An improved method for the mechanisation of inductive proof

Andrew Stevens

Ph.D.

University of Edinburgh

1989

Table of Contents

List of Figures

Acknowledgements

I would like to express my thanks and gratitude to the following people:

My supervisor, Alan Bundy, for support and assistance well beyond the call of duty.

My colleagues, Paul Brna, Fausto Giunchiglia, Frank van Harmelen, Jane Hesketh, Seán Matthews, Alan Smaill, Lincoln Wallen, and Toby Walsh, for a superb academic environment and Bannerman's Bar.

My parents and my girlfriend for support that bridged the gap between the financial fantasies of H.M. Government and realities of Rent, Rates, and the Supermarket.

This thesis is dedicated to Susan who waited, patiently.

This research was supported by SERC grants GR/E/4459.8, and a studentship for the author.

Abstract

If a large-scale software system is to be dependable and maintainable it must be correct. That is, it must satisfy its specification. Unfortunately, ensuring correctness typically requires extensive formal proofs. The complexity of these proofs, which usually require the application of mathematical induction, is such that their manual construction is prohibitively expensive. The wide-spread development of dependable and maintainable software is therefore, at least in part predicated on the construction of automatic theorem provers capable of mechanising these proofs.

An important problem that any such automatic theorem prover must solve is the construction induction schemata that will allow inductive proofs to succeed - appropriate induction schemata. This thesis further develops existing work on the construction of appropriate inductions due to Boyer and Moore.

A rational reconstruction of Boyer and Moore's algorithm for constructing dual inductions is used to develop an enhanced algorithm suitable for a wide variety of logics. This algorithm, which is simpler and more effective than the original, is then developed into an explicit heuristic theory of appropriate inductions. This theory provides the well-defined, refutable, theoretical basis for the development of improved automatic theorem provers that Boyer and Moore's work omits.

A prototype implementation of the reconstructed algorithm for NuPRL type-theory is presented along with an over-view of the complications inherent in the use of this logic in automatic theorem provers.

Chapter 1

Introduction

1.1 Motivation: Formal Program Development

An important aspect of recent work in software engineering has been the development of formal software development methodologies. The aim of these methodologies is to ensure that software is *correct*. That is, to ensure that programs meets their specifications – formulated in some appropriate logic. Therefore, despite a great deal of variation in detail and philosophy, all the formal development methodologies necessarily have a feature in common. One way or another they all require the developer to construct a formal proof of the correctness of the program being developed. It is this proof that is at the heart of the benefits provided by formal development methods:

- A formal proof that a program meets its specification allows implementation bugs to be excluded — including obscure bugs unlikely to be found through example-based testing.

- The writing of a formal specification suitable for use in a correctness proof often exposes ambiguities and inconsistencies that remain hidden in informal specifications. As a result, design errors (or gaps in understanding) can be detected before development begins, when the cost of corrections is low.

- The modification of software to meet changed specifications is significantly easier if it can be guaranteed to meet its original specification. It is thus

considerably easier to maintain a program that has been proved to meet its specification.

The need for proofs is, unfortunately, also the main reason why formal development methodologies have proved difficult to apply in practice. The proofs required to ensure the correctness of even a modest program are typically very lengthy (see chapter 2). As a result, fully formal methodologies are in practice only usable in areas where correctness is absolutely essential and cost a lesser concern. Typically, this means life-critical applications. There is thus a clear need for automatic theorem provers capable of tackling the proofs involved in formal program development. If such theorem provers were available the benefits of formal development methodologies might be reaped in much wider areas of application.

It is this need for automatic theorem provers for program proofs that provided the primary motivation for our investigation of techniques for automating the application of induction in proofs. Program proofs, because they relate to recursive functions and/or iterative procedures, are almost invariably based on inductive reasoning. Any theorem prover capable of automatically constructing program proofs must therefore be able to apply induction automatically.

This practical application is not, however, our only motivation for the investigation of automatic theorem proving techniques. The automatic construction of proofs, we feel, is a subject worthy of academic investigation in its own right. It provides a well-defined problem with which theories of automated reasoning and artificial intelligence can be tested and developed. Program proofs have the right level of difficulty to provide a good test case for the development of the state of the art in these areas. They are not so difficult as to be hopelessly outside the scope of current theories, yet are difficult enough to provide useful feedback on the short-comings of current ideas.

1.2 The Automatic Application of Induction

The key problem that induction presents to an automatic theorem prover is that of constructing good induction schemata. If an automatic theorem prover is to inductive proof it must somehow choose the right induction schema that allows the rest of the proof to be completed once induction has been applied. That is, the automatic theorem prover must find the induction schema *appropriate* to the goal to be proved. It is this problem of appropriate inductions that forms the main subject of this thesis.

We do not, however, seek to tackle this problem from scratch. The literature on this subject, though relatively sparse, contains several worthwhile contributions. The work that stands out amongst this small body of research, is that of Boyer and Moore [BM79]. Their algorithm for finding appropriate schemata - a component of their well-known automatic theorem prover for pure LISP [BM79] - appears to significantly out-perform its rivals (see chapter 2). It is this algorithm that we use as the starting point for our investigation.

Unfortunately, despite the fact that the Boyer-Moore work is very well documented, it is difficult to build on their success directly. Boyer and Moore present the implementation of their algorithm in detail along with a solid body of empirical evidence that shows the algorithm performs well in a variety of situations. This allows us to repeat their results - a key indication of scientific rigour - but is not quite enough to allow us to set about improving their results directly. This is because, as it stands, Boyer and Moore's work lacks an explanatory theory. Boyer and Moore motivate their algorithm informally through examples, but provide no well-defined theory of appropriate inductions that explains its design. As a result, we lack a theoretical ideal we can compare the Boyer-Moore algorithm against so as to find its faults and suggest improvements.

Therefore, our initial aim in this thesis is to develop a theoretical framework that will allow us to do this kind of comparison. We begin in chapter 3 with a basic theory of appropriate inductions and extend it into a explanatory theory

for the Boyer-Moore algorithm. Once this explanatory theory is in place (chapter 4) we then use it to find flaws in the Boyer-Moore algorithm, and develop an improved algorithm that eliminates these flaws. Finally, on the basis of these results we then present a refined theory of appropriate inductions that provides the theoretical foundation for our improved algorithm (also chapter 4). Our overall methodology can thus be described as one of *rational reconstruction.* We have improved on an existing program by developing a principled theory for the task it mechanises, and using the results to develop an improved program that has an explicit theoretical basis.

This implemented program should not, however, be mis-construed as a complete automatic theorem prover. The improved algorithm was built on top of an existing automated theorem prover along with mechanised proof-procedures for applying the inductions suggested and defining recursive data-types and functions[1]. The inductions suggested by the improved algorithm were tested by attempting to prove the resulting induction sub-goals using the Boyer-Moore theorem prover.

In the remainder of this thesis we will adopt the nomenclature suggested for the Boyer-Moore algorithm in [Bun84] and term it *recursion analysis.*

1.3 The NuPRL Proof Development System

The theorem prover within which we initially developed our rational reconstruction was the NuPRL proof development environment [CAB*86]. NuPRL is a large-scale automated theorem proving environment that also provides a sophisticated interactive proof environment. Its internal architecture is based on the LCF system [GMW79]. The system mechanises its logic as an abstract data-type *proof* in the ML programming language [GMW79]. The rules of inference for

[1]Chapter 5 gives details of these procedures.

the logic are implemented as the constructor functions for *proof*. The key benefit of this approach is that it ensures the soundness of any automatic theorem provers that are written. Since the only primitives available to construct *proof*s correspond to the rules of inference for the logic, any *proof*s constructed by an automatic theorem proving procedure must be sound[2].

NuPRL was chosen as the basis of the rational reconstruction primarily on pragmatic grounds:

- It provided a convenient programming sub-system for writing automatic theorem proving programs. The ML language is well-suited to the symbolic processing characteristic of automatic theorem provers.

- A large number of convenient low-level proof-procedures and auxiliary functions were built-in, or loadable from libraries.

- The sophisticated interactive proof facility promised to be a convenient environment for debugging and developing the rational reconstruction.

- Its logic – an enhancement of Martin-Löf type-theory [Mar79] – was very different from the "computational logic" upon which the the original Boyer-Moore algorithm is based. The development of the rational reconstruction to suit it would thus help to expose any assumptions embedded in the recursion analysis procedure about the particular logic used .

- Its logic was claimed to be well-suited to programming-related applications - deductive program synthesis in particular. Yet, no fully automatic induction theorem prover based on it had been constructed. The development of recursion analysis for NuPRL could thus be expected to yield worthwhile results relating to its use in automatic theorem provers. At the very least, NuPRL logic could be expected to provide a reasonable basis for testing recursion analysis on program proofs.

---

[2]Provided, that is, the designer of the logic has made no mistakes!

Unfortunately, the environmental advantages of using NuPRL were never really fully realised. The problem was that NuPRL (an extremely complicated and unwieldy piece of software) required modification to suit the available equipment (SUN 3 work-stations etc). As a result its sophisticated environmental features were never fully realised. Those that were, were largely offset by the rather inhospitable nature of the ML programming facilities provided. These shortcomings eventually motivated a move to a much simpler implementation of the NuPRL logic in Prolog: the Oyster system [Hor88]. NuPRL proper and ML were used only for initial experiments, the main part of the reconstruction was developed with the Prolog implementation of NuPRL type-theory provided by Oyster. A discussion of the environments provided by NuPRL and Oyster may be found in [CAB*86] and [Hor88] respectively.

The NuPRL logic, by contrast, more than lived up to expectations. The results we obtained in applying NuPRL type-theory within the automatic theorem proving environment are documented in chapter 5.

1.4 Definitions and Conventions

We begin the main part of this thesis by introducing the conventions and terminology we will need. It should be emphasised that in introducing this terminology we are *not* attempting to build a fully formal logical system. The terminology is merely a uniform, clearly defined, jargon useful for discussing the automation of inductive proofs without implying the use of any specific logic.

1.4.1 Notational Conventions

We use names beginning with the lower-case letters x, y, z, a, b, c to indicate object level variables.

Names beginning with lower-case letters $d - w$ indicate object-level terms.

We use names beginning with the upper-case letters $A-Z$ to indicate schema variables (meta-variables) ranging over object level terms.

We use the notation: $[Term/Variable, \ldots, Term/Variable]$ to indicate a substitution. For example:

$T[f(x)/x, g(y)/y]$ indicates the result of replacing all the free instances of x and y in some term T with $f(x)$ and $g(y)$ respectively.

The notation $Term[Variable, \ldots]$ is used to indicate a term containing specific free variables. For example: $T[x,y]$ indicates an arbitrary term that has x and y as free variables.

1.4.2 Rules of Inference and Proof

This thesis is an investigation into a particular aspect of automatic theorem proving. As such, we will often need to illustrate rules of inference or proofs as part of our argument. In order to ensure a coherent exposition we must therefore choose a conventional presentation for proofs and rules of inference. The convention we choose is as follows:

We assume the logic used is presented as a sequent-based refinement logic. That is, we assume each goal formula in a proof is a sequent of the form

$Hypothesis \ldots \vdash Conclusion$

and that proofs are trees of goal formulae built top-down with the theorem to be proved as the root formula. Rules of Inference in this context are rules for extending a leaf of a proof-tree (a goal) by attaching zero or more *sub-goals* to it. That is they are presented *upside-down* in the format:

$$\frac{\text{Template for Hypotheses Required} \vdash \text{Template for Conclusion}}{(\text{New Hypotheses for SubGoal} \vdash \text{Conclusion Subgoal}) \ldots}$$

The sub-goals are assumed to inherit all the hypotheses of the goal formula in addition to any new hypotheses added by the inference rule. A proof is *complete* when every leaf node has had an inference rule applied to it that produces no further sub-goals. A simple example proof should illustrate the principle:

$$\dfrac{\dfrac{\dfrac{}{\vdash a \rightarrow a \wedge a}}{a \vdash a \wedge a} \qquad \dfrac{}{a \vdash a}}{a \vdash a}$$

($\rightarrow$ introduction rule)
($\wedge$ introduction rule)
(hypothesis rule)

The reason we use this slightly unusual presentation is because it is the one used in the theorem proving systems (NuPRL and Oyster) within which we have implemented our rational reconstruction. It is, as we will see in chapter 5, quite well suited to presenting certain aspects of the NuPRL type-theory logic. Thus, in summary:

- Proof-trees are constructed by reasoning **backwards** from the theorem to be proved (at the top) to trivial truths (at the bottom). Implication therefore flows from the bottom of the page to the top.

- Inference rules are presented **upside-down** compared with the usual presentation.

1.4.3 Induction Schemata

Induction is far from being a single monolithic proof rule. It is in fact a whole family of subtly related principles for building such rules. In this thesis, however, unless explicitly stated otherwise, we will be concerned only with the simple *well-founded* inductions, defined as follows:

Definition 1 Well-founded Induction Schema

A well-founded induction schema is a rule of inference of the form:

$$\frac{\begin{array}{c} G[v_1,\ldots,v_n] \\ \hline C^1 \quad \vdash G[v_1,\ldots,v_n] \\ \ldots \\ C^{m-1} \quad \vdash G[v_1,\ldots,v_n] \\ C^m, \quad G[t^m_{1,1}/v_1,\ldots,t^m_{n,1}/v_n],\ldots,G[t^m_{1,1}/v_1,\ldots,t^m_{n,p_m}/v_n] \vdash G[v_1,\ldots,v_n] \\ \ldots \\ C^l, \quad G[t^l_{1,1}/v_1,\ldots,t^l_{n,1}/v_n],\ldots,G[t^l_{1,1}/v_1,\ldots,t^l_{n,p_l}/v_n] \vdash G[v_1,\ldots,v_n] \end{array}}{}$$

for which we can prove $C^1 \vee \ldots \vee C^l$ and find a subset of $\{v_1,\ldots,v_l\}$, $\{v_a,\ldots,v_b\}$, a function M and a well-ordering $\ll$ such that we can prove:

$$C^i \rightarrow M(t^i_{1,j},\ldots,t^i_{n,j}] \ll M(v_1,\ldots,v_n)$$

for all $m \leq i \leq l$ and $1 \leq j \leq p_i$.

■

For example, simple induction over the length of a list would be expressed as:

$$\frac{\begin{array}{c} G[l] \\ \hline l = [] \vdash G[l] \\ l \neq [], G[tl(l)/l] \vdash G[l] \end{array}}{}$$

Definition 2 Measured Variables

The *measured variables* of an induction schema are the variables $\{v_a,\ldots,v_b\}$ (see above) that appear in the proof of well-foundedness.

■

Informally, the measured variables of an induction scheme can be thought of as the variables the induction is "over". For example, x is a measured variable of simple structural induction over the Peano integers:

$$\frac{\begin{array}{c} \vdash Goal \\ \hline x = 0 \vdash Goal \\ x \neq 0, Goal[p(x)/x] \vdash Goal \end{array}}{}$$

($p(x)$ is the "predecessor" function introduced on page 8)

Definition 3 Step-substitution.

The *step-substitutions* of an induction schema are the substitutions for variables in the induction's conclusion used to construct the induction hypotheses in the induction's step-cases.

■

For example, the step-substitutions for the induction schema:

$$\frac{\vdash Goal}{\begin{array}{c} x = nil, y = nil \vdash G \\ y \neq nil, G[x/x, tl(y)/y] \vdash G \\ x \neq nil, G[tl(x)/x, y/y] \vdash G \end{array}}$$

are $[x/x, tl(y)/y]$ and $[tl(x)/x, y/y]$.

Note that we deliberately permit trivial substitutions, or substitutions for non-measured variables to appear in the step-substitutions of an induction schema. This is because inductions appropriate to a particular goal may be constrained by a requirement not to substitute for a particular variable, or to substitute in a particular way for some unmeasured variable. We represent these constraints through trivial substitutions or substitutions for non-measured variables.

1.4.4 Functions

Definition 4 Function

For the purposes of this thesis we use the word "function" in a higher order sense. That is, we use it to denote functions, predicates, functionals, and connectives.

■

Definition 5 Primitive Function

A primitive function is a function that is defined axiomatically, rather than defined computationally in terms of already defined functions. Typically these

are the constructor functions introduced to define abstract data-types. For example, the Peano naturals are defined by introducing the functions 's' and '0' with the axioms that:

0 is an integer

$s(x)$ is an integer if x is an integer

■

Definition 6 Defined Function

A defined function is a function defined in terms of the application of previously defined or primitive functions to a set of formal arguments.

■

For example, the definition:

$plus2(x) = s(s(x))$

introduces a simple defined function *plus2* in terms of the application of the (primitive) successor function s.

$p(x) = y$ if $x = s(y)$ is a conditionally defined function.

Definition 7 Recursive Function and Justifying Measure

A recursive function is a defined function that is defined in terms of itself as well as previously defined or primitive functions. Unless explicitly stated otherwise, we will consider only well-founded recursive functions. That is, recursive functions defined

$f(a_1, \ldots, a_n) = D[a_1, \ldots, a_n]$

for which we have found some other function m, a subset of formal arguments $\{a_{i_1}, \ldots, a_{i_m}\}$ and a well-ordering $<$ such that for each recursive call $f(t_1, \ldots, t_n$ in the function's definition D we can prove:

$m(t_{i_1}, \ldots, t_{i_m}) < m(a_{i_1}, \ldots, a_{i_m})$

We term the expression $m(a_{i_1}, \ldots, a_{i_m})$ the *measure* justifying the function's well-foundedness.

■

For example, the function *member* defined:

$$\begin{aligned} member(x, l) \equiv\ & \text{if } x = nil \text{ then } false \\ & \text{else if } x = hd(l) \text{ then } true \\ & \text{else } member(x, tl(l)) \end{aligned}$$

is well-founded as in the only recursive reference in its definition the length of its first argument decreases. The measure justifying *member* is therefore $length(x)$.

Definition 8 Recursion Argument

The *recursion arguments* of a recursive function are the subset of formal arguments $\{a_{i_1}, \ldots, a_{i_m}\}$ used in the proof of well-foundedness constructed for its definition (see definition 7 above). Informally, the recursion arguments of a function can be thought of as those the function's recursion is "over".

■

For example, the recursion argument of the "append" function defined:

$$\begin{aligned} append(x, y) \equiv\ & \text{if } x = [] \text{ then } y \\ & \text{else } cons(hd(x), append(tl(x), y)) \end{aligned}$$

is x since the well-foundedness of the recursion is shown by proving that

$\forall x.0 \leq length(x)$ and $\forall x.x \neq [] \rightarrow length(tl(x)) < length(x)$.

The notion of recursion argument can thus only be meaningfully applied to arguments of recursive functions whose definitions have known proofs of well-foundedness.

1.5 Layout

The layout of the remaining chapters of this thesis is as follows:

Chapter 2 – A survey of related and relevant work.

We motivate more thoroughly the need for automatic theorem provers in formal software development methodologies, and review work directly relevant to our own including the work of Boyer and Moore. On the basis of our review we justify the decision to base our work on the further development of the Boyer-Moore recursion analysis procedure.

Chapter 3 – The Recursion Analysis Procedure.

We present a basic theory of appropriate inductions and analyse in detail the Boyer-Moore recursion analysis procedure. On the basis of this analysis we present a detailed provisional theory for recursion analysis.

Chapter 4 – The Rational Reconstruction.

We use our detailed theory of recursion analysis to find flaws in the Boyer-Moore procedure. On the basis of the results, we construct an enhanced procedure that successfully deals with a wider variety of proofs, and present a refined theory of appropriate inductions that underpins the enhanced procedure.

Chapter 5 – NuPRL Type-Theory.

We present the results obtained in applying our rational reconstruction to a constructive type-theory: NuPRL type-theory.

Chapter 6 – Further Work and Conclusion

We present further work suggested by our investigation and summarise the results we have obtained.

