Fortunately, the theory and practice of this have already been explored by Tarver for a partial evaluator built in **SEQUEL** (see the *MIX* framework available with the **SEQUEL** system files). This allows us to concentrate our efforts on a sub-expression of the *wff* of the consequent, and avoid the manipulations affecting other parts of the consequent *wff*.

Ignoring efficiency for the moment, and assuming that we have implemented equality in some form such as:

$$\frac{\Gamma \vdash expr_2 \colon \beta}{\Gamma,\ (equal\ expr_1\ expr_2)\ \colon true \vdash expr_1 \colon \beta}$$

then we might encode the distributivity law stated above in a tactic which examines the current conjecture for a sub-expression of the form *(plus* e_1 *(1+* e_2*))*, invokes the *distributivity-of-1+-over-plus* rule:

$$\frac{\Gamma, \forall x\mathbf{N}\ \forall y\mathbf{N}\ (equal\ (plus\ x\ (1{+}\ y))\ (1{+}\ (plus\ x\ y))) \vdash D \colon \beta}{\Gamma \vdash D \colon \beta}$$

invokes the rule for elimination of antecedent universal quantifiers twice and then invokes the structured editor to find the appropriate sub-expression(s) in D in order to use the equality rule. In this system, we would need an axiom allowing the introduction of a previously proved lemma as an assumption, in a similar method to the **SEQUEL** primitive *lemma*, but without the need to prove the lemma in a separate sequent.

Such an implementation would lend itself well to the addition of a proof-planning control mechanism along the lines of the Oyster-CLaM system (see [BvHHS91]), rather than to a brute force application of rewriting to a normalised form as was the case for **nqthm** and the current version of **INDUCT**. Major parts of Boyer and Moore's unfolding heuristic are dependent upon a normalised form of the unfolded expression being available, so that a normalising technique would still be required were we to use Boyer and Moore's technique. To a proof-planner implementing rippling, unfolding of a recursive function is merely another type of wave rule so this should not be a problem.

The implementation of the above system in **SEQUEL** would be fairly straightforward, although the animation of these aspects (which are phases of the proof procedure designed to support induction) would be more complex than the current implementation of **INDUCT**. There is still a gap in the support offered by **SEQUEL** for a system such as this. If we prove a subsidiary lemma during the proof of our principle conjecture, then that lemma is available for use (possibly more than once) in that proof. We would like to be able to do two things, however. Once having proved a conjecture, we wish to be able to access it as a system axiom, and it would be useful to add any subsidiary lemmas proved to the system also. If we were to follow the current implementation of **INDUCT**, then all operations would be performed from within the *proof tool*, which certainly allows us to add subsidiary lemmas to the rule base 'on the fly'. However, due to the lack of system provision for storing information other than the current sequent, this might prove difficult to achieve while still allowing access to the full power of **SEQUEL**'s sequent calculus. **SEQUEL** was also not designed to allow the *proof tool* to be used as an auxiliary function of a higher level program. Therefore there is no facility for calling the *proof tool* with a conjecture to be proved and with a tactic to be called upon *proof tool* initialisation. Following a request, Tarver has implemented a partial version of this, *call-prooftool* which initialises the *proof tool* with a supplied sequent (in external syntax). This does not include a tactic name as an argument, however, instead requiring the user to invoke a tactic once the *proof tool* has been initialised.