Chapter 2

Related and Directly Relevant Work

2.1 Context and Related Work

In the introduction to this thesis, we explained that a major motivation for research in automatic theorem-proving is its application to the construction of formally correct software. This is especially true of the work dealing with induction, because induction almost invariably plays a critical role in proofs about programs. We therefore begin our survey of work relevant to this thesis by motivating more thoroughly the need for automatic theorem provers in the production of formally correct software. There are two main approaches to constructing correct programs proposed in the literature: one based on verifying correctness by proving it, the other based on guaranteeing correctness by building programs directly from their specifications so that correctness follows by construction.

2.1.1 The Verification Approach

The verification approach is based on the idea of ensuring a program's correctness by formally proving that it meets its specification. The development methodology it proposes is to gradually decompose a specification until each component is small enough to be realised with an efficient sub-program that can be proved correct. This process usually also involves replacing abstract data-types in the original specification with efficient concrete ones. A process known as "reification". Once the correct sub-programs have been proved correct, the

original specification can be realised by simply composing the sub-programs in the reverse of the specification's decomposition.

The actual source of the correct sub-programs depends on realisation of the verification approach used. In the more mature formalisms which are orientated toward conventional imperative programming environments (e.g. [Spi89, Jon79]) the source is assumed to be a human programmer. An interesting alternative, however, is provided by the *Deductive Program Synthesis* technique [MW80,Biu88], [CAB*86]. In this case, the correct program realising a specification component is mechanically extracted from a proof of its existence. In either case - verification or synthesis - an automatic theorem prover is likely to be a vital component in any practical implementation of this approach. The necessary formal proofs of correctness or existence, though usually quite shallow, are notoriously lengthy. The skilled labour required to construct them manually is prohibitive. For example, a formal proof of even a trivial theorem like

$$\forall l, r.append(reverse(l), r) = acc\_reverse(l, r)$$

where: *append* is the function appending two lists.
reverse is the function reversing a list.
and *acc_reverse* is the function defined by:

$$acc\_reverse(a, b) = \text{if } a = [] \text{ then } b \text{ else } acc\_reverse(tl(a), cons(hd(a), b))$$

requires several tens of steps to complete.

The decomposition steps used to break up a specification may also require lengthy proofs to show their correctness. In particular, the introduction of recursion (or a loop) requires an inductive proof to show it correct. We give a brief over-view of a typical framework for verification-based program development - VDM [Jon79] - section 2.1.4 .

2.1.2 The Specification Reification Approach

The specification reification approach is (at least partly) intended to eliminate the need for expensive explicit proofs of correctness when developing correct programs. The approach is based on the use of declarative programming languages that form executable subsets of logics. The idea is that a specification written in such a logic can then be implemented by simply rewriting it until it inhabits the executable subset. As long as the rewritings - reifications [1] - applied leave the semantics of the specification/program unchanged, correctness is guaranteed. The only major drawback is the restriction to declarative programming languages and logics that have such languages as executable subsets. The system in practice typically uses either pure functional programming languages (subsets of equational logic) or pure logic programming languages (subsets of the predicate calculus).

The development methodology that results is a process of step-wise reification of specifications to eliminate non-executable or inefficient constructs in favour of executable or more efficient ones. It should be emphasised that this process is very different than the kind of reification associated with the verification based approach. In the verification based approach, reification (if it occurs) only eliminates abstract data-types in favour of concrete ones that can be more efficiently implemented. The specification itself, however, remains non-executable.

Given this background, why should an automatic induction theorem prover be useful for development of a program by reification from its specification? An important aim of the reification based approach is, after all, to ensure correctness by construction rather than by an explicit proof. The answer is that, rather than eliminating (inductive) proof, the reification approach in fact merely obscures it. The reification steps required in general do not unconditionally produce a new version of the program/specification equivalent to the original. Instead, equivalence preservation generally depends on the original program/specification

---

[1] Sometimes also known as "refinements"

satisfying some condition. These equivalence preservation conditions need to be proved if the program derived is to be guaranteed correct. The exact form of the proofs requirements that emerge depends on the specific realisation of the reification approach used. For instance:

- Darlington and Burstall's prototypical "unfold/fold" procedure for transforming functional programs into more efficient forms applies equality lemmas to do the required rewriting of function bodies. These equality lemmas need to be proved - which often requires induction.

- The deductive approach to refining logic programs developed by Hogger [Hog81] and Clark and Sickel [CS77] requires a proof that the input/output relation computed by a program is a a superset the input/output relation defined by its specification. This proof often requires induction.

- The transformations used in the extended unfold/fold procedure of Tamaki and Sato [Tam84] only conditionally preserve equivalence. Extensive proofs are often required to show that the equivalence-preservation conditions hold in any given instance. These proofs also often require induction.

The reification based approach thus requires proofs almost as much as the verification based one [San88]. Hence, for the reasons outlined above and expanded on at the end of section 2.1.4, its practical application requires powerful automatic theorem provers. Unfortunately, very little on the work on the specification reification approach has to date considered this issue. The only work known to the author is that of Bauer et al [ea87], Nakajima et al [NY83], and Fujita [FK86]. Only the work of Fujita (covered in section 2.2.5) has a major automatic theorem proving component.

A further point worth reiterating about the reification based approach to program development is that it implies a very close relationship between the specification and programming languages chosen. It implies that any valid specification can be rewritten into an efficiently executable program with the same

declarative semantics. The specification language must thus in practice be either be identical with, or a subsume the programming language. An unfortunate consequence of this constraint is that research in the area is divided into two largely non-interacting camps according to the programming/specification language paradigm advocated. That is, research is split between then logic-programming and functional programming (algebraic specification) communities. We illustrate the kind of framework typically used in the reification approach in the next section with Darlington and Burstall's unfold/fold procedure.

Finally, it should be emphasised that the distinction between the specification reification approach, and the verification approach is by no means hard and fast. There is a significant grey area between logic programming, deductive program synthesis and algebraic specification, and all of these approaches require proofs.

2.1.3 Darlington and Burstall's Unfold/Fold Procedure

Darlington and Burstall's unfold/fold strategy [BD77] is significant because it provides the prototype for almost all procedures for refining clear but inefficient specifications/programs into efficient equivalents. Originally developed for simple functional programs/specifications - it provides a neat overall framework for optimising recursively defined declarative programs.

We will illustrate the strategy, and its neat capture of the generic steps in optimising a recursive program, with a simple example:

$dot(a, b, n) + dot(c, d, n)$

where *dot* is a function for computing vector products defined by[2]:

$$dot(x, y, 0) = 0$$
$$dot(x, y, s(n)) = x_{s(n)} \times y_{s(n)} + dot(x, y, n)$$

---

[2]In its original form Darlington and Burstall's system dealt with functional programs specified by recursion equations

The first step in Darlington and Burstall's strategy is motivated by the observation that significant optimisation of a (declarative) program generally implies the use of a new recursion scheme. The procedure thus begins by supplying the syntactic pre-requisites for defining a new recursion scheme for the program: a new function (f say) is defined that computes the same result as the program. In our example f is simply:

$$f(a,b,c,d,n) = dot(a,b,n) + dot(c,d,n)$$

but in other cases the original program may require some "trick" transformation before it can serve as f's body (see [BD77] for details).

Once f is defined, the next step is to expand the functions used in the body of f's definition by applying their definitions. Each function in the body of f's definition is expanded out by "unfolding" its definition (the unfolding process is defined formally in [BD77]). The aim of this procedure is to provide an explicit representation of one "step" of the computation specified by the program. The recursive parts of the procedure appear as recursive instances of the unfolded functions. In our example unfolding gives:

$$f(a,b,c,d,0) = dot(a,b,0) + dot(c,d,0)$$
$$f(a,b,c,d,s(n)) = dot(a,b,s(n)) + dot(c,d,s(n))$$

and then:

$$f(a,b,c,d,0) = 0 + 0$$
$$f(a,b,c,d,s(n)) = a_s(n) \times b_s(n) + dot(a,b,n) + c_s(n) \times d_s(n) + dot(c,d,n)$$

Once the unfolding has been accomplished, the actual optimisation task is to rewrite the "step" so that (if possible):

- Any redundancy in the computation exposed by the unfolding is eliminated.

  For example, a repeated sub-expression might be eliminated in favour of a variable bound to the result of evaluating the sub-expression once and for all.

- The structure of the step is as efficient as possible.

  For example, $append(append(a, b), c)$ computes the same result as, but is in general much less efficient than, $append(a, append(b, c))$.

- The recursive instances of the unfolded functions appear in such a way that they can be replaced by (preferably tail-recursive) instances of f using f's original definition.

The rewritings that can be performed, of course, depend entirely on the available lemmas mentioning the functions appearing in the unfolded definition of f. These lemmas and their applicability in a particular context need to be proved, at which point an automatic theorem prover would be of considerable value.

In our example we merely apply the definition of + along with its commutativity and associativity to obtain:

$$f(a, b, c, d, 0) = 0$$
$$f(a, b, c, d, s(n)) = a[s(n)] \times b[s(n)] + c[s(n)] \times d[s(n)] + dot(a, b, n) + dot(c, d, n)$$

The final step, once the optimisation is complete, is to fold. The original definition of f is applied to replace recursive instances of unfolded functions with recursive instances of f. In our example this gives us:

$$f(a, b, c, d, 0) = 0$$
$$f(a, b, c, d, s(n)) = a[s(n)] \times b[s(n)] + c[s(n)] \times d[s(n)] + f(a, b, c, d, n)$$

a definition for f that has the overhead of only one recursion rather than two.

Darlington and Burstall's unfold/fold procedure should not, however, be understood as a complete proposal for a refinement approach to program development in itself. It merely provides an elegant basis for one particular family of reifications. Two important issues are not addressed at all:

- A suitably expressive specification language.

  A good specification language should make it easy to express what a program should compute without worrying about how it should be computed. It must nevertheless still permit specifications to be refined to the extent that they become efficiently executable programs.

  The simple functional programming languages the original unfold/fold procedure is based on demonstrably satisfy the latter requirement. Unfortunately they fail the former requirement quite badly. It is often very difficult to specify a computation in a simple functional programming language without deciding roughly how it should be performed. A rather more expressive "wide-spectrum" language is required.

- Automated Application.

  The reification of a specification into a program using the fold/unfold technique (or any other specification reification procedure) is by no means a trivial task. The construction of an appropriate sequence of unfoldings, foldings, and rewritings is almost as difficult as the construction of a formal proof of a program. Thus, as with program verification or deductive program synthesis, unfold/requires a considerable degree of automation before it can provide the basis for a viable development methodology. The amount of skilled labour needed to apply it manually is prohibitive.

The first of these issues - the development of expressive specification formalisms that can be reified into executable programs - has been quite heavily researched. A great deal of current work in algebraic specification [San88,EM85, ST86b] and logic programming [Hog81] focuses on formalisms to allow the reification into programs of more expressive specification languages. This work is reflected in the recent development of function programming languages such as Hope [BMS86], and standard ML [HMM86]. Sannella and Tarlecki have proposed an extended ML language [ST86a] that incorporates facilities for expressing (non-executable) specifications as well as (executable) functions. Recent

versions of the Hope language [Per86] incorporate logic-programming features intended (primarily) as a mechanism for improving the language's expressive power as a specification language. The language is supported by a sophisticated program development environment that allows inefficient Hope specifications to be transformed into efficient programs.

The automation of the unfold/fold procedure (and other reification procedures) has, by comparison, recieved little attention. The main work aimed at the automation of the fold/unfold procedure is that of Feather (a student of Burstall). In [Fea79] Feather describes a language that allows the automated pattern-directed application of fold/unfold, and an informal strategy for its application. Feather's system, however, still requires considerable - almost continuous - user intervention. The difficult formulation of an equivalent to the program to be optimised that allows the optimisation to succeed is still largely accomplished by the user. The rewriting lemmas applied by the system are assumed to have been proved by the user.

2.1.4 VDM - The Vienna Development Methodology

VDM - the Vienna[3] Development Methodology [BJ78,Jon79] - is a typical example of a mature proof-based program development methodology. It is orientated toward manual development of programs in imperative programming languages and is based around an extensive first order specification language. In VDM programs are specified by expressing the conditions on the program's storage locations assumed to hold before execution and the conditions that must hold after execution. A procedure to subtract a natural number from a storage location x holding a larger natural number would, for example, be specified:

---

[3]Named after IBM's Vienna research laboratory where it was originally developed

REDUCE($d : N$)
ext wr $x : N$
pre $x > d$
post $x = \overleftarrow{x} - d$

The first line of the specification merely serves to identify the procedure specified and introduce and type its parameters. In this case the name is "REDUCE" and there is one parameter d. The second line introduces the names and types of any non-local storage locations used - in this case x holding a natural number. The next line expresses the pre-condition - the condition on the storage assumed to hold before execution. In this case it is that the value in x is bigger than that of d. The final line expresses the condition that is specified to hold after execution - the post-condition. The undecorated variables denote the values in storage locations after execution. The $\leftarrow$ decorated variables denote the values in storage locations before execution. Thus, in the example above x is specified to hold d less than its original value after execution.

Even from this simple example it should be abundantly clear that a practical specification language is much more than a raw logic. It is not enough that a specification language simply provides the bare logical machinery necessary for writing specifications and realising them correctly. It must provide convenient notations that allow this to be done concisely and elegantly. The VDM specification language accordingly builds many useful notational features on top of the simple many-sorted predicate calculus outlined above. These provide a comprehensive framework for writing down clear, concise, specifications and implementing them through a formal design process. The VDM language's main features are:

- Notation for abstract data-types.

  VDM provides built-in notation for sets, sequences (lists), maps (look-up tables or arrays), and structured types ("records" or "structures").

- Sub-types.

VDM provides a notation that allows arbitrary sub-types of the types outlined above to be expressed. The sub-type is expressed by associating a "data-type invariant" (a predicate characterising the sub-type) with a base-type defined using the notation mentioned above. This facility allows the convenient specification of efficient concrete data-types (e.g. balanced trees) to replace abstract ones (e.g. sets) during the reification of a specification (see below).

- Partial Operations.

  The VDM specification language has built-in notations to support the use of partial functions and procedures in specifications.

The VDM formalises the program design process as the application of two distinct types of specification transformations: data reification and operation decomposition.

Data reifications correspond to design decisions about data-representation. They rewrite the specification to replace inefficient abstract data-types with more complex concrete data types that can be more efficiently implemented. A typical example of a data reification might be redefining a "dictionary" from a set of words to a list of words containing no duplicate members. The data-type *Dict* defined as:

$Dict = set\ of\ Word$

would be replaced with *Dicta* defined as

$Dicta = list\ of\ Word$ where $invariantDicta(d) \equiv unique(d)$

The relationship between the original abstract data-type and its more concrete replacement is defined by the introduction of a "retrieve function". This maps elements of the concrete data-type onto the elements of the original abstract data-type that they represent. In this case the function would be:

$retrieveDict(d) \equiv setify(d)$

The proof obligation that arises is to show that the reified data-type is adequate - that every element of the abstract type has a representation in its concrete replacement. Any operations on the original abstract would of course need to be respecified using the new concrete type. This of course adds an obligation to prove that the reified operations model the abstract ones. That is that:

$$\forall v_i . preA[retrieveA(v_i)/v_i] \rightarrow preR$$
$$\forall v_i, \overleftarrow{v_i} . (preA[retrieveA(\overleftarrow{v_i})/\overleftarrow{v_i}] \wedge postR) \rightarrow postA[retrieveA(\overleftarrow{v_i})/v_i, retrieveA(v_i)/v_i]$$

where $preA$ and $postA$ are the pre and post conditions of an operation expressed using the original abstract type A.

$preR$ and $postR$ are the new pre and post conditions for the operation expressed using the reified type A.

and v_i are the variables denoting the storage locations (originally of type A) reified to storage locations of type R.

Operation decompositions correspond to design decisions about program structure. They decompose the specification into components whose realisations, when combined with an appropriate control construct will realise the specification as a whole. A typical example of an operation decomposition might be splitting a specification for adding a word to a dictionary according to whether the word is already present or not. The specification

INSERT($w : WORD$, $d : Dict$)
pre $true$
post $d = \overleftarrow{d} \cup \{w\}$

can be replaced by:

INSERT($w : WORD$, $d : Dict$)
pre $true$
if $w \in d$
then pre $w \in d$
post $d = \overleftarrow{d} \cup \{w\}$

else pre $\neg w \in d$
post $d = \overleftarrow{d} \cup \{w\}$
post $d = \overleftarrow{d} \cup \{w\}$

Thus breaking it into two components whose realisations combined with an if-then-else construct will realise the original specification. Once the specification has been broken up sufficiently it is realised by simply writing code for each component. Correctness is ensured by proving that each code fragment achieves its post-conditions under the assumption that its pre-conditions hold. Proof is also required for some operation decompositions, in particular the introduction of loops. It should be noted that these verification proofs, as in most manual verification based program development methods [Spi89], are normally intended to be *informal.* VDM advocates the use of manual natural deduction style proofs with the formality of proofs adjusted to suit its importance. The suggested philosophy is to keep all but the trickiest proofs very informal indeed.

We would argue that this semi-formal manual approach - formal specification, informal proofs - is seriously flawed. The informal proofs are no more likely to be correct than programs written by hand. Some errors may be detected but many others may still be missed through trivial errors of typography or systematic mis-conceptions. Even the best programmers will often see what is *intended* rather than what has been *written.* It is only through mechanically verified fully formal proofs that correctness can be confidently inferred. As we have already noted, the size of these fully formal proofs implies the use of automatic theorem provers.

The need for correctness is not the only factor that implies the use of mechanical theorem provers. The majority of applications software tends to require periodic modification ("maintenance") to keep it in line with changing requirements. In a framework such as VDM each change will require a manual proof of the modified sections. The costs of continuously providing the necessary (very highly skilled) work-force mean that methods such as VDM are inapplicable to the bulk of commercial software development. The only major applications

of manual methods like VDM have been in cost-insensitive areas with relatively static software and a high reliability requirement - systems and avionics software. The widespread adoption of program verification thus implies the development of methodologies based around (at least partially) mechanical proof.

2.2 Directly Related Work

2.2.1 Introduction

As has been mentioned in the introduction, the work presented in this thesis is based on a technique for choosing inductions developed by Boyer and Moore as part of their well-known theorem prover [BM79]. This "recursion analysis" technique is, however, by no means the only method known for constructing appropriate induction schemes. The literature contains numerous alternative approaches that might also provide the basis for further work. This chapter therefore begins the main part of our thesis with a critical survey of the various approaches to automating induction documented in the literature.

The subsection immediately following (2.2.2) provides a brief over-view of the Boyer-Moore theorem prover and the recursion analysis technique. The remaining subsections (2.2.3 to 2.2.6) form a critical survey of the main alternatives. The aim of these subsections is mainly to motivate the decision to base this thesis on the development of recursion analysis.

2.2.2 The Boyer-Moore Theorem Prover

The Boyer-Moore theorem prover (henceforth referred to as the BMTP), is a powerful automatic theorem prover that has been under continuous development by its authors since the early 1970's. The system is based around a simple first-order "computational logic" of the authors own devising that is based on pure LISP. Its salient features are a very rich mechanism for defining well-founded

recursive data-types and functions, and the support of only universal quantification. For example, the theorem:

$$\forall x, y.odd(x + y) \rightarrow odd(x) \vee odd(y)$$

becomes:

```
(IMPLIES (ODD (PLUS X Y)) (OR (ODD X) (ODDP Y)))
```

when expressed in Boyer and Moore's computational logic. All variables are implicitly universally quantified over the whole goal. The logic though not as powerful as full first order predicate calculus is carefully tuned to the BMTP's intended application domains: software and hardware verification. In these area it has proved more than sufficient for most applications.

The system itself is largely orientated toward automatic (rather than automated) theorem proving. The bulk of the system comprises a set of powerful high-level proof procedures which are applied under the control of a sophisticated hard-wired proof strategy. User interaction with the system is limited to the formulation of lemmas, and tagging of these lemmas according to their intended purpose(s). Thus, although the system's proofs are always fully automatic, difficult problems are generally brought within the system's capability by a user-suggested decomposition into lemmas. Nevertheless, given the current state of the art, the complexity of the problems that can be proved without decomposition is impressive. The system has in several instances [BM84,BGM82] been successfully used to verify real programs and VLSI designs. The BMTP provides a sophisticated automatic and automated theorem proving environment capable of dealing with problems of real-world complexity. A description of the latest version of the Boyer-Moore system and the improvements made since [BM79] was published can be found in [BM88a].

The Proof Strategy

Boyer and Moore describe the proof strategy employed in the BMTP by analogy with the kind of recursive cascade or waterfall popularised in M.C.Escher prints. A conjecture to be proved starts at the beginning (top) of the cascade. As it falls, each proof procedure in turn is applied to it until one manages break it into one or more new subgoals. These are then recursively run from the top of the cascade in their own right. Any subgoals proved by the proof procedures disappear (evaporate?) from the cascade. The theorem is proved when no more sub-goals remain to be run through the cascade. A sub-goal that runs off the end of the cascade is regarded as unprovable, and thus causes the system to give up on the conjecture. Figure 2–1 illustrates the design in a diagram.

The (fixed) ordering of the proof procedures in the cascade is dictated mainly by the various dependencies that exist between them, and search control considerations. Thus, for example:

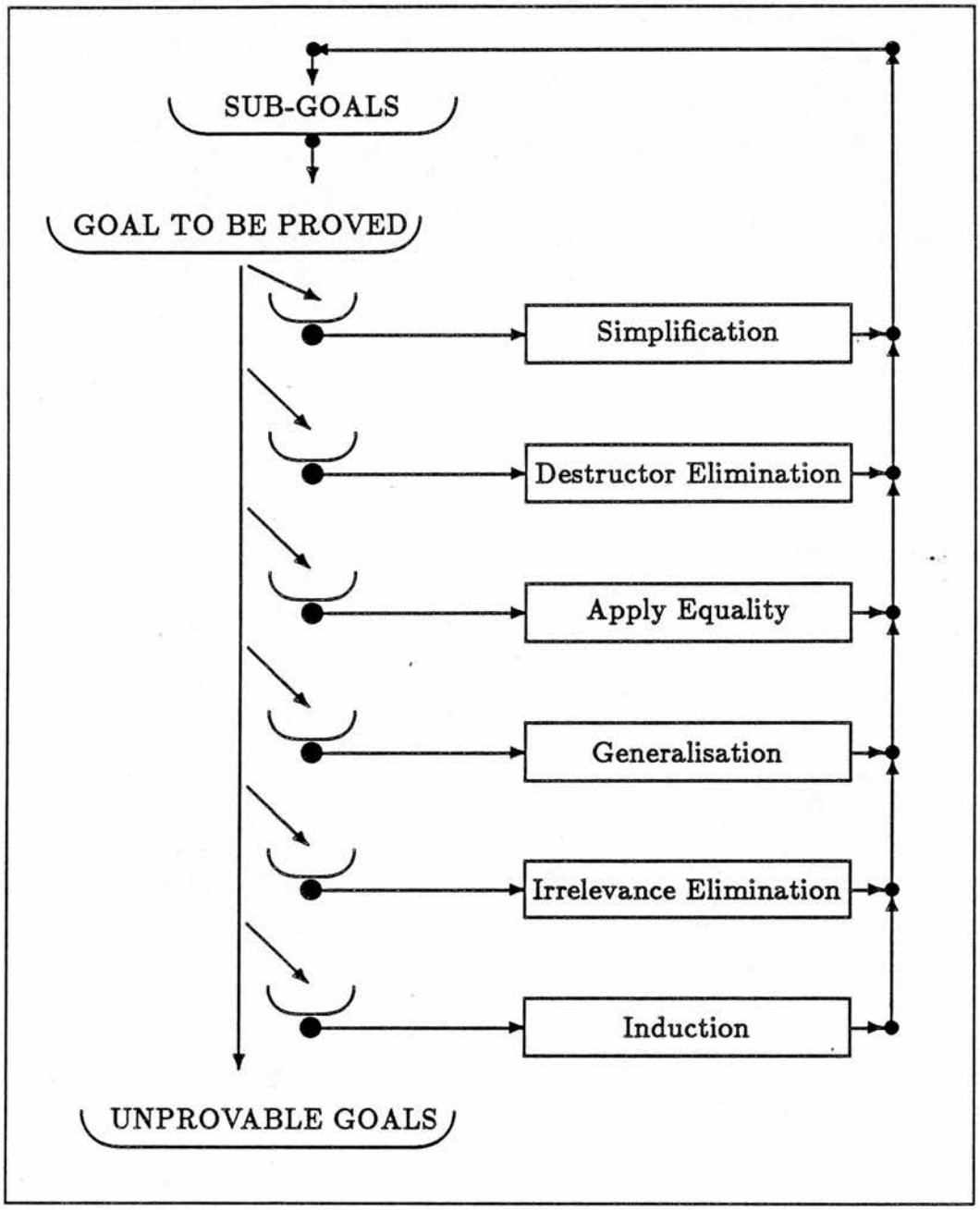
- Induction is last because it relies on the pre-processing performed by other procedures and because it, unlike the other proof procedures, actually increases the complexity of the goal it is applied to. For example, a simplification left until after induction has been performed may need be performed once for each case of the induction, rather than just once.

- Simplification and destructor elimination, which are "safe" because they cannot produce non-theorems from theorems, come before generalisation which is "risky" because it can.

- Irrelevance elimination comes before induction because choosing an induction can be much easier if irrelevant terms are eliminated. It comes after everything else because it may eliminate as non-theorems goals which are in fact theorems.

It should be emphasised that Boyer and Moore's proof strategy is by no means just an exhaustive application of the various proof procedures. Only part of the

Figure 2–1: The Boyer and Moore Proof Strategy



(See subsection 2.2.2 for details of the proof procedures).

proof strategy is made explicit in the ordering of the proof procedures in the cascade - an equally important part is implicit in the ways the proof procedures effect each others applicability. For example, equality will more often than not only be applied following an induction. The reason is that once equality has been applied, it generally requires an induction to introduce new equality hypotheses before worthwhile new equality reasoning can be done.

The combination of cascade ordering and interactions results in an effective proof strategy that constrains search considerably more than immediate inspection of the cascade design would suggest. This implicit proof strategy is roughly as follows:

1. Try to prove the goal using computation and propositional reasoning.

2. If sub-goals are left, try to apply any available equalities, and see if 1 can make any more progress.

3. If there are still any sub-goals left, get them into a form suitable for induction.

4. For each subgoal:

   Find an appropriate induction, and apply it.

5. For each of the resulting sub-goals:

   apply the induction hypotheses (if any) and goto 1.

The advantage of the cascade implementation is that it allows the proof strategy to be flexible enough to deal with situations requiring deviations from the "usual" plan. The effectiveness of this approach can be gauged by the fact that the BMTP is in fact incapable of back-tracking. Its performance is entirely the result of the control strategy implicit in the design of the proof procedures and cascade. Brute force search plays no role.

The Proof Procedures

The proof procedures, in the order they are applied, are listed below. It should be emphasised that in the space available it is simply not possible to do justice to the subtleties of the various heuristics employed. The in-depth description of the induction procedure (admittedly the most complicated) occupies the whole of the next chapter. In recent implementations an additional proof procedure -a linear arithmetic decision procedure - is also integrated into the proof strategy [BM88b].

1. **Simplification** (pp.92 – 129 [BM79])

   This procedure is essentially the uniform theorem-proving stage in the Boyer-Moore theorem prover. It normalises the goal into clausal form, and then simplifies the literals of the clauses by a heuristically guided application of the function definitions, axioms, and suitable lemmas known to the system. Any clause containing a literal reduced to "true" is then eliminated as proved. It should be noted that clausal form is used purely as a convenient normal form in the BMTP. It is not a resolution based theorem prover.

2. **Destructor Elimination** (pp.130 – 144 [BM79])

   Destructor elimination is a heuristic procedure for eliminating "awkward" destructor functions in terms of constructors that are easier to deal with. For example, if a goal is in terms of x and $p(x)$ then it usually makes things easier if these are rewritten to $s(y)$ and y respectively. The basic mechanism of the procedure is to search for, and try to apply suitable lemmas of the form: $Hyp \rightarrow term = variable$.

3. **Equality Application** (pp.145 – 150 [BM79])

   The equality application procedure's role is to appropriately use any equalities introduced as hypotheses of sub-goals. It applies a complicated set of heuristics to guide the selection and application of equalities according to the context of the sub-goal in which they appear.

4. **Generalisation** (pp.151 – 158 [BM79])

   An important problem in inductive proofs is the almost paradoxical observation that certain goals can only be proved by induction if they are strengthened by generalisation. Boyer and Moore's generalisation procedure deals with an important class of such situations that commonly occur after one induction has already been applied. Namely, the situations where induction leaves step case sub-goals with identical sub-terms appearing in the induction hypothesis and conclusion. In this situation, subject to certain heuristic restrictions, the generalisation procedure will eliminate the common sub-terms in favour of a new variable.

5. **Irrelevance Elimination** (pp.159 – 162 [BM79])

   The role of the Irrelevance Elimination procedure is to eliminate irrelevant hypotheses from sub-goals. This is useful, because irrelevant hypotheses can confuse the choice of appropriate inductions, and can also expose sub-goals that are trivially unprovable. The procedure works by finding "islands" of literals in the clausal form of goals not sharing any common variables with the rest of the literal. Any islands that appear to be falsifiable according to a various heuristic rules are eliminated as irrelevant.

6. **Induction**

   The final proof procedure, induction, finds and applies inductions appropriate to the goals it is given. It is, in many ways, the heart of Boyer and Moore's theorem prover both in terms of design and research focus. The proof-strategy is heavily optimised for inductive proofs, and the proof procedures carefully designed to support induction. Indeed the bulk of the proof procedures the system implements are largely, or entirely, related to the induction process. For example, the only real purpose served by generalisation and destructor elimination procedures is to make goals more suitable for induction. Similarly, the procedure for applying equalities is strongly aimed at dealing with the kind of equality hypotheses introduced

by induction. In terms of research results, almost all the major contributions of the Boyer and Moore system relate to induction. The outstanding example, is of course the recursion analysis technique, that provides the basis both of Boyer and Moore's induction procedure and of the work presented in this thesis. In the remainder of this subsection we present a brief over-view of the technique - sufficient to permit comparison with its rivals in the literature - and consider the motivation behind its choice as the basis of this thesis. A more detailed analysis is postponed until chapter 3.

Recursion Analysis

The idea behind Recursion Analysis is to construct an induction that matches as closely as possible the recursion schemes of as many terms as possible in the goal to be proved. The motivation for this approach comes from an analysis of the role of induction in proofs: induction's role is invariably to eliminate recursive terms from the goal by introducing related instances of those terms in the induction hypotheses. A good choice of induction introduces instances of recursive terms in the hypotheses that can later also be introduced into the conclusion and thus eliminated by hypothesis. These latter term instances are determined by the recursion schemes of the recursive terms in the goal. Thus, the induction schemes to apply are those that match the recursion schemes of the recursive terms in the goal to be proved. The following example illustrates the principle. Consider a proof of:

$\vdash even(x) \leftrightarrow \neg odd(x)$

where *even* and *odd* are defined:

$even(x) \equiv$ if $x = 0$ then $true$ else if $x = 1$ then $false$ else $even(x - 2)$

$odd(x) \equiv$ if $x = 0$ then $false$ else if $x = 1$ then $true$ else $odd(x - 2)$

We induce according to the recursion schemes of *even* and *odd* using the well-founded induction:

$$\frac{\vdash G[x]}{x = 0 \vdash G[x] \quad x = 1 \vdash G[x] \quad x \neq 0, x \neq 1, G[x-2] \vdash G}$$

to give:

$x \neq 0, x \neq 1, even(x-2) \leftrightarrow \neg odd(x-2) \vdash even(x) \leftrightarrow \neg odd(x)$
$x = 1 \vdash even(x) \leftrightarrow \neg odd(x)$
$x = 0 \vdash even(x) \leftrightarrow \neg odd(x)$

apply the recursive definitions of *even* and *odd*

$x \neq 0, x \neq 1, even(x-2) \leftrightarrow \neg odd(x-2) \vdash even(x-2) \leftrightarrow \neg odd(x-2)$
$x = 1 \vdash false \leftrightarrow false$
$x = 0 \vdash true \leftrightarrow true$

Apply hypotheses.
QED.

The recursion analysis procedure is implemented in 4 main parts:

1. **Definition Time Analysis.**

   The first stage occurs whenever a new recursive term is being defined. The term definition's nested structure is first flattened to produce distinct cases implied by its conditional structure from from which a matching induction scheme can readily be extracted. For example, from flattening the recursive definition:

   $$member(x, l) = \text{if } l = [] \text{ then } false \text{ else if } x = hd(l) \text{ then } true \text{ else } member(x, tail(l))$$

   we obtain the recursion equations[4]:

   $l = [] \rightarrow member(x, l) = false$
   $l \neq [] \wedge x = hd(l) \rightarrow member(x, l) = true$
   $l \neq [] \wedge x \neq hd(l) \rightarrow member(x, l) = member(x, tl(l))$

[4]N.B. In some languages such as Prolog recursive definitions are already in this form. In this case the flattening stage can be skipped.

and hence the induction scheme:

$$\frac{G}{\begin{array}{c} l = [] \vdash G \\ l \neq [], x = hd(l) \vdash G \\ l \neq [], x \neq hd(l), G[tl(l)/l] \vdash G \end{array}}$$

The final step is to prove the resulting induction scheme well-founded by finding well-founded measures which decrease under the induction's substitutions, and throw away any pre-conditions irrelevant to well-foundedness. For example, $length(l)$ is such a well-founded measure for the induction scheme above as:

$$0 \leq length(l) \text{ and } l \neq [] \rightarrow length(tl(l)) < length(l)$$

Since $x \neq hd(l)$ is not required for the proof of well-foundedness we discard it to produce the tidied induction scheme:

$$\frac{G}{\begin{array}{c} l = [] \vdash G \\ l \neq [], G[tl(l)/l] \vdash G \end{array}}$$

The combination of the tidied induction and the various well-founded measures is termed the recursive term's *induction template*, and is stored for later use during proofs.

2. The first stage of recursion analysis applied at proof time is template instantiation. The aim of this stage being to collect all the induction schemes analogous to the recursive terms in the goal to be proved to provide the raw material for constructing an induction to suit it.

   The procedure is essentially straight-forward; for each recursive term in the goal, the induction templates are collected and instantiated to reflect the instantiation of the recursive term. If the resulting induction scheme is no longer guaranteed to be well-founded it is discarded. For example the goal

$$x < y \wedge y < z \rightarrow x < z$$

would produce the following induction schemes:

$$\frac{G}{\begin{array}{c} x = [] \vee y = [] \vdash G \\ x \neq [], y \neq [], G[tl(x)/x, tl(y)/y] \vdash G \end{array}} \qquad \frac{G}{\begin{array}{c} z = [] \vee y = [] \vdash G \\ z \neq [], y \neq [], G[tl(z)/z, tl(y)/y] \vdash G \end{array}}$$

$$\frac{G}{\begin{array}{c} x = [] \vee z = [] \vdash G \\ x \neq [] \wedge z \neq [] \wedge G[tl(x)/x, tl(z)/z] \vdash G \end{array}}$$

3. Once all the analogous induction schemes have been collected the next two phases - ***subsumption*** and ***merging*** attempt to combine those that can be applied together as single induction. The aim is to build large inductions that will deal with as many recursive terms in the goal as possible. The basis of the procedure is to pair up induction cases that substitute identically for common variables. If every case of a pair of induction schemes can be paired in this way, the two inductions are replaced a single scheme combining the effects of both. This latter is simply the induction scheme produced by combining the paired cases of the component induction schemes. For example, in the example above, any two induction schemes would combine to give:

$$\frac{G}{\begin{array}{c} x = [] \vee y = [] \vee z = [] \rightarrow G \\ x \neq [], y \neq [], z \neq [] \wedge G[tl(x)/x, tl(y)/y, tl(z)/z] \rightarrow G \end{array}}$$

This would then merge with the third inductions scheme without change, to give an induction combining the effects of all three.

The details of the procedure used to accomplish the merge, and the complications it has to deal with can be found in subsection 3.5.

4. The final step in Recursion Analysis is to choose one of the combined induction schemes for actual application. Two basic criteria are employed:

   The most important criterion is based on a heuristic analysis of the likelihood that an induction will fail. That is, the likelihood that it will leave

recursive terms that cannot be eliminated. Inductions judged likely to fail, in Boyer and Moore's terminology flawed inductions, are discarded. For example, given the goal:

$$append(a, append(b, c)) = append(append(a, b), c)$$

the Boyer-Moore theorem prover would produce two inductions schemes after merging. One scheme would combine the effects of the inductions derived from $append(a, append(b, c))$ and $append(a, b)$. The other would be that derived from $append(b, c)$. The induction based on the recursion scheme of $append(b, c)$ would then be discarded as flawed. The reason is that the induction hypothesis would contain $append(a, tl(b)$ which $append(a, b)$ cannot be rewritten to match. The induction would fail by leaving a recursive term $append(a, b)$ that could not be eliminated.

The second, lesser, criterion for selection is based on a measure of the amount of progress an induction is likely to make if it succeeds. The remaining inductions are given a score combining the number of recursive functions whose recursion schemes they are analogous to, and the closeness of the analogy. The induction with the highest score is the one chosen for actual application.

Why Recursion Analysis?

Given this background, why was Boyer and Moore's recursion analysis technique chosen as the basis for further development? The main strengths of Boyer and Moore's work in this role are that:

- Its empirical performance is excellent. As a component of the BMTP it has been tested on a wide variety of problems and found to perform well, even on the larger more complicated ones.

- The technique is clearly and unambiguously documented at the implementation level ([BM79]), and a runnable version was readily available.

- There is no fundamental reason why the technique could not be adapted to suit almost any system. It is an almost entirely meta-level technique, largely independent of the logic underlying the system it is part of. The changes necessary to adapt it to logics other than Boyer and Moore's "computational logic" are not trivial, but there is no a priori reason why this cannot be done.

As we will see in the following subsections, although many of the alternative approaches make worthy contributions, all compare badly with recursion analysis in one or more of these areas.

2.2.3 Aubin

One of the major pieces of work in the area of automatic proof of inductive theorems is that of Aubin [Aub76]. The overall design of Aubin's system, which is partly based on early work by Boyer and Moore, is very similar to that the BMTP[5]. It too is implemented as a collection of specialist proof procedures that are applied, in sequence, to the any sub-goals remaining to be proved. It also uses a very similar logic based on a pure functional programming language. The similarity extends right up to basic principle on which the selection of appropriate inductions is based. Aubin's system, like Boyer and Moore's, aims to find inductions that enable rewriting to produce as close a match as possible between induction hypotheses and conclusions.

The main difference between the two systems lie in the procedure used to achieve this aim, and in the kind of induction principle supported. Aubin's system only deals with a very restricted form of induction - structural induction. In this form of induction, the induction scheme must be expressed in a form where the conclusion differs from the hypothesis only by the application of constructor

[5] Aubin was student of Boyer

functions to the variables being induced over. For example, the induction scheme we have so far expressed as:

$\vdash G[x]$ (x integer) if: $\vdash G[0]$ and $x \neq 0, G[p(x)] \vdash G[x]$

becomes:
$\vdash G[x]$ (x integer) if: $\vdash G[0]$ and $G[x] \vdash G[s(x)]$

when expressed as an Aubin-style structural induction.

The key advantage of this restricted approach is that it makes it possible for a simple analysis to predict the effect of an induction on a goal. Aubin's induction finding technique exploits this simplification in the method it uses to choose an induction. Instead of focusing on the recursive terms in a goal (see subsection 2.2.2) it concentrates on their arguments. It tries to find the variables which when induced over will permit symbolic evaluation to completely "ripple-out". That is, to enable the functions nesting the variable to be rewritten one after the other until the whole nested structure matches the induction hypothesis. For example consider the goal:

$\vdash even(x + x)$

We induce on x giving:

$x \neq 0, even(x + x) \vdash even(s(x) + s(x))$

In order to match the induction hypothesis with the conclusion, we need to "ripple-out": rewrite the term immediately nesting x $(+)$, in such a way as to enable the rewriting of the term nesting that $(even)$.