Together, these omissions make **SEQUEL** difficult to program in its current form for what is envisaged here, without some additions to the system code. The most pressing need would be provision for an automatic calling of the *proof tool* with both an initialising sequent and a tactic to be performed. Such a capability would allow two important developments of the improved **INDUCT** framework. Firstly, a conjecture presented for proof would not need to be stored within the confines of the *proof tool*, allowing information such as the original conjecture and an indicator as to how the conjecture should be used to be stored only as local variables rather than as globals within the *proof tool* environment. Secondly, the automatic use of the *proof tool* would then be facilitated during analysis of new recursive function definitions when attempting to satisfy the definition principle. Currently, such analysis must be carried out by calling a tactic from within the *proof tool*, which is an inelegant solution at best. There remain two problems which would require changes to the system code for the *proof tool*, both of which stem from the lack of any provision within **SEQUEL** for information other than the current sequent to be held linked to, but not part of, the current sequent. While it is possible to define global variables to hold such information, the lack of dynamic system links between the *proof tool* and the global variables prevents full use of the sequent calculus representation inherent in **SEQUEL**. For example, if we wish to hold the same set of information for each sequent in the stack, then we must define a global variable which lists these sets in the appropriate order, and prevent the user performing a **SEQUEL** primitive *rotate*, which changes the order of the sequents on the stack. Automatic use of *rotate* would also require the appropriate rotation of any global variables held. The second circumstance in which we might wish to hold information which is not strictly part of the current sequent is when we prove a subsidiary lemma as part of a proof. Apart from wishing to use such a lemma in future proofs without having to prove it more than once, we might wish to use it within more than one sequent currently on the stack (**SEQUEL** does not provide a mechanism for doing this other than performing the proof of the lemma for each sequent). The latter problem may well be a general criticism of the **SEQUEL** interface. The former problem only occurs in incremental frameworks such as this, where each lemma proved is added to the rule base of the system. One solution is to include in each sequent an extra hypothesis *t-expr*, say as *info * information* where *info* is a *wff* which holds the information we require, such as the original form of a lemma and a flag to indicate how it is to be used if proved. The original solution to this problem was to include such information as part of the consequent *t-expr* and to work around it when manipulating the sequent and remove it by use of **SEQUEL**'s external syntax. Working around such an extra assumption would be relatively simple and although the external syntax mechanism would not allow the *t-expr* to be removed completely before printing, it might be reduced to just an indicator that internal information is stored there. Another possibility for the storage of the original form of a conjecture would be to adopt an *answer-sequent*. Thus instead of the rule lemma:

$$\frac{\Gamma \vdash \mathit{lemma}{:}\,\mathit{true} \qquad \Gamma,\ \mathit{lemma}{:}\,\mathit{true} \vdash \Delta{:}\,\alpha}{\Gamma \vdash \Delta{:}\,\alpha}$$

we would have the rule

$$\frac{\Gamma \vdash \mathit{lemma}{:}\,\mathit{true} \qquad \Gamma,\ \mathit{lemma}{:}\,\mathit{true} \vdash \mathit{lemma}{:}\,\mathit{true} \qquad \Gamma,\ \mathit{lemma}{:}\,\mathit{true} \vdash \Delta{:}\,\alpha}{\Gamma \vdash \Delta{:}\,\alpha.}$$

Once we had proved the $\Gamma \vdash \mathit{lemma}{:}\,\mathit{true}$ sequent, we could then allow the automatic proof of the $\Gamma,\ \mathit{lemma}{:}\,\mathit{true} \vdash \mathit{lemma}{:}\,\mathit{true}$ sequent to prompt the addition of *lemma* as a system rule. However, use of the **SEQUEL** primitive *rotate* to switch the order of the sequents in the stack could invalidate such a move, producing unsound results.

Justification for implementing this proposed system would require a gain in clarity, efficiency or capability for **INDUCT** over **nqthm** and the other systems already implemented. In terms of raw manipulations per second, **INDUCT** could never compete with **nqthm** since it has overheads that cannot be avoided (in terms of its more complex proof structure which is written in a framework for general frameworks, rather than one which is optimised for the particular syntax of **INDUCT**). The more advanced features now implemented in **nqthm**-1992 (see [BM88] and proposed 2nd ed.) would not lend themselves to easy implementation under this system, nor would it repay the effort taken to implement them. However, there are ways in which a **SEQUEL** framework implementing the core ideas of [BM79] would be worthwhile. The inner workings of **nqthm** are contained in approximately 100,000 lines of Lisp, and other implementations are of similar size. **SEQUEL** is much more compact, and yet more easily understood (in my opinion) than either Lisp or Prolog (the likely languages for such implementations). Further development of a **SEQUEL** framework by a new researcher would be much simpler than further development of **nqthm** or other such systems due to the cleaner environment. The **SEQUEL** *proof tool* provides a consistent environment for developing different frameworks, and as such the integration of ideas from differing logics is much simpler than it would be otherwise.

5.3 Constructive Type Theory and $\mathbf{TT}_0$

Take, for example, Thompson's $\mathbf{TT}_0$, a basic implementation of which is already available in **SEQUEL**. $\mathbf{TT}_0$ works by manipulating the *types* in *t-exprs* instead of the *wffs* in order to prove a conjecture. The *wff* develops (usually) from a single variable into an expression defining a proof of the original conjecture. In the case of quantification, the *wff* will be a λ-expression defining an anonymous function which produces a proof of the conjecture. Thus if we wish to prove that $\forall x{:}\,\mathbf{N} \exists y{:}\,\mathbf{N}(<\ x\ y)$ this would be the conjecture

$$\vdash z{:}\,(\forall x{:}\,\mathbf{N}\ \exists y{:}\,\mathbf{N}\ (< xy))$$

and we might get the resulting λ-expression $\lambda x.(\mathit{1}{+}\ x)$ as inhabiting the type.

A full explanation of constructive type theory and typed lambda calculus is beyond the scope of this thesis, but a quick overview will be presented. Using Heyting's Semantics for First Order Logic (FOL) (taken from [Tar95]) we have:

A proof of $p \to q$ is a function *f* that maps a proof of p to a proof of q.

A proof of $p \wedge q$ is a pair $\langle p, q \rangle$ where p is a proof of p and q is a proof of q.

A proof of $p \vee q$ is a pair $\langle i, p \rangle$ where either (a) $i = 0$ and p is a proof of p, or (b) $i = 1$ and p is a proof of q.

A proof of $\neg\ p$ is a proof of $p \to \perp$ where $\perp$ is any absurdity.

A proof of $\forall x{:}\, A\ B$ is a function *f* which, for each member a of the domain A, maps a proof of a to a proof of *B[a/x]*.

A proof of $\exists x{:}\, A\ B$ is a pair $\langle a, p \rangle$ where a is a member of the domain A and p is a proof of $B[a/x]$.

This may look fairly straightforward and not that different from the normal semantic interpretation. Upon closer examination, however, there are some interesting implications of Heyting's Semantics. *The law of the excluded middle* is the first to become apparent. In the standard semantics for FOL, we have the rule $\vdash A \vee (\neg A)$ which means that we must be able to prove A or $\neg A$, whatever A is. However, in intuitionistic logic (as this interpretation is called, as opposed to the classical logic which is the more usual interpretation), this is not the case, since it may be beyond our powers to prove either, and being able to prove $\vdash A \vee (\neg A)$ in intuitionistic logic requires that we have proved at least one of the disjunct terms to be a theorem. There are some other standard theorems of classical logic that do not hold in intuitionistic logic:

$p \vee \neg p$

$\neg(p \wedge q) \to (\neg p \vee \neg q)$

$\neg\forall x{:}\, A\ B \to \exists x{:}\, A\ \neg B$

The last of these is probably the most interesting. Heyting's Semantics state that a proof of $\exists x{:}\, A\ B$ involves production of a proof p that a member $a \in A$ satisfies $B[a/x]$. This differs from classical logic in that there is no provision for indirect proof. To prove that something exists we must produce at least one instance of it, and it is this fact that gives rise to the description of such logics as *constructive* since in proving something exists we *construct* an instance. So, for example to prove that for each natural number there exists at least one natural number greater than it, the proof would be a function which computes such a number. Such a proposition has an infinite number of solutions: λ *x.(1+ x)*, λ *x.(1+ (1+ x))* ..., and many others. This also introduces the idea of *proofs as programs*, where the proof of a proposition that something exists can be thought of as a function to compute it. Thus the proof that for each two natural numbers there exists another natural number equal to their sum returns a λ-function which computes that sum. We finally come to a link between this work and **INDUCT**, which is that a constructive type theory framework requires induction as a means of proof. Induction within constructive type theory is still an active area of research, in both how induction should take place in general and in identifying which inductions to perform to prove a particular theorem. One problem in constructive type theory is that there are often many different answers, all correct. Where the proof of a proposition is a function, we wish that function to be as efficient as possible. The way a proof of an existential proposition is performed influences the resulting expression, and some work has been done on ways to constrain the resulting λ-function to ensure that it adheres to second order properties such as being tail-recursive. Development of the **SEQUEL** framework for $\mathbf{TT}_0$ and intergration of the analysis methods from **INDUCT** into a single framework could produce a powerful constructive type theory theorem prover.