$x \neq 0, even(x + x) \vdash even(s(x + s(x)))$

to:

$\neq 0, even(x + x) \vdash even(s(s(x + x)))$

to:

$\neq 0, even(x + x) \vdash even(x + x)$

Fortunately, the variables that might enable rippling-out are actually quite easy to characterise. At each stage in rippling out, a term has to be rewritten into a reduced instance of itself plus other terms enabling the same to be done to the term nesting it. Thus, at each stage, the term's recursive definition (or a lemma similar to it) must be applicable. Thus, Aubin selects for induction the variables at the positions at which the standard call-by-need symbolic evaluation (unfolding) algorithm [MNV71,Vui73] gets "stuck" when applied to the goal to be proved. For example, consider the goal:

$$x + (y + z) = (x + y) + z$$

The call-by-value algorithm will try unfolding $x+(y+z)$ and then $(x+y)+z$. It will immediately get stuck on x in $x + (y + z)$ because to unfold $+$ the first argument must have the form $s(\ldots)$, and x being a variable itself be unfolded to produce this form. In trying to unfold $(x + y) + z$ it will recursively attempt to unfold $x + y$ since $x + y$ does not have the form $s(\ldots)$ necessary to unfold $+$. It will then of course again get stuck on x, as this is neither recursively unfoldable nor of the form $s(\ldots)$. Thus x is the only variable it is necessary to induce over to enable rippling-out in the inductive proof of $x + (y + z) = (x + y) + z$ is x[6].

This, of course, is just the basic principle on which Aubin's approach is based. The actual procedure used by Aubin embeds numerous refinements needed to implement the procedure in practice:

Firstly, it does not, in fact, a collect a single huge list of variables to induce on, as is suggested by the discussion above. Instead variables that need to be induced on together are grouped, and the actual induction applied based on the group most likely to produce a successful induction. The aim is to find an induction likely to succeed, rather than an extravagant effort that is probably doomed to failure. The basis for the grouping is quite straight-forward: variables that

---

[6]This approach was used in the early versions of the BMTP that Aubin based his system on. It was eventually discarded by Boyer and Moore.

need to be induced on together are simply those appearing together as recursive arguments in the same terms. For example, consider the goal:

$$x \leq y + a \wedge a \leq b \rightarrow x \leq y + b$$

where we define $\leq$ by: $x \leq y \equiv$if $x = 0$ then *true*
else if $y = 0$ then *false*
else $p(x) \leq p(y)$

Clearly there is no point inducing on x unless y is induced on too, as to ripple-out both arguments of $\leq$ must have the form $s(\ldots)$. Similarly, there is little point inducing on a unless b is induced on. Thus, in this example, there are two alternative groups of variables that might be induced over: x, y and a, b, each dealing with different parts of the goal.

The procedure used to find the grouping is based on Boyer and Moore's technique. The system begins by collecting as distinct groups the sets of recursion arguments of recursive terms where the symbolic evaluation rule gets stuck. These groups are then exhaustively merged so that any groups sharing variables are combined. The end result is a smaller number of merged groups with the (desired) property that any variables that need to be induced on together appear in the same group.

The final selection of a single merged group of variables as the variables to be induced over is based on the usage of the various groups of variables in the goal to be proved. The system prefers groups of variables with a minimum number of members appearing as non-recursion arguments of recursive terms. This preference is motivated by the "flawing" problem outlined at the end of section 2.2.2. In the event of a tie the group guaranteeing the deepest rippling-out is chosen. If there is still a tie, groups that do not contain variables already induced over are preferred.

Once the variables to be induced over have been chosen, all that remains is to construct the appropriate structural induction scheme. The system begins by computing exactly how much each variable will be decomposed when the

recursive terms it appears in are symbolically evaluated. This information is then used to construct a structural induction over each variable that introduces enough structure to allow all recursive terms they appear in to be symbolically evaluated. The final induction applied is simply the composition of all the individual structural inductions. For example, consider the goal:

$$\vdash even(x) \wedge even(y) \rightarrow even(x + y)$$

The groups of variables that need to be induced over are $\{x\}$ or $\{y\}$. The group $\{y\}$ can however be rejected as it would involve substitution for a variable in a non-recursion argument position: the y in $x+y$. This leaves $\{x\}$ as the set of variables to be induced over. The most x's structure (as a Peano natural) would be decomposed during unfolding is twice (during the unfolding of $even(x)$), thus the induction applied is:

$$\frac{\vdash G[x]}{\vdash G[0/x] \quad \vdash G[s(0)/x] \quad G[x] \vdash G[s(s(x))]}$$

For simple theorems, the performance of Aubin's systems is very competitive with that of Boyer and Moore. It finds the appropriate inductions, for a similar (or perhaps even lesser) amount of computational effort. Unfortunately, for more complicated problems its performance is markedly worse:

- The restriction to structural induction is far from uniformly beneficial. It means that Aubin's system can only express many quite common functions in unnatural or convoluted ways. It is not possible to express recursive functions whose recursive arguments are not reduced in a straight-forward structural way. A typical example is the "quicksort" function for sorting lists:

$$\begin{aligned} qs(l) \equiv\ & \text{if } l = [] \text{ then } [] \\ & \text{else } append(qs(elts_{<}(hd(l), tl(l))), cons(hd(l), qs(elts_{>}(hd(l), tl(l))))) \end{aligned}$$

The overall effect is that Aubin's system is really only useful for simple problems. On complex problems the convoluted style necessary to express many functions actually makes the problem harder not easier to solve.

- The success of an induction may require appropriate substitutions for variables other than those being induced over. For example, in the goal from section 2.1.1

$$\vdash acc\_reverse(l, acc) = append(reverse(l), acc)$$

the variable *acc* needs to be instantiated with $cons(head(l), acc)$ in the induction hypothesis for the induction hypothesis to match the rippled-out induction conclusion.

Unfortunately, unlike Boyer and Moore's system, Aubin's system makes no attempt to prefer inductions for which such substitutions can be found, to those where they cannot.

- The heuristics used in Aubin's system to choose the induction to be applied, are much less accurate than Boyer and Moore's equivalents. For example, Boyer and Moore explicitly check whether substituting for a variable appearing as a non-recursion argument is likely to cause problems. Aubin simply assumes that it will.

Thus, in summary, it would appear that Aubin's work although worthy, has largely been superseded by more recent work by Boyer and Moore. Boyer and Moore's work is thus clearly the better choice as a basis for further work. This is not to say that Aubin's work is irrelevant. In some areas, notably generalisation (which we have not considered) and the analysis of rippling out, Boyer and Moore's system is actually rather less sophisticated that Aubin's.

2.2.4 The Karlsruhe Induction Theorem Proving System

The Karlsruhe system [BHHW88] is a second generation induction theorem prover based largely on the work of Aubin [Aub76] and Boyer and Moore [BM79]. Unfortunately (for the purposes of this thesis) the emphasis of the Karlsruhe researchers is largely on problems other than the selection of appropriate inductions. The system's main innovations are its ability to deal with existentially quantified variables [Biu88], and an extremely efficient procedure for proving the well-foundedness of recursive term definitions [Wal88] and/or induction schemes. Its induction procedure is by contrast relatively unsophisticated:

The main approach used is to apply what the Karlsruhe researchers term the "most nested function" heuristic [Biu88]. The system simply applies the induction scheme that corresponds to the recursion scheme of a most nested function - an *mnf*. That is, a recursive function instance, all of whose arguments contain no recursive function instances. Functions whose recursion arguments[7] are non-variables or are inappropriately quantified for induction are of course ignored. If no suitable most nested function can found, the system simply falls back onto Aubin's procedure. For example, given the goal:

$$\forall x, y. \exists z. (y \leq x \rightarrow z + y = x)$$

the most nested function heuristic will pick the induction corresponding to the recursion scheme of $\leq$ because $y \leq x$ is the only mnf whose recursion argument is a universally quantified variable.

Given its simplicity the mnf-heuristic is remarkably effective. It is cheap to apply, and in a surprisingly large number of situations it produces an excellent choice of induction. It is certainly a useful extension to the Aubin procedure, as it deals with complicated non-structural inductions just as easily as with simple structural inductions.

---

[7] See subsection 1.4

Unfortunately, as with Aubin's procedure, the mnf-heuristic's simplicity and efficiency in simple proofs is bought at the expense of gross inaccuracy in more complicated situations. If there is more than one suitable mnf in a goal, the Karlsruhe system simply makes an arbitrary choice. It, unlike Boyer and Moore's system, makes no attempt to choose on the basis of their relative merits. The end result is that complex goals with multiple most nested functions frequently result in wildly incorrect choices of induction. A further problem with the Karlsruhe system was its status as a relatively recent on-going research project. Information regarding the theoretical motivations for its design was incomplete and relatively light-weight[8].

Thus, despite its more recent design, the Karlsruhe system is probably a poor basis for further work in comparison with Boyer and Moore's.

2.2.5 Kanamori & Fujita - the Argus/V verification system

Kanamori et al's Argus/V verification system [Kan86] is a large scale automatic theorem prover for proving properties of pure Prolog (Horn-clause) programs. For purposes of this discussion, we will however ignore the bulk of its design, and focus only on the component responsible for formulating and applying inductions. This component (due to Kanamori and Fujita [FK86]) is of particular interest because of the induction principle that it implements.

Computational Induction

Instead of the well-founded induction principle we have discussed so far Kanamori and Fujita's system is based on the computational induction principle of de Bakker and Scott [dBS69]:

---

[8]This situation is improving as the authors begin to publish.

$$\frac{W[\text{fix}(f)/x]}{\vdash W[\bot_D/x] \qquad W \vdash W[f(x)/x]}$$

where x inhabits domain D,
$\bot_D$ is the bottom element of D,
f is a continuous function in D,
and fix is the fix-point operator.

This, when reformulated to suit Horn-clause programs, gives the following induction principle for goals containing predicates defined by a set of Horn-clauses:

$$\frac{\forall x_1, \ldots, x_n . r(x_1, \ldots, x_n) \rightarrow W[x_1, \ldots, x_n]}{\forall C_i', \ldots, \forall C_k'}$$

where: r is an n-ary predicate defined by a program made up of clauses $C_1, \ldots, C_k$.

and: $\forall C_i'$ is the universal closures of the clause C_i with all occurences of r replaced by similar occurrence of the formula W. That is, the universal closure of C_i with each atom $r(t_1, \ldots, t_n)$ replaced by $W[t_1/x_1, \ldots, t_n/x_n]$.

For example, from the (pure Prolog) definition for `reverse`

```
reverse( [], [] ).
reverse( [HX|TX], RX ) :- reverse( TX, RTX ), append( RTX, [HX], RX )
```

we obtain the computational induction scheme:

$$\frac{\forall a, b . reverse(a, b) \rightarrow Q[a, b]}{Q[[]/a, []/b]}$$

$$\forall hx, tx, rx, rtx . Q(tx, rtx) \wedge append(rtx, [hx], rx) \rightarrow Q([hx|tx], rx)$$

The main advantage of computational induction compared with the usual well-founded induction principle is that it enables us to deal with partial recursive functions and predicates as easily as ordinary well-founded ones. This is of course crucial if we wish to reason about non-terminating programs such as operating systems or theorem-provers. The following (toy) example due to Apt and van Emden illustrates the benefits:

We assume two predicates p and q defined as

```
p(a) :- p(X),q(X).
p(s(X)) :- p(X).
q(b).
q(s(X)) :- q(X)
```

The execution of $p(s^i(a))$ never terminates and the execution of $p(s^i(b))$ fails finitely. Thus, p never holds. This is of course is quite impossible to prove using well-founded induction since p is not well-founded. However, if we apply computational induction to:

$\forall x.p(x) \rightarrow false$ (which is equivalent to $\forall p.\neg p(x)$)

we obtain $\forall x.false \wedge q(x) \rightarrow false$ and $false \rightarrow false$ both of which are trivially *true*, thus showing that p never holds.

Finding a Computational Induction

As described above, the computational induction principle is of only very limited applicability. An induction on a goal G can only be based on atom R such that G is tautologically equivalent to $R \rightarrow B$ for some formula B, and all R's arguments variables must be universally quantified over the whole goal.

For example, in

$$\forall x, y.x \leq y \wedge plus(x, s(y), z) \rightarrow x < z$$

only $x \leq y$ can be used as the basis of a computational induction. An induction cannot be based on $x < z$ because the goal cannot be rewritten to bring it to the left of an implication dominating the rest of the goal. $plus(x, s(y), z)$ is excluded because one of its arguments is a non-variable.

One of primary achievements of Kanamori and Fujita's system is in relaxing these conditions so that R's arguments can be arbitrary terms containing only variables universally quantified over the whole goal. Their basic idea is to specialise R's program P. For the purposes of the induction, R can be

treated as an atom whose arguments are all variables. For example, we can treat $plus(x, s(y), z)$ in this way if we specialise *plus*'s program

```
plus(0,Y,Y).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z)
```

to

```
plus(0,s(Y),s(Y)).
plus(s(X),s(Y),s(Z)) :- plus(X,s(Y),Z)
```

The resulting computational induction can of course also be specialised in exactly the same way.

The equivalence of R and the old P with R and the new specialised P is ensured by requiring that R is *closed* with respect to P. R is closed with respect to P if, in every clause in P whose head is a ground instance of R, every recursive call is also a ground instance of R. An atom in general form (i.e. with only variables as arguments) is of course trivially closed.

For example, `reverse(X,Y)` is closed with respect to the program:

```
reverse( [], [] ).
reverse( [HX|TX], RX ) :- reverse( TX, RTX ), append( RTX, [HX], RX )
```

whereas `reverse(X,[HY|TY])` is not because if `TY` were instantiated to `[]` the recursive call to `reverse` would not be an instance of this atom.

The first step in Kanamori and Fujita's procedure is thus to identify the atoms with the correct context and variable quantification to act as bases for computational induction. These are termed "dominant" atoms in the goal. Any dominant atoms that are not closed[9]. are then rewritten into more general forms

---

[9]The details of the matching procedure for detecting non-closure and the procedure for finding atoms with the correct context may be found on pp.285,287 of [FK86].

that are closed: they are replaced by equivalent conjunctions of equalities and more general instances of themselves, so that all contextually suitable atoms in the goal can in fact be used as the basis for computational inductions. For example, the non closed atom `reverse(X,[HY|TY])` would be replaced by the conjunction

```
reverse(X,Y),Y=[HY|TY].
```

The procedure is not, however, as might be expected, simply to use the trivially closed instance of the atom in general form. Instead considerable effort is expended to find the least general atom necessary to close the atom with respect to its defining program[10]. Unfortunately, the reason why all this complication is favoured over rewriting all atoms into predicates in general form conjoined with equalities is not explained in the [FK86,Kan86].

Once all dominant atoms have been closed the actual construction of the induction scheme proper begins. The procedure used is to collect all dominant atoms and to exhaustively "merge" them, much as Boyer and Moore merge induction schemes. The "merge" in this case consists of replacing pairs of dominant atoms with a single atom equivalent to their conjunction that provides the basis of a single computational induction. We illustrate the procedure with an example due to Kanamori and Fujita - the "merge" of `reverse(N,L)` and `append(N,[X],M)`. We assume the predicates are defined as:

```
append([],L,L).
append([H—T],A,[H—B]) :- append(T,A,B).

reverse([],[]).
reverse([H—T],R) :- reverse(T,A),append(A,[H],R).
```

The merging procedure is as follows:

---

[10]The existence of a unique minimal generalisation can be proved.

1. The first step is to check that attempting the merge of the two atoms is worthwhile. The atom's must have some variables in common, and at least one pair of clauses from their defining programs must treat the shared variables identically. In our illustrative example, the merge of `reverse(N,L)` and `append(N,[X],M)`, the shared variable is N, which is is treated identically by both atoms' definitions.

2. A new recursive predicate over the union of the variables mentioned in the two atoms, equivalent to their conjunction, is synthesised using Tamaki and Sato's version of the fold/unfold procedure [Tam84].

   (a) A new predicate over the union of the variables in the atoms is defined as the conjunction of the two atoms.

   ```
   new-p(L,M,N,X) :- reverse(N,L), append(N,[X],M).
   ```

   (b) Each atom in the resulting clauses' tail is unfolded once to give an expanded, but equivalent definition for the replacement predicate. In our example the first unfold gives:

   ```
   new-p([],M,[],X)     :- append([],[X],M).
   new-p(L,M,[H'|T'],X ):- reverse(T',A'),
                           append(A',[H'],L),append([H'|T'],[X],M).
   ```

   which, after the second unfold becomes:

   ```
   new-p([],[X],[],X).
   new-p(L,[H'|T'],[H'|B'],X) :- reverse(T',A'),
                           append(A',[H'],L),append(T',[X],B').
   ```

   (c) Tamaki and Sato's procedure ends with the exhaustive application of the fold transformation in the tails of the expanded definition's clauses. The merging criterion guarantees that at least one fold will occur using the original un-expanded definition of the replacement predicate. The end result is guaranteed to be a new recursive replacement predicate that provides the basis for a computational induction that can replace those of the two original atoms.

In our example, we fold append and reverse using the original unexpanded definition for new-p to give:

```
new-p([],[X],[],X).
new-p(L,[H'|T'],[H'|B'],X) :- new-p(A',B',T',X),append(A',[H'],L)
```

3. When the clauses defining the new predicate recursively have been constructed the new predicate replaces the conjunction of the two merged atoms in the goal to be proved. In our example `new-p(L,M,N,X)` replaces `reverse(N,L)` and `append(N,[X],M)`.

Once the dominant atoms in the goal have been merged as far as is possible, Kanamori and Fujita simply choose one at random as the basis for the induction to be applied.

An Evaluation

Kanamori and Fujita's work clearly makes an important contribution in that it provides an powerful induction principle suitable for automatic application in the verification of logic programs. Unfortunately, the actual application of this induction principle is relatively naive. The procedure it implements is, to all intents an purposes, merely a simplified version of the Boyer-Moore procedure adapted for dealing with pure Prolog rather than functional programs. Specifically:

- The Kanamori-Fujita procedure, like recursion analysis, bases the induction schemes it builds on the recursion schemes of the terms present in the goal to be proved. However, unlike recursion analysis no attempt is made to consider the goal as a whole. Only the dominant atoms figure in the choice of induction scheme. If these are not typical of the goal as a whole quite the wrong induction will be chosen, or even no induction found. For example, Kanamori's system will be unable to find a sensible induction for:

$$sum(1, x, s) \rightarrow (times(x, s(x), s_2) \wedge plus(s, s, s_2))$$

where *sum* is defined such that $sum(a, b, s) \leftrightarrow \Sigma_{i=a}^{b} i = s$

Boyer and Moore's system will easily pick the right induction, that based on *times*.

- The final stage of the procedure - the selection of the merged induction that is actually to be applied - is completely random in Kanamori and Fujita's system. Inductions highly likely to fail are just as likely to be chosen as those almost certain to succeed. For example, consider the goal:

$$sum(x, xy, z) \wedge plus(y, x, xy) \rightarrow plus(z, z, z_2 \wedge times(xy, s(xy), pxy) \wedge times(x, s(x), px) \wedge subtract(pxy, xy, z_2)$$

One choice of induction (that based on *sum*) is quite disastrous as it has quite inappropriate side-effect on the x in *plus*. The alternative (based on *plus*) goes through quite easily as it has no such inappropriate side-effects. Kanamori and Fujita's system will simply make a random choice and hope for the best.

In contrast the Boyer-Moore procedure will reliably choose the correct induction. A check is made to eliminate inductions likely to fail because of side-effects, and a scoring system applied to choose the induction likely to make the most progress if successful.

Clearly, as with the Karlsruhe system, Kanamori and Fujita's work, though worthy in its own right, is a poor choice as the basis for further work in comparison with Boyer and Moore's.