5.4 SEQUEL as an Enabling Technology

SEQUEL brings together various ideas that have been present in the functional programming and computational logic communities for many years. It's syntax, combining the flexibility of Lisp with the ease of programming and readability of a priority rewrite language, produces an elegant and powerful language. It's basis in Lisp allows access to a large, well-defined environment, providing easy access to optimised system functions and allowing complex procedures unachievable in **SEQUEL** to be programmed in Lisp.

The type-checking mechanism is both flexible and clear, something lacking most of the other typed functional languages available at present (e.g. Haskell has no provision for tracing the progress of the type-checker through the process of checking a function definition).

The sequent calculus extensions provide a common ground for the definition of types and logical frameworks, and although implementation of some logical systems requires considerable translation from the original logical paradigm to sequent calculus, the resulting framework is often much clearer than the original.

The clarity of **SEQUEL** code and the transparency of the type-checking mechanism are both operationally expensive in terms of processing power, and despite some gains made in the development of optimising subroutines in the **SEQUEL** to Lisp compiler (such as the *fastcode* optimiser which locally binds repeated terms), hand-written Lisp will probably always be slightly faster than compiled **SEQUEL** in a large implementation. The small performance degradation is more than offset by the tremendous gains in programmer time that are made when considering the implementation of theorem provers. Most of the verification systems implemented today are designed for the testing and development of logical paradigms. Very few are developed (at least initially) with a real-world application in mind (such as CYC, or the use of **nqthm** to verify VLSI designs). Thus, the clarity of the proof process and the speed at which it is possible to amend, enhance or extend the framework is far more important than gaining every last microsecond of speed from the frameworks produced. Thus, **SEQUEL** is an good tool for logicians developing and analysing the capabilities of different logics for various tasks or investigating properties of logics, since prototyping and development of frameworks to automate proof in these logics is what **SEQUEL** has been designed for.

The time taken to translate a logic into a form for implementation in **SEQUEL** will usually be more than repaid by the researcher not having to spend their time implementing a system for manipulating the terms of the logic. Use of the well-known sequent calculus allows researchers with little knowledge of **SEQUEL** to grasp the nature of the logical frameworks implemented without having to learn a new notation for each logic presented.

SEQUEL itself has developed over the course of this project, so that there are aspects of the project that have been implemented using inelegant software methodologies, and at times the implementation of **INDUCT** has shown the need for capabilities not then available. Enhancements to the current system are possible, as with all computer languages — as with natural language, progress demands change and evolution to avoid extinction — but its current form is highly useful for researchers working in theorem proving without the support of a large group of research assistants implementing fragments of a larger system.

5.5 Final Remarks

Inductive theorem proving is an immense topic — there have been two large groups of researchers extending the capabilities and usefulness of inductive theorem provers for more than fifteen years, in Edinburgh and Austin, with countless others working alone or in small groups in many places in the world. Developing another system to compete with those already produced seems at first to be a useless procedure. However, the systems that exist today are those that have developed over many years, taking inspiration from many sources, with heavy dependence on obscure heuristics and dense implementations. Implementation of a new system designed for typed induction, probably incorporating a constructive type theoretic approach, would clarify much of the theoretical basis for the current state of the art, and allow a degree of rationalisation (due to the clarity of **SEQUEL**'s sequent calculus) which may well open up new avenues of research into the capabilities of inductive theorem proving.

Appendix A

BNF Grammars for SEQUEL

A.1 The Core Language

This is the BNF grammar for the core language:

```
<input> ::= <term> | <list-structure> | <function-definition>
            | (<identifier> <terms>) | (<identifier>)
<function-definition> ::= (define <identifier> <rewrite-rules>)
                  |  (define <identifier> <signature> <rewrite-rules>)
<signature> ::= {<types> -> <type>} | {-> <type>}
<types> ::= <type> | <type> <types>
<type> ::= (list <type>) | (or <type> <type>) | <identifier> | <variable>
<rewrite-rules> ::= <rewrite-rule> | <rewrite-rule> <rewrite-rules>
<rewrite-rule> ::= <test> -> <result> | <test> <- <result>
                  | -> <result> | <- <result>
<test> ::= <test-element> | <test-element> <test>
<test-element> ::= <pattern> | <guard>
<pattern> ::= <constant> | <variable> | <list-structure>
<list-structure> ::= [] | [<patterns>] | [<patterns> | <variable>]
<patterns> ::= <pattern> | <pattern> <patterns>
<guard> ::= (<identifier> <terms>) | (<identifier>)
<terms> ::= <term> | <term> <terms>
<term> ::= <pattern> | (<identifier> <terms>)| (<identifier>)
<result> ::= (<identifier> <terms>) | (<identifier>) | <term>
```

Context conditions:

1.<identifier> is a symbol that is not a <variable>.

2.A <variable> is _, w, x, y, z or any symbol ending with ?.

3.A <constant> is anything that is not a <list-structure> or a <variable>.

4.A <guard> may not contain a variable that does not come before it in the same rule.

5.A <result> may not contain a variable that does not come before it in the same rule.

6.% signs must balance in a rule.

A.2 Sequent Calculus Extensions

The BNF grammar for **SEQUEL**'s sequent calculus notation is:

```
<theory> ::= <interactive-theory> | <non-interactive-theory>
<interactive-theory> ::=
(theory <identifier> :interactive yes <A-axioms>)
                | (theory <identifier> :interactive yes
                                      :pattern <yes-no>
                                      <A-axioms>)
<yes-no> ::= yes | no
<A-axioms> ::= <A-axiom> | <A-axiom> <A-axioms>
<A-axiom> ::= <A-sequents> thus <A-sequents> | thus <A-sequents>
      | <preamble> <A-sequents> thus <A-sequent>
              | <preamble> thus <A-sequent>
<preamble> ::= :name <identifier>
               | <side-conditions>
               | :name <identifier> <side-conditions>
               | <side-conditions> :name <identifier>
               | :name <identifier> <side-conditions> <parameters>
               | :name <identifier> <parameters> <side-conditions>
               | <side-conditions> :name <identifier> <parameters>
               | <side-conditions> <parameters> :name <identifier>
               | <parameters> :name <identifier> <side-conditions>
               | <parameters> <side-conditions> :name <identifier>
<side-conditions> ::= <side-condition>
               | <side-condition> <side-conditions>
<side-condition> ::= (bind <variable> (<symbols>)) | (<symbols>)
<symbols> ::= <symbol> | <symbol> <symbols>
<parameters> ::= <parameter> | <parameter> <parameters>
<parameter> ::= :parameter (<variable> <type>)
<A-sequents> ::= <A-sequent> | <A-sequent> <A-sequents>
<A-sequent> ::= <A-context> |- <t-expr>
<A-context> ::= () | <A> | <t-expr> , <A-context>
<t-expr> ::= <term> * <type>
<non-interactive-theory> ::= (theory <identifier> <B-axioms>)
                             | (theory <identifier>
                                             :interactive no
                                             <B-axioms>)
<B-axioms> ::= <B-axiom> | <B-axiom> <B-axioms>
<B-axiom> ::= <B-sequents> thus <B-sequent>
              | thus <B-sequent> | <B-sequent> iff <B-sequent>
<B-sequents> ::= <B-sequent> | <B-sequent> <B-sequents>
<B-sequent> ::= <A> |- <t-expr>
```

Context conditions:

1.In a <non-interactive-theory> the <identifier> has to be the same as the <type> in the <t-expr> following 'iff' or 'thus'.

2.In 'iff' constructions any variable occurring in the output sequent schemas must appear in the input sequent schema.