2.2.6 Inductionless Induction

In recent years a considerable literature has built up relating to term rewriting techniques known as "inductionless induction". These techniques - also known as inductive completion procedures - allow inductive theorems to be proved without the need to explicitly construct and apply an appropriate induction scheme.

Their theoretical basis in the theory of universal algebra is, however, rather different from that of the other techniques discussed in this chapter. We must therefore precede discussion of the technique itself with the introduction of some appropriate terminology [11].

Definitions

We assume throughout that F is a finite set of function symbols, and that V is a countably infinite set of variables.

A universal algebra $\mathrm{A}(F \cup V)$ is the set of terms we can build from F and V.

An equational theory is a set of equations $t_1 = t_2$ with $t_1, t_2 \in \mathrm{A}(F \cup V)$, we write the equivalence relation induced by an equational theory E as $=_E$.

An inductive theorem of an equational theory is an equation $M = N$ such that $M\phi =_E N\phi$ for all substitutions ϕ such that $N\phi$ and $M\phi$ are ground $(M\phi, N\phi \in AF)$.

A rewrite rule is a pair of terms $l \rightarrow r$.

A rewriting system is a set of rewrite rules. Given a rewriting system R we write $t \rightarrow_R s$ if and only if

$\exists t'$ a subterm of t, ϕ a substitution, and a rewrite rule $l \rightarrow r \in R$ such that $t[r\phi/t'] = s$ and $t' = l\phi$.

We write the reflexive, symmetric and transitive closure of the reduction relation $\rightarrow_R$ as $=_R$.

A rewriting system is said to be *finite-terminating* if there exists no infinite sequence of rewritings. That is if there is exists no sequence of terms t_i such that:

$t_1 \rightarrow t_2 \rightarrow \ldots \rightarrow t_n \rightarrow \ldots$

---

[11] We base this section and the next on an excellent brief introduction to inductive completion presented by Bill Mitchell of Manchester University at [Mit88]

The *normal forms* of a term under a finite-terminating rewriting system are the the terms it can be rewritten to that cannot themselves be further rewritten.

A finite-terminating rewriting system is *complete* if and only if all the normal forms of any term are identical. The common normal form of a term is called its *canonical form.*

We say a complete rewriting system R is equivalent to an equational theory E if and only if the relation $=_R$ is equivalent to the equivalence relation $=_E$ induced by E.

An Inductionless Induction Proof Procedure

In contrast with the work we have discussed so far, Inductionless induction is not itself a specific proof technique. It is rather, an particular approach to automatic theorem proving that can be implemented in several different ways. It is based on the application of the following theorem relating inductive theorems to rewriting systems:

If R is a rewriting system equivalent to $R \cup \{M = N\}$

then $M = N$ is an inductive theorem of $R \leftrightarrow \neg\exists t_1, t_2 . t_1 =_R t_2 \wedge \neg t_1 =_R t_2$

This theorem allows a conventional (possibly inductive) proof of $M = N$ as an equational theory R to be replaced by the construction of a complete rewriting system R for $M = N \cup R$ and a test on R's properties. The first step - the construction of R is relatively straight forward. The Knuth-Bendix completion procedure [KB70] provides an automatic procedure for generating rewriting systems equivalent to equational theories. Given a set of equations and an ordering on terms, the Knuth-Bendix procedure systematically enumerates the rewrite rules needed for an equivalent complete rewriting system. If the procedure terminates (which it need not) the rewrite rules generated form a complete rewriting system equivalent to the equational theory input.

The difficult part, on which the various inductionless induction methods differ, lies in setting up R in such a way that the conditions (*) can be efficiently recognised mechanically. A fairly typical approach is that taken by Huet and Hullot in [HH82]. They make the problem of recognising when R has the (*) property tractable by restricting the form of the equational theories they consider. The restriction they apply is that the equational theory R satisfies the "principle of definition". This requires that there is a proper subset C of the functions F in R's term algebra $\mathrm{A}(F \cup V)$ such that

$\forall f \in \mathrm{A}(F) \exists b \in AC.f =_R b \wedge \neg \exists b' \in AC.b' \neq b \wedge b' =_R b$ holds.

That is, the functions used must have designated subset of constructor functions C such that every ground term of R's term algebra is equal to a unique term made up only of constructors from C.

This restriction on R means that $M = N$ can be shown to be an inductive theorem of R when R has the easily tested property that $\forall l \rightarrow r \in \mathrm{R}.l \notin \mathrm{A}(C \cup V)$. It also allows $M = N$ to be shown *not* to be an inductive theorem of R whenever R contains a rewrite rule $l \rightarrow r$ where $l, r \in \mathrm{A}(C \cup V)$. Huet and Hullot use these properties of equational theories satisfying the principle of definition to construct the proof-procedure below:

1. Apply Knuth-Bendix algorithm:

   IF no further rewrite rule is needed to complete R
   THEN STOP "$M = N$ is a theorem of R"
   ELSE generate rewrite rule $l \rightarrow r$ needed.

2. IF $l, r \in \mathrm{A}(C \cup V)$
   THEN STOP "$M = N$ is not a theorem of R"

3. IF $l \in \mathrm{A}(C \cup V)$ and $\neg r \in \mathrm{A}(C \cup V)$
   THEN STOP "Give up - cannot decide whether $M = N$ is a theorem of R"

4. GOTO 1

Some other significant implementations are those of [Toy86], [Jou86], [Gog80], [KNZ86].

An Evaluation

Unfortunately, despite its meta-mathematical elegance, the inductionless induction approach is probably of little relevance to a general-purpose theorem prover. The most crucial limitation is both obvious and inherent: inductionless induction is only applicable to equational theories. Neither variable binding constructs nor theorems with pre-conditions can be adequately treated in an equational setting. Thus, a lot of interesting domains are simply outside the capability of the technique. In actual implementations of inductionless induction things are usually even worse.

Even if inductionless induction techniques can be applied, they typically suffer a lack of completeness. A poor choice of term order can make the Knuth-Bendix algorithm fail to terminate even if the inductive theorem being proved is true. The mechanical construction of useful term orderings is still an open research problem. This lack of completeness compares most unfavourably with conventional uniform theorem provers based on conventional resolution techniques.

A further point is that, contrary to what might be supposed, the method does not actually avoid induction. The proof of the usual induction sub-goals are merely hidden within the Knuth-Bendix procedure as problems of rewriting rewrite rules into canonical form. The choice of a particular induction is effectively hard-wired into the inductionless induction procedure used. The majority of inductionless induction methods can thus only a few very simple induction schemes.

Finally, as with any uniform theorem proving approach, inductionless induction methods make it very difficult to control and/or analyse proofs. It would be really very difficult to apply higher level search control knowledge in an inductionless induction based theorem prover. Analysing and patching around

failures, or learning control knowledge would be similarly difficult. As such it is to be expected that on larger problems it would be difficult to avoid a combinatorial explosion in theorem-provers based on inductionless induction.

Thus, in summary, inductionless induction proof procedures suffer from serious proof-theoretic and search-control weaknesses. The main flaws are that they are incomplete, suit only weak logics, and are difficult to scale up to suit large problems. None of these fundamental flaws apply to a comparable extent to Boyer and Moore's approach to inductive theorem proving, which therefore provides a superior basis for further work in the area.

2.3 Summary

In this chapter we have sought to address two main issues. Firstly, the reason why the automatic construction of induction schemata is a subject worthy of investigation, and secondly why we chose to base our investigations on the work of Boyer and Moore.

We have seen that the primary motivation for the automation of inductive proofs lies in the need to ensure the correctness of software. The various methodologies for the development of formally correct programs invariably require numerous, often lengthy, inductive proofs. The particular form the required proofs take varies from methodology to methodology, but the need for them does not. Since the informal manual construction of these proofs is both unreliable and expensive, it is clearly worth investigating their mechanical construction. Hence, as successful inductive proofs depend on the use of an appropriate induction schema, the automatic construction of (appropriate) induction schemata is a topic worthy of research.

The motivation for the use of Boyer and Moore's work as the basis for this thesis is, as we have seen, equally straight-forward. The bulk of existing research into the mechanical construction of appropriate induction schemata is subsumed by Boyer and Moore's work. Specifically:

- The work of Aubin is based on early work by Boyer and Moore, and has been almost entirely superceded by more recent work by Boyer and Moore. It is, in any case, extremely restricted as to the types of programs it can be used to prove properties of.

- The Karlsruhe induction theorem proving system, although superior to Boyer and Moore's system in other respects, constructs induction using a reimplementation of the (superceded) technique developed by Aubin.

- The Argus/V verification system developed by Kanamori et al, is based an induction principle rather different from that used by Boyer and Moore. The procedure by which it constructs induction schemata is, however, merely a partial implementation of that developed by Boyer and Moore.

The only other major body of research into mechanising inductive proofs is that based on inductive completion procedures. Inductive completion based theorem proving system, however, have several inherent drawbacks when compared with Boyer and Moore's technique:

- They are restricted only to very weak, equational, logics.

- Since they are based on brute-force search techniques, their performance is likely to deteriorate rapidly when larger problems are tackled.

- They can "apply" only a very limited range of induction schemata.

Chapter 3

RECURSION ANALYSIS

3.1 Introduction

A pre-condition of improving any existing technique is a detailed understanding of its working and the theoretical ideas that motivate it. Unfortunately, although Boyer and Moore document the first aspect of Recursion Analysis excellently, their treatment of the second is quite perfunctory. They make relatively little attempt to motivate and justify Recursion Analysis in terms of a coherent theoretical framework. It is specifically this lack of a theoretical framework that we address in this chapter.

We begin by constructing an overall framework for Boyer and Moore's approach in terms of the meta-theory of induction. Sections (3.3 to 3.6) then deal with each of the components of Boyer and Moore's design in turn. In each case, the theoretical framework is first elaborated to motivate the component's presence, and then used to (as far as possible) reconstruct and justify Boyer and Moore's design. The use of our theoretical framework to build an enhanced, rational reconstruction of Recursion Analysis we leave until chapter 4.

3.2 A Theory of Recursion Analysis

If we are to understand why Recursion Analysis (often) succeeds in finding an induction appropriate to a particular goal we must first develop a reasonable understanding as what it is that makes an induction appropriate. Surprisingly perhaps, formal proof-theory provides little help in this area. A proof theoretic characterisation that an induction is appropriate if it is part of a proof (if one exists) says nothing about how appropriate inductions might be found. Worse, it cannot even be decided without completing a proof. Fortunately, by considering the informal *meta-theory* of induction we can do rather better.

Figure 3–1: Induction Eliminating Recursive functions

If $+$ has the recursive definition: $x + y =$ if $x = 0$ then y else $s(p(x) + y)$ we prove $x + 0 = x$ as follows:

$$\dfrac{\vdash x + 0 = x}{\dfrac{x = 0 \vdash x + 0 = x}{\dfrac{x = 0 \vdash 0 = x}{\square}} \qquad \dfrac{x \neq 0, p(x) + 0 = p(x) \vdash x + 0 = x}{\dfrac{x \neq 0, p(x) + 0 = p(x) \vdash s(p(x) + 0) = x}{\dfrac{x \neq 0, p(x) + 0 = p(x) \vdash s(p(x)) = x}{\square}}}$$

The key point is that the use of induction in a proof is almost invariably correlated with the proof steps that eliminate recursive functions [1]. This correlation arises directly from the nature of proofs involving defined functions: to minimise their complexity it is desirable that they are proved at as high a level of abstraction as possible. That is, that the defined functions are expanded as little as possible. Nevertheless, quite often a point is reached where further progress is blocked until defined functions are expanded in favour of the simpler functions

[1] Strictly speaking literals containing recursive functions - more on this point later in this section.

used to define them. In the case of non-recursively defined functions the task is simple. The functions can be unfolded (see [BD77]) and replaced by their (suitably instantiated) definitions.

Unfortunately, this is cannot be done with recursively defined functions. The definitions of such functions, by definition, contain instances of themselves. Simply unfolding such functions would thus merely change the recursive functions present rather than eliminating them as desired. An induction, however, if correctly chosen *can* eliminate recursive functions. In induction step cases the induction hypotheses provide reduced instances of the recursive functions to be eliminated. Given a good choice of induction the application of lemmas or symbolic evaluation can allow these to be used to eliminate the recursive functions in the conclusion by hypothesis. Similarly, in the base cases of an induction the recursive functions are applied to objects that are in some sense minimal. If the induction is well-chosen, this can allow the recursive functions to be eliminated by applying the base cases of their definitions (or similar lemmas). Thus, the overall effect of a well-chosen induction is to eliminate recursive functions, replacing them with functions present in their recursive definitions or related lemmas.

Figure 3–1 illustrates these effects with a simple example. The induction eliminates $+$ in the step case by introducing an instance of $+$ that exactly matches that in the conclusion after the application of unfolding. $+$ is eliminated in the base-case because the hypothesis that $x = 0$ allows the base case of its definition $0 + x = x$ to be applied.

Given this meta-theoretic characterisation of induction, a workable characterisation of appropriate inductions becomes trivial. We say an induction is appropriate if and only if it enables the elimination of at least one recursive function in the goal it is to be applied to. This characterisation *can* be tested without completing the whole proof in advance.

How then does Recursion Analysis construct induction schemata likely to satisfy this characterisation of appropriateness? The answer is that it makes use of the more fundamental relationship between induction and recursion that

underlies the informal connection outlined above. Induction and recursion are duals - an induction schema is a recursively defined proof schema. Hence, the recursion schema used in a function's recursive definition defines (in Boyer and Moore's terminology "suggests") an analogous dual induction schema and vice versa. Figure 3–2 illustrates the relationship with a simple example.

Figure 3–2: Dual Induction Schema of "+"

Definition		Dual Induction Schema
$x + y =$	if $x = 0$	$\vdash G$
	then y	$x = 0 \vdash G$
	else $s(p(x) + y)$	$x \neq 0, G[p(x)/x] \vdash G$

where G is an arbitrary sentence of the logic used.

Formally, we define the induction *dual* to a recursive function (and vice-versa) as follows:

Definition 9 Dual Induction/Recursion

An induction schema is *dual* to a recursive function if there is a one to one correspondence between the induction's cases and the cases of the function's definition such that:

The conditions governing each induction case are identical to the conditions governing the corresponding case of the function and each step-case's step-substitutions are identical to the substitutions for the function's formal arguments in the recursive calls present in the corresponding case of the function definition.

■

The significance of this induction/recursion duality lies in the effects dual inductions produce when they are applied to goals containing a function they are dual to. In this situation the application of symbolic evaluation is guaranteed to eliminate the function in the base-cases and produce instances of the function

in the induction conclusion matching those in the induction hypotheses. Consider the base-cases: the identity between induction base-cases and non-recursive cases of the function's definition means that symbolic evaluation must expand the function using one of its non-recursive cases. Since the expansion cannot, by definition, re-introduce the function, the function is eliminated. The argument in a step-case is very similar: the identity between step-cases and recursive cases means that the instantiation of the induction hypotheses matches that of the recursive calls in the case of the function's definition applicable to the induction conclusion. Thus, trivially, the instances of the function in the induction conclusion after symbolic evaluation must match those in the induction hypotheses. This, heuristically at least, implies that the function can be eliminated from the proof by hypothesis. Figure 3–3, illustrates the principle with a (typical) proof where a goal about two recursively defined functions $\leq$ and *eq* is proved by application of their dual induction.

Figure 3–3: The application of Dual Induction

GIVEN:

$x \leq y \equiv$ if $x = 0$ then *true*
else if $y = 0$ then *false*
else x-1 $\leq$ y-1

AND: $eq(x,y) \equiv$ if $x = 0$ then $y = 0$
else if $y = 0$ then *false*
else $eq(x\text{-}1, y\text{-}1)$

PROVE: $eq(x,y) \rightarrow x \leq y$

We induce stepping down 1 on x and y:
$\neg x = 0,\ \neg y = 0,\ (eq(x\text{-}1, y\text{-}1) \rightarrow (x\text{-}1 \leq y\text{-}1)\ \vdash\ (eq(x,y) \rightarrow x \leq y)$
$x = 0,\ \ \neg y = 0\ \vdash eq(x,y) \rightarrow x \leq y$
$x = 0,\ \ y = 0\ \ \vdash eq(x,y) \rightarrow x \leq y$
$\neg x = 0,\ y = 0\ \ \vdash eq(x,y) \rightarrow x \leq y$

Symbolic evaluation of the conclusion gives:
$\neg x = 0,\ \neg y = 0,\ (eq(x\text{-}1, y\text{-}1) \rightarrow x\text{-}1 \leq y\text{-}1) \vdash (eq(x\text{-}1, y\text{-}1) \rightarrow x\text{-}1 \leq y\text{-}1)$
$x = 0,\ \ \neg y = 0\ \vdash$ *false* $\rightarrow$ *true*
$x = 0,\ \ y = 0\ \ \vdash$ *true* $\rightarrow$ *true*
$\neg x = 0,\ y = 0\ \ \vdash$ *false* $\rightarrow$ *false*

All of which eventually reduce to $\vdash$.

The elimination of the function in the step-case is only heuristic largely because it might be nested inside other functions. In this case - as we suggested in the footnote above - the whole of the literal containing the recursive function needs to be eliminated. The expansion of the recursive function has to start a "rippling-out" [2] effect whereby the rewriting of one function to match the induction hypothesis enables the rewriting of the function surrounding it until the whole literal matches. This rippling out cannot, in general, be guaranteed. Nevertheless, although by no means universal (see section 4.6.2), rippling-out seems to occur often enough for dual inductions to provide a very useful basis for an automated induction prover.

Similarly, if the function on which the dual is based itself has functions nested within it, it may only be possible construct an approximate dual. In this case also the elimination of the function cannot be guaranteed. It may not be possible to rewrite the induction conclusion and hypotheses so that the partial match produced by the approximate dual becomes exact. A typical example of a situation in which only an approximate dual can be constructed is the goal:

$$\vdash x \leq x + y$$

We cannot provide an exact dual for $\leq$ because we cannot suitably instantiate its second argument in the induction hypothesis as this argument is a non-variable.

Recursion Analysis is thus simply a procedure for exploiting the useful property of dual inductions outlined above in practice. It exhaustively combines the dual inductions of the recursive function instances in a goal to produce induction schemata likely to eliminate as many as of these recursive functions as possible. If there is a choice between different combinations, it attempts to pick the combination heuristically least likely to fail. The following sections of this chapter detail exactly how it goes about this task and copes with the many complications that arise.

---

[2] We borrow this terminology from [Aub75]

3.3 Definition Time Analysis

Unsurprisingly, the first step in the recursion analysis algorithm is to construct the dual inductions of the goal to be proved. That is, to construct the induction schemata dual to the recursively defined functions in the goal. The construction procedure in fact falls into two distinct parts: the initial derivation of the well-founded inductions dual to the recursive functions defined in the theorem-prover (*induction templates* in Boyer and Moore's terminology); and the instantiation of these duals to reflect the particular instances of the functions present. In this section we focus entirely on the first part. We leave the second part to section 3.4.

The title of this section reflects the fact that induction template derivation occurs when recursive terms are defined rather than during proofs when recursion analysis proper is applied. The reason for this distinction is simple: a function's induction templates are the same in every proof. It makes sense to construct them in advance and record them, rather than rebuild them every time recursion analysis is applied.

3.3.1 Constructing a Function's Dual Induction

A dual induction schema bears a very direct relation to the function definition it is dual to. The conditions governing its cases are the same as those governing the branches of the definition it is dual to. The induction hypotheses for each case of the induction are given by the recursive calls in the corresponding branch of the definition. Each distinct recursive call corresponds to a step-substitution (see 3) that substitutes the induction variables in the same way that the recursive call instantiates the definition's formal arguments. The construction of the inductions dual to a simple recursive definition is therefore, in principle, quite straight forward. All that is required is to recursively traverse the definition's conditional structure to find its conditional-free branches. For each branch found

a case of the dual induction can be built from the recursive calls present in the branch and the conditions governing it. The *member* function and its dual listed below illustrate a simple example of this construction

$$
\begin{array}{ll}
member(x, l) \equiv & \qquad \vdash G \\
\quad \text{if } l = nil \text{ then } false & \qquad l = nil \vdash G \\
\quad \text{else if } hd(l) = x \text{ then } true & \qquad \neg l = nil, hd(l) = x \vdash G \\
\quad \text{else } member(x, tl(x)) & \qquad \neg l = nil, \neg hd(l) = x, G[tl(l)/l] \vdash G
\end{array}
$$

Figure 3–4 gives Boyer and Moore's algorithm for constructing dual inductions (pp.165 of [BM79]).

It is important to note that this algorithm explicitly leaves out the construction of the base-cases of the dual induction. The motivation for this apparently perverse omission is to simplify the implementation of the recursion analysis procedure. Several important parts of the algorithm are significantly simpler if the base cases for inductions are omitted and only reconstructed after recursion analysis is complete.

The reconstruction of the base-cases relies on the fact that the induction's case-analysis is the flattened equivalent of a function's nested conditional structure. The cases arising from flattening one branch of a conditional all have a governing condition that appears negated in the cases arising from the other branch. For example, in the dual induction of *member* shown above $l = nil$ is un-negated only in the first base-case, and $hd(l) = x$ is un-negated only in the step-case. This property of the property of the conditions governing the inductions cases means that the induction case-analysis is disjoint as well as complete[3]. That is, if S is the set of cases in the induction and C_i is the governing condition of case i, then

$\forall k \in S. C_k \leftrightarrow \neg \bigvee_{i \in S-\{k\}} c_i$ and $\bigvee_{i \in S} c_i$ hold.

This allows us to deduce that if B is the subset of base-cases in S then

[3]This of course assumes that the conditions contain only total functions.

Figure 3–4: Algorithm to Build Dual Induction

1. *branch* := body of function's definition

   governing := *nil*

   stepcases := $\emptyset$

2. CALL findcases(*branch*, *governing*)

3. EXIT step cases of dual induction = *stepcases*

4. PROCEDURE findcases(*branch*, *governing*)

   IF *branch* = if *cond* then *thencase* else *elsecase* AND
   there are no recursive references in *cond* AND
   recursive references in *thencase* $\not\subseteq$ those in *elsecase* AND
   recursive references in *elsecase* $\not\subseteq$ those in *thencase*
   THEN
   findcases(*thencase*, *cons*(*cond*, *governing*))
   findcases(*elsecase*, *cons*($\neg$*cond*, *governing*))
   ELSE
   IF there are no recursive references in *branch*
   THEN RETURN
   ELSE
   recrefs := recursive references in *branch*
   stepsubs := instantiations for formal arguments in *recrefs*
   stepcases := *stepcases* $\cup$ step-case (*governing*, *stepsubs*)
   RETURN

$\bigvee_{j \in B} c_j \leftrightarrow \neg \bigvee_{i \in S \setminus B} c_i$.

So that we can reconstruct the base-cases of the induction schema as a single combined base case with the governing condition $\neg \bigvee_{i \in S \setminus B} c_i$. For example, the dual induction of *member* outlined above is represented in the recursion analysis procedure by its step-case:

$$\neg l = nil, \neg hd(l) = x, G[tl(l)/l] \vdash G.$$

When the induction is applied the base-cases are reconstructed as a single combined base-case to give the induction schema:

$$\frac{\vdash G}{\begin{array}{c}\neg(\neg l = nil \wedge \neg hd(l) = x) \vdash G \\ \neg l = nil, \neg hd(l) = x, G[tl(l)/l] \vdash G\end{array}}$$

In the remainder of this chapter we will assume this reconstruction procedure and leave the base cases of induction schemata implicit. Some further points about this algorithm which we will return to in later chapters are that:

1. It cannot deal with conditional structures nested inside the arguments of ordinary functions.

2. It never produces governing conditions with embedded recursive references.

3.3.2 Proving Well-foundedness

Once the induction dual to a function definition has been extracted, the next step in recursion analysis is to prove it well-founded. The system searches for a tuple of measures on a subset of the variables induced over which is reduced under all of the induction's step-substitutions. The potentially enormous search-space is pruned by the use of a special class of user-formulated lemmas - *induction lemmas* - which express the known instantiations of measures which can be shown reduced under well-founded orderings. Figure 3–5 gives several typical examples of induction lemmas. Instead of exhaustively trying out different measures in (expensive) attempted well-foundedness proofs, the BMTP merely

collects the induction lemma instances which could be used to prove that the induction's step-substitutions reduce a well-founded measure. That is, for each step-substitution the system collects the induction lemma instances whose conclusions show some well-founded measure is reduced by the step-substitution, and whose pre-conditions follow from the conditions governing the step-substitution. For example, consider the dual induction of the function *merge*, defined by:

$$\begin{aligned} merge(l,r) \equiv\ & \text{if } l = nil \text{ then } r \\ & \text{else if } r = nil \text{ then } l \\ & \text{else if } hd(l) < hd(r) \text{ then } cons(hd(l), merge(tl(l), r)) \\ & \text{else } cons(hd(r), merge(l, tl(r))) \end{aligned}$$

which has the form

$$\frac{\vdash G}{\begin{array}{l} \neg l = nil, \neg r = nil, hd(l) < hd(r), G[tl(l)/l, r/r] \vdash G \\ \neg l = nil, \neg r = nil, \neg hd(l) < hd(r), G[l/l, tl(r)/r] \vdash G \end{array}}$$

Given the induction lemmas in figure 3–5 the system would collect the induction lemma instance $length\_tl[l/l]$ for the first step-substitution and $length\_tl[r/l]$ for the second. These would be shown to apply under the conditions governing the step-substitutions by proving that:

$$\neg l = nil \wedge \neg r = nil \wedge hd(l) < hd(r) \rightarrow \neg l = nil$$

and

$$\neg l = nil \wedge \neg r = nil \wedge \neg hd(l) < hd(r) \rightarrow \neg r = nil$$

Once the induction lemmas have been collected tuples of measures from which proofs of well-foundedness can be constructed can be found very efficiently. The induction lemmas associated with each step-substitution give the measures it reduces under a well-founded ordering. A tuple of measures therefore shows the induction well-founded under a lexicographic ordering if, for each step-substitution:

1. There exists a leftmost member of the tuple appearing in one of the induction lemmas collected for the step-substitution.

2. The measures appearing left of the reduced measure in the tuple are unchanged by the step-substitution.

For example, the induction dual to *merge* can be proved well-founded under the pair of measures $(length(l), length(r))$.

The tuples of measures to try out are systematically generated in order of ascending size. First single measures mentioned in the collected induction lemmas are tried out, then pairs, then triples and so on. An important refinement that prevents excessive search is that tuples whose measures share variables are excluded. Such tuples are highly unlikely to show the induction well-founded since the sharing of variables precludes one measure changing whilst others are unchanged.

Figure 3–5: Induction Lemmas

1. $length\_tl == \neg l = nil \rightarrow length(tl(l)) < length(l)$
2. $length\_1stHalf == \neg l = nil \rightarrow length(1stHalf(l)) < length(l)$
3. $length\_2ndHalf == \neg l = nil \rightarrow length(2ndHalf(l)) < length(l)$

Boyer and Moore's efficient procedures that realise this scheme for proving well-foundedness with a minimum of search may be found on pp.180-183 of [BM79]. A point worth noting is that a proof of well-foundedness for a function's dual induction also provides a proof of well-foundedness for its definition. Boyer and Moore exploit this useful fact so as to avoid explicit proofs of well-foundedness for newly defined functions. Well-foundedness is instead ensured by accepting only function definitions whose dual inductions can be proved well-founded as outlined above.

3.3.3 Eliminating Redundancy

Once a function's dual induction has been proved well-founded the final definition-time step is to tidy it up by removing redundant tests from its case-analysis. Redundant tests are governing conditions in a case of an induction that can be removed without material effect on the induction's form (the induction hypotheses it builds) or well-foundedness. The case-analyses of dual inductions tend to contain redundant tests because, in practice, only part of a function definition's conditional structure relates directly to its recursion schema. The remainder is specific to the particular computation the function implements, and produces redundant case-analysis conditions in its dual induction schema. For example, the conditional structure of the definition for *member* tests whether $x = hd(l)$. This test, although it is needed to make *member* compute the right value, is completely irrelevant to *member*'s recursion schema. As a result $x = hd(l)$ appears as a redundant test in the case-analysis for *member*'s dual induction schema:

$$\frac{\vdash G}{\neg l = nil, \neg x = hd(l), G[l/tl(l)] \vdash G}$$

The governing condition $\neg x = hd(l)$ is completely extraneous to the induction schema and can be removed without materially changing it.

Redundant tests need to be removed because they cause additional "weak" base cases in the induction. If C appears as a redundant test in the hypotheses to the step-cases of an induction, we must prove a base-case under the hypothesis that $\neg C$ holds. Such base-cases can be very hard to prove since C is, by definition, completely irrelevant to the induction. Boyer and Moore give a good example of this effect in pp.176-177 of [BM79]. Furthermore, redundant tests can also obscure situations in which several different functions have the same dual induction(s). This can cause a great deal of effort to be wasted in considering trivial variants of the same basic induction.

Boyer and Moore eliminate redundant conditions by replacing the original governing conditions of an induction's cases with the pre-conditions of the induction lemma used to prove that case well-founded. The idea is that since

the well-foundedness proof requires the pre-conditions of the induction lemmas to be shown, a case-analysis composed of these pre-conditions is, by definition, non-redundant[4]. For example, the induction schema dual to *merge* which was proved well-founded above would be rewritten to give:

$$\frac{\vdash G}{\begin{array}{l}\neg l = nil, G[tl(l)/l, r/r] \vdash G \\ \neg r = nil, G[l/l, tl(r)/r] \vdash G\end{array}}$$

Induction cases whose governing conditions become identical after tidying are combined into single induction cases whose step-substitutions are the union of those combined.

The well-foundedness of the tidied inductions that result is guaranteed, because the induction lemmas applied to prove well-foundedness with original case-analysis can still be applied. The completeness of their case-analyses is guaranteed because the base-cases will (eventually) be constructed to complete the partial case-analysis provided by the step-cases.

An important point to note is that the tidied inductions produced by different tuples of measures and hence different induction lemmas can differ significantly. Thus, since it is not possible to foretell which variants might eventually be required and which not, recursion analysis has to construct as many as it can. It searches for tuples of measures justifying well-foundedness exhaustively, until all distinct tuples have been tried out. For each tuple found the resulting tidied induction and the tuple itself are recorded, indexed under the function they were constructed from. It is this list of measure/induction schema pairs that form what Boyer and Moore term the *induction templates* for a function.

---

[4]This of course assumes the user is not prone to formulating lemmas with redundant pre-conditions!

3.4 Instantiating templates to reflect a Goal

The first proof-time step in recursion analysis is to use the induction templates produced at definition-time to construct the induction schemata of dual to the goal to be proved. For each recursive function instance in the goal the induction templates for that function are looked up and instantiated to match that particular instance of the function. The resulting induction schemata are the true raw material of recursion analysis procedure. Each is dual to one or more terms in the goal, and hence provides a building block for induction schemata that might eliminate those terms. We say that each induction schema *deals with* the terms it is dual to.

The instantiation procedure proper is quite simple: the induction schemata and measures from the induction templates merely have their formal argument variables replaced with the actual arguments in the relevant term instance. For example, if *areverse* is defined as

$$areverse(l, a) \equiv \text{ if } l = nil \text{ then } a \text{ else } areverse(tl(l), cons(hd(l), a)$$

its induction template is

$$\frac{\vdash G}{\neg l = 0, G[tl(l)/l, cons(hd(l), a)/a] \vdash G} \quad \text{with justifying measure } length(l).$$

and would produce the induction schema dual to $areverse(p, b)$ by applying the substitution $[p/l, b/a]$ to the template to give the induction schema

$$\frac{\vdash G}{\neg p = 0, G[tl(p)/p, cons(hd(p), b)/b] \vdash G} \quad \text{with justifying measure } length(p)$$

Instead, the real problem is ensuring that the instantiated induction schemata remain *well-formed* where well-formedness of induction schemata is defined as follows:

Definition 10 Induction Schema Well-formedness

An induction schema is *well-formed* provided that it is well-founded and that its step-substitutions[5] are unambiguous and substitute only for free variables.

■

Boyer and Moore ensure well-formedness in two stages. The first stage is to check whether the induction template will remain well-founded once it has been instantiated. Boyer and Moore call this checking that the template *applies*. If the template cannot be shown to apply its instantiation is discarded. The well-foundedness check Boyer and Moore use is to test whether the instantiation required is such that the original proof of well-foundedness for the template will still apply when it is instantiated. They express this test in terms of two sets of terms extracted from the instantiations for the measured arguments of the induction template. The measured arguments are those formal arguments of the function/induction schema that are referenced in the measure justifying their well-foundedness (and are hence relevant in the template's proof of well-foundedness). Boyer and Moore term the two sets of terms the *changeables* and *unchangeables*.

The changeables are the terms instantiating changing arguments - measured arguments that are changed by the induction's step-substitutions. The unchangeables are the variables embedded in the terms instantiating unchanging arguments - measured arguments that only have empty substitutions in the induction's step-substitutions. Consider, for example, the induction schema dual to the function

$$\begin{aligned} sumto(i,l) \equiv & \text{ if } i \geq l \text{ then } 0 \\ & \text{else } i + sumto(i+1,l) \end{aligned}$$

This schema has form:

$$\frac{\vdash G}{x \leq l, G[i+1/i, l/l] \vdash G}$$ and is justified by the measure $|l - i|$.

Its changing arguments are $\{i\}$ and its unchanging arguments are $\{l\}$. Hence when it is instantiated to suit the goal:

$$\vdash x < y \rightarrow 2 * sumto(x, y + z) = (y + z) * (y + z) + 1) - x * (x + 1)$$

the only changeable is the variable x and the unchangeables are the variables y,z.

A template applies only if the changeables are all distinct variables, and none also appears in the unchangeables. The first part of the test is easily justified: since the induction cannot sensibly substitute for non-variables the changeables *must* all be variables. For example, we cannot instantiate the template for *sumto* in the goal

$$\vdash y < x \wedge x < z \rightarrow sumto((x - y), z) = \ldots$$

because we would end up inducing over a non-variable $(x - y)$. The changeables all need to be distinct - different from each other - because otherwise we might have to substitute in two different ways for the same variable. For example, if *gcd* is defined as

$$\begin{aligned} gcd(x, y) \equiv\ & \text{if } x = y \text{ then } x \\ & \text{else if } x < y \text{ then } gcd(x, y - x) \\ & \text{else } gcd(y, x) \end{aligned}$$

the dual induction is $\dfrac{\vdash G}{\begin{array}{l} x < y, G[x/x, y - x/y] \vdash G \\ x > y, G[y/x, x/y] \vdash G \end{array}}$ justified by the measure (x, y)

and we cannot instantiate the template for *gcd* in the goal $\vdash gcd(x, x) = x$ because x would need to be replaced by both x and $x - x$ in one of the induction hypotheses.

The second part of the test is a little more complicated. The key idea is to observe that, by definition, the unchanging measured arguments are not changed by the un-instantiated schema's step-substitutions. Thus, if the original proof of well-foundedness is to apply to the instantiated schema, the terms instantiating

these arguments must not changed by the step-substitutions of the instantiated induction. Thus, no changeable (a variable that is changed by the step-substitutions) may also appear in the unchangeables. For example, we cannot instantiate the template for *sumto* in the goal

$$\vdash 2 * sumto(a, b + a) = (a + b) * (a + b + 1) - a * (a + 1)$$

because $a + b$ would become $(a + 1) + b$ in the induction hypothesis. In order to show well-foundedness we would have to prove $|(a+1)+b-(a+1)| < |a+b-a|$ (which is false). The original well-foundedness proof which shows $|y - (x+1)| < |y - x|$ would no longer apply.

Boyer and Moore detail the justification of their template applicability test on pp.185-186 of [BM79].

The second step in ensuring the well-formedness of the instantiated template is to ensure the well-formedness of the instantiated substitutions for unmeasured arguments which the applicability test ignores. The approach taken here is rather different since these substitutions only effect the heuristic adequacy of the induction, rather than its soundness. The substitutions are simply tidied until they are well-formed: substitutions for non-variables and unchangeables are discarded along with substitutions clashing with the substitutions already constructed for the unchangeables. Any other clashes between substitutions are resolved by discarding some arbitrarily. In the cases where such clashes occur, and ill-formed substitutions have to be discarded, the resulting induction schema is of course only an approximate dual to the term it was constructed for. For example, if we have the goal

$\vdash palindrome(areverse(p, p))$ where *areverse* is defined

$areverse(l, a) \equiv$ if $l = nil$ then a else $areverse(tl(l), cons(hd(l), a)$

it is impossible to construct an exact dual for $areverse(p, p)$ as this would entail substituting both $tl(p)$ and $cons(hd(p), p)$ for p in the induction hypothesis. The best we can do is the induction schema constructed by the procedure outlined above

$$\frac{\vdash G}{\neg p = nil, G[p/tl(p)] \vdash G}$$

which retains the instantiation for p that allows the induction to be proved well-founded.

3.5 Subsumption and Merging

3.5.1 Subsumption (and Repeated Induction Schemata)

Once the induction schemata dual to recursive function instances in a goal have been collected, recursion analysis' next step is to eliminate those schemata subsumed by others. The aim is to minimise duplicated effort in later stages of recursion analysis when alternative inductions based on the collected duals are evaluated. The notion of subsumption used derives directly from the meta-theoretic foundations underlying the use of dual inductions [6]. An induction schema deals with some terms T_i in a goal when it produces induction hypothesis instances of T_i that match those produced by unfolding T_i in the induction conclusion. An induction schema therefore subsumes another schema if the other only deals with a subset of the terms it deals with.

Given this notion of subsumption, there are in fact two quite distinct ways in which one induction can subsume another. The first possibility is relatively straight forward. An induction schema will subsume another if it is an *extension* of the other. That is, if it has additional step-substitutions and/or it induces over additional variables. The induction hypotheses of an extension, by definition, contain a subset in which the function(s) dual to the un-extended schema are instantiated just as in the un-extended schema's induction hypotheses. Any

[6] It should be emphasised that the analysis of the subsumption check presented here is very much a reconstruction. Boyer and Moore themselves provide only a skeleton justification by way of examples (see pp.190 [BM79]).

terms matched in the un-extended schema must therefore also be matched in the extension. Hence, the extension subsumes the un-extended schema. For example, in the goal

$$\vdash length(l) + length(a) = length(areverse(l, a))$$

the induction schema dual to *areverse*

$$\frac{\vdash G}{\neg l = nil, G[tl(l)/l, cons(hd(l), a)/a] \vdash G}$$

subsumes the induction schema dual to $length(l)$

$$\frac{\vdash G}{\neg l = nil, G[tl(l)/l] \vdash G}$$

because it will produce an induction hypothesis in which $length(l)$ is instantiated exactly as it would be in the induction produced by the induction dual to $length(l)$.

The second possibility is rather more subtle. An induction schema is also subsumed by schemas that are *repeated forms* of itself - induction schemata that capture the effect of applying it repeatedly. For example, an induction schema with one step-case

$$\neg x = 0, \neg p(x) = 0, G[p(p(x))/x] \vdash G$$

is the doubled form of the induction schema with one step-case

$$\neg x = 0, G[p(x)/x] \vdash G$$

The subsumption of an induction schema by repeated forms is shown as follows:

We know that for some terms T_i in the goal to be proved the unrepeated schema produces instances of T_i in the induction hypotheses that match those produced by unfolding T_i in the induction hypothesis. Thus, by definition, the induction hypotheses produced by a repeated form of the schema will contain instances of T_i that will match those produced by repeated unfolding of T_i in the induction conclusion. A repeated form of an induction schema therefore deals

with all terms dealt with by the unrepeated schema, and hence subsumes it. For example, given *even* defined

$even(n) \equiv$ if $n = 0$ then $true$ else if $p(n) = 0$ then $false$ else $even(p(p(n))$

and the goal

$$\vdash even(x) \wedge even(y) \rightarrow even(x + y)$$

then the induction schema dual to $even(x)$

$$\frac{\vdash G}{\neg x = 0, \neg p(x) = 0, G[p(p(x))/x] \vdash G}$$

subsumes the induction schema dual to $x + y$

$$\frac{\vdash G}{\neg x = 0, G[p(x)/x] \vdash G}$$

We can match $x + y$ in the induction conclusion with the instance of $+$ in the induction hypothesis by unfolding it twice:

$$\frac{\frac{\ldots \rightarrow even(p(p(x)) + y) \vdash \ldots \rightarrow even(\underline{x + y})}{\ldots \rightarrow even(p(p(x)) + y) \vdash \ldots \rightarrow even(s(\underline{p(x) + y}))}}{\ldots \rightarrow even(\underline{p(p(x)) + y}) \vdash \ldots \rightarrow even(s(s(\underline{p(p(x)) + y})))}$$

Given that it will detect subsumption of either or both kinds, Boyer and Moore's subsumption test is remarkably simple. Paraphrased, it is defined as follows:

- An induction schema is regarded as subsumed by another if it induces over a subset of the variables induced over by the other, and if each of its cases is subsumed by a case of the other.