3.In a <non-interactive-theory> the <B-sequent> that follows a token of 'iff' cannot unify with any <B-sequent> that follows any other token of 'iff'.

4.<A> here is a terminal, not a category A. The BNF grammar here overlaps with the **SEQUEL** syntax.

Thus the BNF grammar for the *proof* type is:

```
<proof> ::= <proof-object>
<proof-object> ::= [] | [<sequents>]
<sequents> ::= <sequent> | <sequent> <sequents>
<sequent> ::= [<typed-exprs> |- <t-expr>]
<typed-exprs> ::= [] | [<t-exprs>]
<t-exprs> ::= <t-expr> | <t-expr> <t-exprs>
<t-expr> ::= [<wff> * <type>]
```

Context conditions:

1.*Proofs* must be produced by typed functions with output type *proof*.

Appendix B

Part of a Propositional Calculus Proof

A sample session with a propositional calculus framework would be:

```
==================================================
Step 1 [1]

?- (a => c)

1. (a => b)
2. (b => c)

TAB II>> INDIRECT-PROOF
==================================================
Step 2 [1]

?- false

1. (~ (a => c))
2. (a => b)
3. (b => c)

TAB II>> REWRITE ~P=>Q==>P&~Q 1
==================================================
Step 3 [1]

?- false

1. (a & (~ c))
```

```
2. (a => b)
3. (b => c)

:

==================================================
Step 8 [1]

?- false

1. b
2. a
3. (~ c)
4. ((~ b) v c)

TAB II>> V-LEFT
==================================================
Step 9 [2]

?- false

1. (~ b)
2. b
3. a
4. (~ c)

TAB II>> CONTRADICTION
==================================================
Step 10 [1]

?- false

1. c
2. b
3. a
4. (~ c)

TAB II>> CONTRADICTION

yes
```

Which corresponds to the sequent calculus proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash}{c,\ b,\ a,\ (\neg c)\vdash\bot} \qquad \dfrac{\vdash}{(\neg b),\ b,\ a,\ (\neg c)\vdash\bot}}{b,\ a,\ (\neg c),\ ((\neg b)\vee c)\vdash\bot}}{\vdots}}{(a\wedge(\neg c)),\ (a\Rightarrow b),\ (b\Rightarrow c)\vdash\bot}}{(\neg(a\Rightarrow c)),\ (a\Rightarrow b),\ (b\Rightarrow c)\vdash\bot}}{(a\Rightarrow b),\ (b\Rightarrow c)\vdash(a\Rightarrow c).}$$

Appendix C

An Interactive Proof

```
SEQUEL Proof Tool

==================================================

Step 1 [1]

?- "Nothing to Prove!"

INDUCT>> prove-lemma (equal (plus x (plus y z))
(plus y (plus x z))) rewrite

We are trying to prove:
(equal (plus x (plus y z))
       (plus y (plus x z)))
We need to attempt an induction.



Please choose one of the following schemes:

0: No induction



1:
(and (implies (zerop y) goal)
     (implies (and (not (zerop y))
                   (rewrite y (1- y) goal))
              goal))
2:
(and (implies (zerop x) goal)
     (implies (and (not (zerop x))
                  (rewrite x (1- x) goal))
             goal))

Please enter choice of Scheme by number or No Induction (0): 1
```

Both inductions are equally valid, so choose one at random.

```
Please choose amongst the generalisations below.



While trying to prove:

(implies (equal (plus x (plus 1146? z))
                (plus 1146? (plus x z)))
         (equal (plus x
                      (1+ (plus 1146? z)))
                (1+ (plus 1146? (plus x z)))))
The following sub-expressions have been identified
as suitable for generalisation:

(plus x z) below ((0))
(plus 1146? z) below ((0))

Gen1: (plus x z) below ((0))



Gen2: (plus 1146? z) below ((0))



Select one of:
0: None of the above.
1: None of the above (abort proof-procedure).
2: All of the above.
3: Some of the above.

Please enter choice (0-3): 1
```

If we prove the lemma

(equal (plus a? (1+ b?)) (1+ (plus a? b?)))