- A case is regarded as subsumed by another if each of its step-substitutions is subsumed by one of the other's step-substitutions.

- A step-substitution is subsumed by another if the term it substitutes for any given variable is a subterm of, or identical to, the term substituted for that variable by the other substitution.

A further condition that no unchangeable of the subsumed schema should be a changeable of the subsuming schema (cf. section 3.4) ensures soundness. The test's necessity is easily shown:

We assume some induction schema $S1$ subsumes another $S2$ which deals with terms T_i in the goal to be proved. In each case of $S1$ the induction hypotheses thus, by definition, contains all the instances of the terms T_i produced by (repeated) unfolding of T_i in the induction conclusion. Thus, in each case of $S1$ the step-substitutions must be a superset of (extensions to) the step-substitutions of a case of (a repeated form of) inductions dual to T_i. Hence, in each case of (a repeated form of) $S2$ each step-substitution must substitute terms identical to those substituted by a step-substitution of a case of $S1$. Since the step-substitutions of a repeated form are produced by composing those of an unrepeated schema, $S2$ must substitute subterms of or terms identical to those substituted by $S1$. Hence Boyer and Moore's test will detect $S2$ subsumed by $S1$. Thus, since $S1$ and $S2$ are free Boyer and Moore's subsumption test must necessarily detect schemata that are subsumed by another.

The sufficiency of the test, however, is a rather different story. The price of simplicity in Boyer and Moore's subsumption test is that it is too weak. It does not properly check that the instantiations of the subsuming induction hypotheses are in fact those of an extension of a repeated form of the subsumed induction. Instead, it merely checks that a subsumed induction hypothesis is instantiated subterms of the instantiation of the induction hypothesis subsuming it. This is a reasonable approximation - the induction hypothesis instantiations of a schema are subterms of those of its repeated forms - but not exactly right. There are many undoubtedly non-subsumed inductions schemata, that can be found to be "subsumed" in this way. For example,

$$s \neq nil, hd(s) \neq nil, G[tl(tl(s))/s] \vdash G$$
$$s \neq nil, hd(s) \neq nil, G[hd(tl(s))/s] \vdash G$$

will be regarded as "subsuming"

$$s \neq nil, G[tl(s)/s] \vdash G$$

which is clearly wrong, as a function dual to the latter schema cannot possibly expand to give an instance of itself with an argument matching $hd(tl(s))$.

Similarly, the test does not actually check that the case-structure of the subsuming induction schema really is that of a (possibly repeated) extension of the subsumed induction. It merely checks that each case can be subsumed by at least one of the cases of the subsuming schema. This, for example, means that the induction schema with one step-case

$$\neg x = nil, G[tl(x)/x] \vdash G$$

would be incorrectly recognised as subsumed by the induction schema

$$\frac{\vdash G}{\begin{array}{c}\neg x = nil, even(length(x)), G[tl(x)/x] \vdash G \\ \neg x = nil, \neg even(length(x)), G[2ndhalf(x)/x], G[1sthalf(x)/x] \vdash G\end{array}}$$

whose second step-case is most unlikely to deal with the terms dealt with by the "subsumed" induction.

In practice, however, the subsumption test appears to perform very well. The situations where the flaws outlined above are significant seem very rare. To date, no uncontrived examples where an induction schema is incorrectly subsumed have been found by the author.

3.5.2 Merging

Even with subsumed schemata eliminated, the induction schemata dual to the recursive terms in a goal are still some way away from providing useful inductions for the goal as a whole. The problem is that the schema whilst producing matches for some recursive terms in the goal, will in general produce mis-matches for recursive terms that share variables with the ones it is dual to. For example, consider the goal:

$$\vdash x \leq y \wedge y \leq z \rightarrow x \leq z.$$

If we apply the induction dual to $x \leq y$ terms we produce the induction step-case

$$p(x) \leq p(y) \wedge p(y) \leq z \rightarrow p(x) \leq z \vdash x \leq y \wedge y \leq z \rightarrow x \leq z$$

in which we cannot eliminate $x \leq y$ because the the other recursive terms referencing x and y have been mis-instantiated. The same problem also occurs if any of the other induction's dual to terms in the goal are applied.

The inductions that avoid this problem, and hence allow recursive terms to be eliminated are those that combine the effects of the inductions dual to the recursive terms that share variables. For example, the goal shown above can be dealt with by applying the induction schema

$$\neg x = 0, \neg y = 0, \neg z = 0, G[p(x)/x, p(y)/y, p(z)/z] \vdash G$$

which combines the effects of the inductions dual to all three $\leq$ terms.

The next step in recursion analysis is therefore to combine induction schemata that *overlap* - induce over shared variables. Overlapping schemata cannot however simply be combined by composition - they would interfere with each other. The variables substituted in common would be substituted for twice, while the others would only be substituted for once. The term instances in the induction hypotheses would thus no longer match the results of expanding the terms in the induction conclusion, and the induction would probably fail. The approach is instead to *merge* the inductions by combining cases with step-substitutions that substitute the shared variables identically so as to produce a combined induction schema that combines the effects of both. For example, the induction schemata

$$\frac{\vdash G}{\neg x = nil, \neg y = nil, G[tl(x)/x, tl(y)/y] \vdash G}$$

and

$$\frac{\vdash G}{\begin{array}{c}\neg y = nil, z = 0, G[tl(y)/y, z/z] \vdash G \\ \neg y = nil, \neg z = 0, G[tl(y)/y, p(z)/z] \vdash G\end{array}}$$

merge to give the induction schema

$$\frac{\vdash G}{\begin{array}{c}\neg x = nil, \neg y = nil, z < c, G[tl(x)/x, tl(y)/y, p(z)/z] \vdash G \\ \neg x = nil, \neg y = nil, \neg z < c, G[tl(x)/x, tl(y)/y, z/z] \vdash G\end{array}}$$

which combines the effects of both.

Overlapping induction schemata, whose instantiations of shared variables in their induction hypotheses differ, cannot be merged, as there is no single induction schema that can combine the effects of both. Non-overlapping induction schemata do not need to be merged since they can be safely applied one after the other.

Boyer and Moore's merging algorithm, like the subsumption check, is straightforwardly defined on the structure of induction schemata:

- Two overlapping schemata $S1$ and $S2$ are merged by merging each case of one schema into every case of the other it can be successfully merged with.
- A case $C1$ is merged into a case $C2$ by conjoining $C1$'s governing conditions with those of $C2$ and merging $C1$'s step-substitutions into as many of $C2$'s as is possible. The case merge succeeds provided each of $C1$'s step-substitutions merge into at least one induction hypothesis of $C2$.
- A substitution $S1$ is merged into a substitution $S2$ by taking the union of their individual substitutions for variables. The merge fails if the substitutions for any common variables are not identical.

The merge as a whole succeeds if every case of the first schema could be merged, and no case of the second schema was merged more than once. The details of Boyer and Moore's own presentation of their algorithm can be found on pp.192-193 of [BM79].

The well-foundedness of the merged schema is easy to show:

- The case-analysis of the merged schema is complete, since the (implicit) base case's pre-condition is defined as the negation of the disjunction of the step case pre-conditions.
- Step-substitutions of the merged schema are well-formed by construction. They are the union of well-formed induction schemata that substitute identically on all common variables.

- The merged schema is well-founded since the well-foundedness proof for $S2$ can be applied without change. The pre-conditions governing each case contain those of the corresponding case in $S2$ and the step-substitutions are identical to those of $S2$ except possibly for substitutions for additional variables.

The correctness of the procedure is, however, a rather different matter. We cannot show the merging procedure is correct because we have not properly formalised what it is supposed to achieve. Boyer and Moore are no help because - just as with subsumption checking - they only justify their procedure informally through examples.

What they appear to have missed is that an induction substitutes appropriately for a set of recursive terms if and only if it subsumes the inductions dual to those recursive terms. The merging procedure is supposed to build the simplest *common subsuming schema* for the schemata merged. That is, a single induction schema that subsumes them both. This observation also allows gives proper justification for the exhaustive application of merging. The end result of exhaustive merging is a set of least complex induction schemata subsuming maximal subsets of the induction schemata dual to terms in the goal to be proved. The merged schemata are, in other words, the simplest induction schemata dual to recursive terms in the goal to be proved that avoid inappropriate substitutions in other recursive terms as far as possible.

Once this theoretical framework for merging is in place we can prove the soundness of the merging procedure quite readily. The schema resulting from a merge subsumes both merged schemata because:

- The merged schema is the union of the merged and unmerged cases of $S2$. Hence, each case of the second schema appears in the merged schema, possibly with substitutions for additional variables. The merged schema is thus an extension of $S2$ and hence subsumes it.

- Each case of the $S1$ is merged into at least one case of $S2$. Hence, an extension of each case of $S1$ appears in the merged schema. The first schema is thus also subsumed, but this time *only under Boyer and Moore's weak subsumption test.* This is because, in addition to cases subsuming $S1$ cases, the merged schema may contain additional cases with step-substitutions quite different from any in $S1$.

As we will see in chapter 4, the merging procedure is far from complete. There is at least one important situation where it will fail to find the common subsuming schema of a pair of induction schemata.

3.6 Final Selection

The induction schemata produced by merging provide what is, in effect, the complete set of distinct inductions that could sensibly be applied to the goal to be proved. Each merged schema subsumes the dual induction schemata of one or more recursive terms in the goal, and avoids inappropriate substitutions in other recursive terms if at all possible. No two merged schemata can be replaced by a common subsuming schema, as otherwise the merging procedure would have already done so.

This does not, however, mean that each of the schemata constructed by merging is just as useful as any other. Some inductions will be much more likely to fail, that is to leave unprovable induction subgoals, than others. Similarly, some inductions if they succeed, will make more progress, that is eliminate more recursive term instances, than others. The final step in recursion analysis is therefore a procedure to select the merged induction schema that is likely to make the most progress, with the least risk of failure. In Boyer and Moore's system the selection process falls into two phases: *flaw checking* and *score selection.*

3.6.1 Flaw Checking

The job of the flaw checking procedure is to detect and mark those induction schemata that seem likely to fail. The test employed is based on the observation that the main, readily predictable, causes of induction schema failure are induction's side-effects. An induction schema's side-effects are the substitutions it makes in its induction hypotheses for variables present in recursive terms it does *not* deal with. For example, the side-effect of applying the induction dual to $areverse(l, a)$ to the goal:

$$\vdash areverse(l, a) = append(reverse(l), a)$$

is the substitution of $cons(hd(l), a)$ for a in the second argument position of *append*.

Side-effects cause induction failures when they produce terms in the induction hypothesis that cannot be matched against the induction conclusion. That is, when the side-effect substitution clashes with the step-substitutions of the induction(s) dual to the terms in which the side-effect appears. A good illustrative example (borrowed from [BM79]) is the theorem:

$$append(x, append(y, z)) = append(append(x, y), z)$$

Recursion analysis up to flaw checking gives two induction suggestions, one for induction on x and one on y. The y suggestion is flawed because it will substitute $tl(y)$ for y in the second argument of the fourth "append" - an argument position that will remain unchanged after symbolic evaluation of the induction conclusion. Thus, a match between hypothesis and symbolically evaluated conclusion will be impossible, and the induction is likely to fail.

Boyer and Moore do not, however, laboriously compare each induction schema with all the schemata dual to recursive terms in the goal to be proved to test whether it is flawed. Instead, they applying a much simpler test (see pp.195 [BM79]) which we can paraphrase as:

> An induction schema s_1 is flawed if there exists an induction schema s_2 such that some measured variable of s_1 changed by step-substitutions s_1 is common to s_2.

Boyer and Moore justify this test on the basis that the induction schemata under consideration are the result of merging the schemata dual to the terms in the goal. This means that if a schema s_1 has variables v_i in common with another induction schema s_2 it must substitute inappropriately for v_i in some term s_2 deals with. If it didn't, then s_1 and s_2 would have been combined by the merging procedure. The flawing test ignores shared variables unchanged by an induction's step-substitutions - *unchanging* variables - because only non-identity substitutions can cause side-effects. A recursive term left unchanged by an induction schema always matches between induction hypothesis and conclusion.

The reason shared unmeasured changing variables are ignored is rather more subtle. If we analyse the inductive proofs performed by the Boyer-Moore theorem prover it soon becomes apparent that such variables are almost invariably accumulators in the function definitions to which the induction is dual. That is, they carry some constructed result down through the function's recursion. This means that side-effects produced by such substitutions that occur in argument positions that are destructed can almost always be eliminated. Consider, for example, the goal:

$$evens\_acc(l, n) = plus(n, evens(l))$$

where *evens* and *evens_acc* are defined by:

$$\begin{aligned} evens(l) \equiv\ & \text{if } l = nil \text{ then } 0 \\ & \text{else if } even(hd(l)) \text{ then } s(evens(tl(l)) \\ & \text{else } evens(tl(l)) \end{aligned}$$

$$\begin{aligned} evens\_acc(l, a) \equiv\ & \text{if } l = nil \text{ then } a \\ & \text{else if } even(hd(l)) \text{ then } evens\_acc(tl(l), s(a)) \\ & \text{else } evens\_acc(tl(l), a) \end{aligned}$$

The appropriate induction dual to *even_acc* produces a side-effect for n in the term $plus(n, \ldots)$. This side-effect, however, does not matter because since the

side-effect constructs n, and the recursion of *plus* destructs n, symbolic evaluation will cancel the two out. The proof of the sub-goal in which the side-effect would, superficially, appear to cause the induction to fail has the form:

$$evens\_acc(tl(l), s(n)) = plus(s(n), evens(tl(l))), even(hd(l))$$
$$\vdash evens\_acc(l, n) = plus(n, evens(l))$$

The application of symbolic evaluation however produces the goal:

$$evens\_acc(tl(l), s(n)) = s(plus(n, evens(tl(l)))), even(hd(l))$$
$$\vdash evens\_acc(tl(l), s(n)) = plus(n, s(evens(tl(l))))$$

in which the side-effect is eliminated, so allowing the induction to succeed.

Boyer and Moore ignore side-effects caused by substitutions for unmeasured variables because, almost invariably, they are of this harmless eliminable type.

Nevertheless, as published, Boyer and Moore's flawing test is incorrect. The combination of exceptions for unchanging and unmeasured variables shared conceals a serious bug. We present the details of this bug, and its solution in chapter 4.

3.6.2 Score Selection

Once the flaw checking has eliminated schemata likely to fail due to side-effects, the induction to be applied is selected on the basis of a simple heuristic scoring function. Each induction schema s has associated with it the score:

$$\Sigma_{t \in D(s)} m(t, s)/n(t)$$

where $D(s)$ is the set of recursive terms s deals with, $n(t)$ is the number of arguments possessed by the recursive term t and $m(t, s)$ is the number of arguments to t that the induction schema s substitutes for correctly.

The induction schema applied to the goal to be proved is the one with the highest score. If all the schemata are flawed, the flawing check is ignored and

the flawed induction schema with the highest score chosen. If there is a score-draw, schemata that deal with "awkward" terms with complicated non-structural recursion schemes are preferred.

The scoring procedure is probably best illustrated by example. Consider the goal:

$$\vdash append(a, append(b, c)) = append(append(a, b), c)$$

This produces three dual inductions: one dealing with $append(a, append(b, c))$ one for dealing with $append(a, b)$ and one dealing with $append(b, c)$. The first two merge to give a single induction dealing with both terms. This induction schema is unflawed and has score of 1.5 since it deals correctly with both arguments of $append(a, b)$ and the first argument of $append(a, append(b, c))$. The induction schema dealing with $append(b, c)$ is flawed (see above) and has a score of 1 since it deals correctly with both arguments of $append(b, c)$.

Unfortunately, as with much of the recursion analysis procedure, Boyer and Moore do not really justify their scoring heuristic. We are therefore, once again forced to construct the underlying theory from scratch. The three main considerations that the scoring function appears to reflect are that:

- Heuristically, an induction is more likely to fail if it does not substitute correctly in all argument positions of a term that it deals with. This is because arguments positions that are not substituted appropriately will differ in the induction hypothesis and conclusion. The induction will fail unless there happens to be some rewriting of the induction conclusion and hypothesis that can restore the match. Thus: an induction that enables an exact match between induction hypotheses and conclusion on some recursive term should be preferred to one that enables only a partial match.

- In general, the successful elimination of a recursive term requires rather more than a match on that term between induction hypotheses and conclusion. Usually, matches between many other terms are also required before the literal in which a recursive term appears can be eliminated by hypoth-

esis. An induction that enables a match on many recursive terms should therefore be preferred to one that enables a match on only a few. Thus, since an induction is likely to enable matches on terms that it deals with, an induction that deals with many recursive terms should be preferred to one that deals with only a few.

- Complicated non-structural induction schemata are derived from simple structural induction schemata. This means that complicated induction schemata quite often enable matches on recursive terms with simple structural induction schemata, but not vice-versa. Therefore, since the aim is to choose an induction schema that enables as many matches as possible, complicated induction schemata should be preferred to a simple induction schemata.

Clearly, the Boyer-Moore scoring function is relatively crude. It is questionable, for example, whether the very small penalty applied to inductions that only produce partial matches between induction hypotheses and conclusion is correct. An induction schema that leaves a mismatched argument in every recursive term it deals with can score higher than an induction schema that produces only exact matches. Furthermore, the scoring function only takes into account only a subset of the matches an induction has to achieve to succeed. It largely ignores the the matches an induction may need to achieve through rippling-out or the rewriting of recursive terms that only partly matched with the induction hypotheses.

In chapter 4 we therefore propose a more sophisticated scoring function, that more accurately reflects the ways in which an induction dual to recursive terms in a goal may fail.

3.7 Summary

Boyer and Moore's work on the automatic construction of appropriate induction schemata suffers from a serious flaw: the procedure they propose - recursion analysis - lacks an explanatory theory. In this chapter we have tackled this problem by identifying the theoretical explanation for each step in the Boyer-Moore algorithm.

As the basis for these explanations we identified two key meta-theoretic properties of induction. Namely, that the appropriateness of inductions can be equated with the elimination of recursive terms, and that, heuristically, inductions *dual* to recursive terms in a goal eliminate those recursive terms. The explanation for recursion analysis these basic principles provide us with is as follows:

1. The first phase of recursion analysis - *definition time analysis* - constructs well-founded induction schemata dual to the recursive functions defined in the system.

   For each recursive function definition added to the system:

   The raw induction schema dual to the function is constructed by flattening the conditional structure of the functions' definitions.

   The induction schema found is, if possible, proved well-founded by finding well-founded measures on the variables it induces over that are reduced under its step-substitutions. If no well-foundedness proof can be found the function definition is rejected.

   The well-founded induction schema is tidied to eliminate weak base-cases by removing redundant conditions from the conditions governing its step-cases. The distinct, tidied induction schemata that result are recorded along with their justifying well-founded measures as *induction templates* for the recursive function.