for use as a rewrite rule, then the above conjecture will become

(implies (equal (plus x (plus 1146? z)) (plus 1146? (plus x z)))
(equal (1+ (plus x (plus 1146? z))) (1+ (plus 1146? (plus x z))))),

in which the *(equal (1+ c?) (1+ d?))* term will be rewritten to *(equal c? d?)*, giving us the conjecture

(implies (equal (plus x (plus 1146? z)) (plus 1146? (plus x z)))
(equal (plus x (plus 1146? z)) (plus 1146? (plus x z)))),

which is trivial to prove, given that the hypothesis and the conclusion of the implication are the same expression. So, we abort the proof procedure and prove an extra lemma.

```
====================================================
Step 2 [2]


?- (implies (equal (plus x (plus 1146? z))
                (plus 1146? (plus x z)))
         (equal (plus x
                      (1+ (plus 1146? z)))
                (1+ (plus 1146? (plus x z)))))


INDUCT>> prove-lemma (equal (plus x (1+ y)) (1+
(plus x y))) rewrite


We are trying to prove:
(equal (plus x (1+ y))
       (1+ (plus x y)))
We need to attempt an induction.



Please choose one of the following schemes:


0: No induction




1:
(and (implies (zerop x) goal)
     (implies (and (not (zerop x))
                   (rewrite x (1- x) goal))
              goal))
```

```
Please enter choice of Scheme by number or No Induction (0): 1


(equal (plus x (1+ y))
       (1+ (plus x y)))
has been proved true!
Adding it as a rewrite rule.
;;; Compiling function induct-rewrite-ns
```

Having proved the new lemma, **INDUCT** returns to the original lemma and runs the proof procedure with the new lemma as part of the rule base, proving it immediately.

```
(equal (plus x (plus y z))
       (plus y (plus x z)))
has been proved true!
Adding it as a rewrite rule.
;;; Compiling function induct-rewrite-s
;;; Compiling function induct-rewrite-ns
==================================================
Step 3 [1]

?- "Nothing to Prove!"

INDUCT>>
```

Bibliography

[Aub79] R Aubin. Mechanising Structural Induction. *Theoretical Computer Science*, pages 329–362, 1979.

[B$^+$89] Alan Bundy et al. A Rational Reconstruction and Extension of Recursion Analysis. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365, 1989.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, 1988.

[Bur69] R.M. Burstall. Proving Properties of Programs by Structural Induction. *Computer Journal*, pages 41–48, 1969.

[BvHHS91] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with Proof Plans for Induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Dil90] Antoni Diller. **Z** *An Introduction to Formal Methods*, chapter 6. Wiley, 1990.

[Duf91] David Duffy. *Principles of Automated Theorem Proving*, chapter 8. Wiley, 1991.

[Moo93] J. Strother Moore. Private Communication, 1993.

[MW77] J.H. Morris and B. Wegbreit. Subgoal Induction. *CACM*, 20(4):209–222, 1977.

[PHH87] G. Plotkin, F. Harper, and F. Honsell. *A Framework for Defining Logics.* Wiley, 1987.

[Ste88] Andrew Stevens. A Rational Reconstruction of Boyer and Moore's Technique, for Constructing Induction Formulas. In *Proceedings of the Eighth European Conference on Artificial Intelligence*, pages 565–570, 1988.

[Tar93a] Mark Tarver. A Language for Implementing Arbitrary Logics. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 839–844, 1993.

[Tar93b] Mark Tarver. *The* **SEQUEL** *Manual*, 1993. Available on-line with **SEQUEL** files.

[Tar95] Mark Tarver. *Functional Programming and Automated Deduction in Sequel.* Wiley, 1995. (Forthcoming).

[Tho91] Simon Thompson. *Type Theory and Functional Programming.* Addison-Wesley, 1991.

[Vad93] Sunil Vadera. Generalisation for Induction. Technical report, University of Manchester Department of Computer Science, 1993. Technical Report UMCS-93-6-8.

[Wal88] Christopher Walther. Argument-Bounded Algorithms as a Basis for Automated Termination Proofs. In *Proceedings of the 9th International Conference on Automated Deduction*, pages 602–621, 1988.