2. The second phase - *template instantiation* - uses functions' induction templates to construct the induction schemata dual to the recursive terms in the goal to be proved. The induction templates for each recursive function appearing in the goal to be proved are instantiated to reflect the instances of the functions present in the goal. These induction schemata provide the raw material for constructing a workable induction dual to recursive terms in the goal, and hence appropriate to the goal.

3. The third phase - *subsumption checking* - eliminates induction schemata that only deal with terms dealt with equally well by others. That is, induction schemata *subsumed* by others are discarded.

4. The fourth phase - *merging* - wherever possible replaces pairs of inductions that overlap (share variables) with a single induction that subsumes both - a *common subsuming schema.*

   The intended effect is the construction of the induction schemata dual to terms in the goal that, as far as possible, avoid substitutions in other terms that could cause them to fail. These schemata represent the set of inductions that have some chance of enabling the elimination of recursive terms from the goal to be proved.

5. The fifth phase - *Final Selection* - selects from the induction schemata produced by merging the induction most appropriate to the goal to be proved, from those produced by merging. It discards any inductions that produce substitutions in terms they do not deal with that are likely to cause the induction to fail. That is, it discards inductions with non-empty side-effects that are unlikely to be eliminable – *flawed* inductions. From those that remain, it then finally chooses the schema that produces the closest match between induction hypotheses and conclusion and/or makes the most progress in the proof.

   The intended result is the selection of the induction schema most likely to produce a match between induction hypotheses and conclusion that will

enable recursive terms to be eliminated. In short, the induction most likely to be appropriate to the goal to be proved.

In the next chapter we use this initial theoretical explanation to find the weaknesses in the Boyer and Moore algorithm, and propose an improved algorithm. On the basis of these results we then present a heuristic theory of appropriate inductions that justifies the recursion analysis approach.

Chapter 4

The Rational Reconstruction

4.1 Introduction

In the previous chapter we have reconstructed a basis for Boyer and Moore's recursion analysis procedure in terms of the meta-theory of induction. In this chapter we show how, by considering the theoretical motivation behind each part of the original design, we can significantly improve it. The end result, we claim, is an algorithm that not only performs better the original, but that is also much more solidly based in theory. Recursion analysis operates in five distinct phases:

1. Definition Time Analysis constructs the well-founded induction schemata dual to the recursive functions defined in the system.

2. Template Instantiation constructs the induction schemata dual to the recursive terms in the goal to be proved.

3. Subsumption Checking eliminates induction schemata that only deal with terms dealt with equally well by others.

4. Merging replaces pairs of inductions that overlap (share variables) with a single induction that subsumes both - a *common subsuming schema.*

5. Final Selection selects from those produced by merging the induction least likely to fail that will make the most progress in the proof.

In this chapter, as in chapter 3, the structure of our presentation deliberately follows that of the procedure we are describing. In each section we deal with just one component of the recursion analysis procedure - in the order in which it is applied. The original implementation of each component is compared with our theoretical framework and the results used to motivate and justify an enhanced realisation of that component of recursions. Flaws inherent in recursion analysis, or issues related to but outside the rational reconstruction are dealt with in chapter 6. The low-level implementational issues that arose when the rational reconstruction was implemented for a specific logic - NuPRL type-theory - are dealt with in chapter 5.

4.2 Constructing Dual Inductions

Recursion analysis begins with the construction of the raw induction schemata dual to the recursive functions defined in the system being used. That is, for each recursive function defined in the system, recursion analysis constructs an (inductive) proof schema whose recursion scheme is identical to that used in the function's definition. The induction schema dual to a recursive term can be defined as follows. For each recursive call in the function's definition, the dual induction has a step-substitution that builds an induction hypothesis instantiated in the same way as the recursive call. For example, if the function f is defined

$$
\begin{aligned}
f(x,y) \equiv\ & \text{if } x < 0 \\
& \text{then } f(-x, y+y) \\
& \text{else if } x > 0 \\
& \text{then } (\text{ if } y < 0 \text{ then } 1 + f(x-2, y+1) \text{ else } f(x-2, y+1)) \\
& \text{else } y
\end{aligned}
$$

then a well-founded induction schema dual to it must have the step-substitutions $[-x/x, y+y/y]$ and $[x-2/x, y+1/y]$. The conditions governing the case(s) in

which the step-substitutions appear are identical to those that govern the corresponding recursive calls in the function's definition. So that the raw induction schema dual to f has the form:

$$\frac{\vdash G}{\begin{array}{c} x < 0, G[-x/x, y + y/y] \vdash G \\ \neg x < 0, x > 0, y < 0, G[x - 2/x, y + 1/y] \vdash G \\ \neg x < 0, x > 0 \neg y < 0, G[x - 2/x, y + 1/y] \vdash G \end{array}}$$

The only complication is that mutually exclusive conditions that govern step-cases with identical step-substitutions can be dropped because they must be redundant in well-founded inductions schemata. Such conditions $C, \neg C$ must be redundant because in each case the necessary condition for well-foundedness W is identical. This means that for the induction to be well-founded:

$$(C \wedge B \rightarrow W) \wedge (\neg C \wedge B' \rightarrow W)$$

must hold for some B B' which by a trivial tautology implies that

$$(B \rightarrow W) \wedge (B' \rightarrow W)$$

Thus, in the example, dual induction schema listed above we drop the governing conditions $y < 0$ and $\neg y < 0$ since they are mutually exclusive and govern the same step-substitution. The induction schema that results is:

$$\frac{\vdash G}{\begin{array}{c} x < 0, G[-x/x, y + y/y] \vdash G \\ \neg x < 0, x > 0, G[x - 2/x, y + 1/y] \vdash G \\ \neg x < 0, x > 0, G[x - 2/x, y + 1/y] \vdash G \end{array}}$$

Boyer and Moore's algorithm for constructing dual inductions is, unfortunately, only a partial implementation of these theoretical principles. The procedure listed in section 3.3 (or outlined on pp.165 [BM79]) suffers from two distinct flaws that can prevent it from constructing an induction dual to a, well-founded, recursive function definition.

One problem is that the algorithm's traversal of a function definition's conditional structure bottoms out whenever it encounters a conditional whose condition contains a recursive call. It completely ignores such recursive conditionals

and any conditionals nested within them when deducing the conditions governing recursive calls in the function definition. It is thus quite unable to construct dual induction schemata for functions whose well-foundedness depends on conditions containing recursive calls. A typical example is the function *minimums* which finds the minimum elements in a list under a partial ordering $<^*$:

$$\begin{aligned} minimums(l, o) \equiv\ & \text{if } l = nil \text{ then } cons(o, nil) \\ & \text{else if } hd(l) <^* o \text{ then } minimums(tl(l), hd(l))) \\ & \text{else if } o <^* hd(l) \text{ then } minimums(tl(l), o) \\ & \text{else if } len(minimums(tl(l), o)) < len(l) \\ & \text{then } minimums(minimums(tl(l), o), hd(l)) \\ & \text{else } cons(o, minimums(tl(l), hd(l))) \end{aligned}$$

The well-foundedness of this recursive definition (and hence that of its dual induction schema) depends on the fact that the recursive call

$$minimums(minimums(tl(l), o), hd(l))$$

is governed by the condition $len(minimums(tl(l), o)) < len(l)$.

The other problem in the Boyer-Moore algorithm lies its criterion for ignoring pairs of negated and unnegated conditions that govern identical recursive calls. The algorithm's traversal of a function definition's conditional structure bottoms out when it reaches conditionals where the recursive calls present in one branch are a subset of those present in the other. This means that all conditions governing recursive calls due to such conditionals and those nested within them are ignored. This is completely incorrect. The governing conditions due to a conditional term can only be ignored as redundant if the recursive calls present in each branch are identical. If one branch of a conditional contains recursive calls not present in the other, it is possible that the well-foundedness of those recursive calls depends on the governing conditions the condition introduces. For example, Boyer and Moore's algorithm will ignore all conditions nested within if $skipable(hd(l)) \ldots$ in the definition

$$
\begin{aligned}
unexpected(l, e) \equiv\ & \text{if } l = nil \text{ then } nil \\
& \text{else if } skipable(hd(l)) \text{ then } unexpected(tl(l), e) \\
& \text{else if } e = nil \text{ then } cons(hd(l), unexpected(tl(l), e) \\
& \text{else if } hd(l) = hd(r) \text{ then } unexpected(l, tl(e)) \\
& \text{else } cons(hd(l), unexpected(tl(l), tl(e))
\end{aligned}
$$

It will, as a result, fail to find an induction schema dual to this function definition, because the well-foundedness of the induction schema depends on the fact that $\neg e = nil$ holds in the last two cases.

Even if the recursive calls in each branch of a conditional are identical, it is still necessary to consider the conditionals nested within it. These may introduce governing conditions crucial to the well-foundedness of the recursive calls nested within them. The fact that Boyer and Moore's algorithm always ignores such conditionals means that it will, for example, fail to construct an induction dual to the function:

$$
\begin{aligned}
lineparse(l, q) \equiv\ & \text{if } q \\
& \text{then if } l = nil \text{ then } cons(spacechar, nil) \\
& \qquad \text{else if } hd(l) = quote \text{ then } lineparse(tl(l), true) \\
& \qquad \text{else } cons(hd(l), lineparse(tl(l), false) \\
& \text{else if } l = nil \text{ then } nil \\
& \text{else if } hd(l) = backslash \text{ then } lineparse(tl(l), true) \\
& \text{else } cons(uppercase(hd(l), lineparse(tl(l), false))
\end{aligned}
$$

The well-foundedness of the induction dual to this function depends entirely on the governing conditions introduced by the conditionals nested inside the if q... conditional.

Fortunately, is not too difficult to implement a sound replacement for Boyer and Moore's algorithm. The only real complication is the need to keep track of recursive calls appearing in the conditions governing other recursive calls. The rationally reconstructed algorithm - a direct realisation of the definition for duals outlined above - is listed in figure 4–1.

Figure 4–1: Algorithm to Construct Induction Dual to Recursive Function

1. *branch* := definition of function
   recrefs := *nil*
   govern := *nil*
   stepcases := *nil*

2. CALL dualcases(*branch*, *recrefs*, *govern*)

3. EXIT Step cases of raw dual induction = *stepcases*

4. PROCEDURE dualcases(branch,recrefs,govern)

   IF *branch* = if *cond* then *truebr* else *falsebr*
   THEN
   newrecrefs := *recrefs*∪ recursive calls in *cond*
   IF recursive calls in *truebr* = recursive calls in *falsebr*
   THEN
   CALL dualcases(*truebr*, *newrecrefs*, *govern*)
   CALL dualcases(*falsbr*, *newrecrefs*, *govern*)
   ELSE
   CALL dualcases(*truebr*, *newrecrefs*, *cons*(*cond*, *govern*)
   CALL dualcases(*falsbr*, *newrecrefs*, *cons*(¬*cond*, *govern*)
   ELSE
   newrecrefs := *recrefs*∪ recursive calls in *branch*
   substs := instantiation of formals in *newrecrefs*
   stepcases := *stepcases* ∪ (*govern*, *substs*)

   RETURN

4.3 Proving Dual Inductions Well-Founded

Once a raw induction schema has been constructed the next step is to prove it well-founded and eliminate any remaining redundant governing conditions. As we have seen in section 3.3.2, Boyer and Moore effectively combine these two tasks into one: that of finding a set of induction lemmas, one per step-substitution, that show the step-substitutions all reduce some tuple of well-founded measures. Such a set of lemmas, as well as allowing the induction schema to be proved well-founded, also permits redundant governing conditions to be eliminated. Each step-case's governing conditions can be replaced by the pre-conditions of the induction lemmas used to show the well-foundedness of its step-substitutions. These are presumed to be free of redundant conditions. For example, the raw induction schema dual to the member function defined in section 3.3 is

$$\frac{\vdash G}{\neg l = nil, hd(l) = x, G[tl(l)/l] \vdash G}$$

We can prove this induction schema well-founded by applying the induction lemma $\neg l = nil \rightarrow length(tl(l)) < length(l)$. This allows us to tidy the induction schema into:

$$\frac{\vdash G}{\neg l = nil, G[tl(l)/l] \vdash G}$$

which omits the redundant governing condition $hd(l) = x$.

For each set of induction lemmas found, the tidied well-founded induction schemata that results is recorded, along with the tuple of measures used, as an *induction template* for the function dual to the raw induction schema.

In terms of efficiency, Boyer and Moore's approach works admirably: the induction lemmas, in effect, index the proofs of well-foundedness for specific measures. Instead of trying all measures, the system can find those likely to be useful through a simple syntactic match. The costly proofs that a measure is reduced under given substitutions need only be performed once - when the

corresponding induction lemmas are introduced. The proof of well-foundedness for an induction schema is reduced to a proof that the conditions governing the induction's step-substitutions imply those of the matching induction lemmas. Such a proof is, in general, considerably simpler than an explicit proof of well-foundedness where the conditions governing the step-substitutions have to be shown to imply the reduction of the relevant well-founded measure. For example, it is much easier to show the induction outlined above well-founded by proving that

$$\neg l = nil, x = hd(l) \vdash \neg l = nil$$

and applying the induction lemma, than it is to prove well-foundedness directly by proving:

$$\neg l = nil, x = hd(l) \vdash length(tl(l)) < length(l)$$

The flaw in Boyer Moore's approach is that it is left to the user to formulate and prove, in advance, all but the most trivial induction lemmas. Only the induction lemmas corresponding to simple structural induction on the recursive data-types known to the system are formulated automatically. The problem is not just that this omission introduces user intervention into an otherwise automatic technique. The nature of the user intervention is such that obscure flaws can, invisibly, be introduced into the system. There is, after all, no guarantee that the user will supply sensible sets of induction lemmas. The lemmas introduced may contain obscure formulations of pre-conditions, resulting in expensive well-foundedness proofs. Worse, the user might even introduce induction lemmas with redundant pre-conditions, resulting in "tidied" induction schemes containing irrelevant case pre-conditions[1].

[1]See section 3.3.3

4.3.1 Eliminating Redundant Preconditions

The identification of preconditions that are redundant in the absolute sense that a theorem can still be proved without them is, of course, intractable. If it were possible to decide when hypotheses were redundant then we would have a decision procedure for theorems of the form:

$$(A \wedge B \rightarrow C) \rightarrow (A \rightarrow C)$$

which would imply a decision procedure for arbitrary theorems B.

This does not mean, however, that *all* redundant hypotheses are equally difficult to recognise. We can identify (and hence eliminate) many redundant pre-conditions in induction lemmas, quite inexpensively. Redundant pre-conditions often appear as *idle* hypotheses in the proof of the induction lemma. That is, they appear as hypotheses that are either unused or only used in redundant parts of the proof. Idle hypotheses can be detected through an analysis of the structure of the proof in which they appear[2].

In the remainder of this section we present a procedure for recognising idle hypotheses in a proof performed in the sequent calculus style described in section 1.4.2. Since, however, the analysis performed by this procedure is by no means transparent, we begin by introducing and explaining the precise notions of redundancy that it makes use of.

Definition 11 P-Required Sets

The *P-required sets* for a sub-goal $H \vdash C$ in a proof P are the minimal $H'(\subseteq H)$ for which $H' \vdash C$ can be proved by a proof constructed by eliminating idle proof steps from P.

■

---

[2]We are, of course, assuming that the theorem prover used records the structure of the proofs it builds.

We define P-required sets and not a P-required set because there may be choices involved in eliminating idle proof-steps. Eliminating one idle proof step may prevent the elimination of another and vice-versa. The minimal H' for which $H' \vdash C$ can be shown by a proof that eliminates idle proof-steps from P may therefore not be unique.

The intuition behind this notion is that for any H', a P-required set for $H \vdash C$, there is a proof of $H \vdash C$ in which the hypotheses $H \setminus H'$ are idle, and hence redundant. For example, in the proof in figure 4–2 we can replace the proof for $H \vdash C$ with the proof for $H' \vdash C$ marked † by discarding the proof-steps marked *. This means that all the hypotheses $(H \setminus H')$ must be redundant.

Definition 12 Idle Sets

The P-idle sets of hypotheses for a sub-goal $H \vdash C$ in a proof P are the sets $(H \setminus r)$ such that r is a P-required set for $H \vdash C$.

■

Definition 13 Throwback and Idle Proof Steps

A *throwback* of a goal $H \vdash C$ in a proof P is a proper descendant of $H \vdash C$ of the form $H' \vdash C$ whose proof P' is such that a P'-required set for $H' \vdash C$ is contained in H.

A proof-step s is *idle* in a proof P if s is contained in a sub-proof of P, P'', whose root $H' \vdash C$ has a throwback whose proof P' does not contain s[3].

■

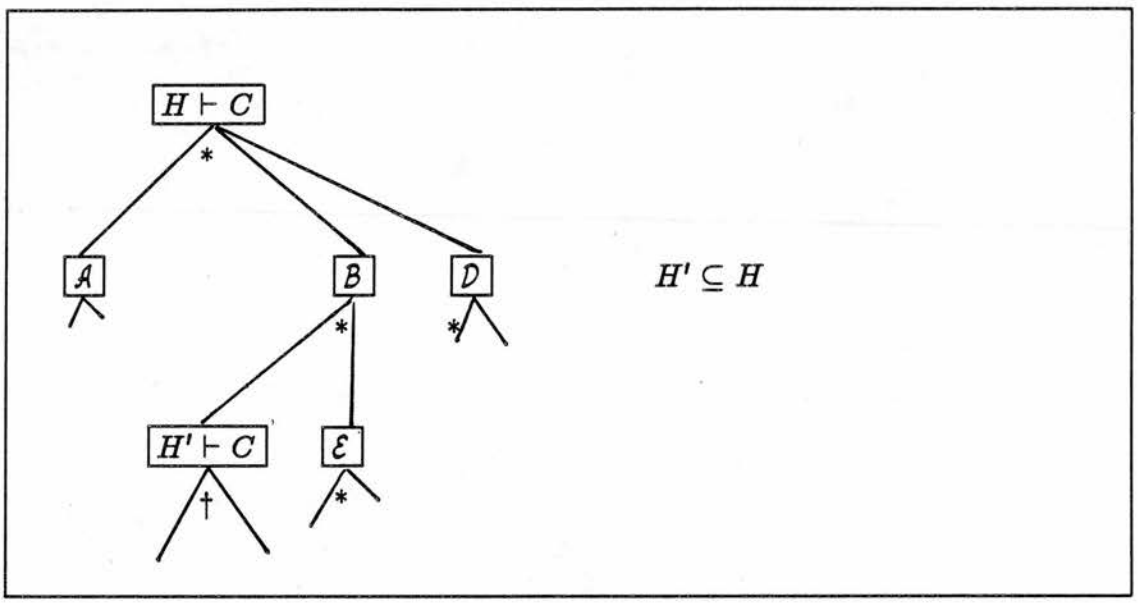
The intuition behind these notions is that the existence of throwbacks indicates redundancy in the structure of a proof. The throwback can be completed

---

[3]Note that the circularity between these definitions and that of required sets is well-founded since P' is a proper sub-proof (sub-tree) of P.

with a proof that requires only a subset of its ancestor's hypotheses. Therefore, by applying this same proof to its ancestor, a complete proof can be obtained that omits all descendants of the ancestor not in the throwback's proof. The example in figure 4–2 illustrates this principle.

Figure 4–2: Throwbacks and Idle Proof Steps



If the hypotheses in H' required in the sub-proof marked † are a subset of H then the proof-steps marked * are all idle because the sub-proof marked † can be used to prove $H \vdash C$.

A more concrete illustration of the intuitions behind these definitions is given by the simple example proof-tree in figure 4–3.

4.3.2 A Procedure to Find Required Hypotheses

The procedure we use to find the P-required sets for P, a proof of the goal $H \vdash C$, is listed in figure 4–5. Informally, the idea behind this procedure is that it recursively traverses a proof, and on the way up from this recursion constructs the P-required sets for the sub-goals of the proof. It does this in four steps:

1. It works out *req'd_here* – the P-required sets for the current goal that correspond to proofs that make use of the existing rule applied to the goal.

Figure 4–3: Example Proof with Redundant Hypothesis and Sub-Proofs

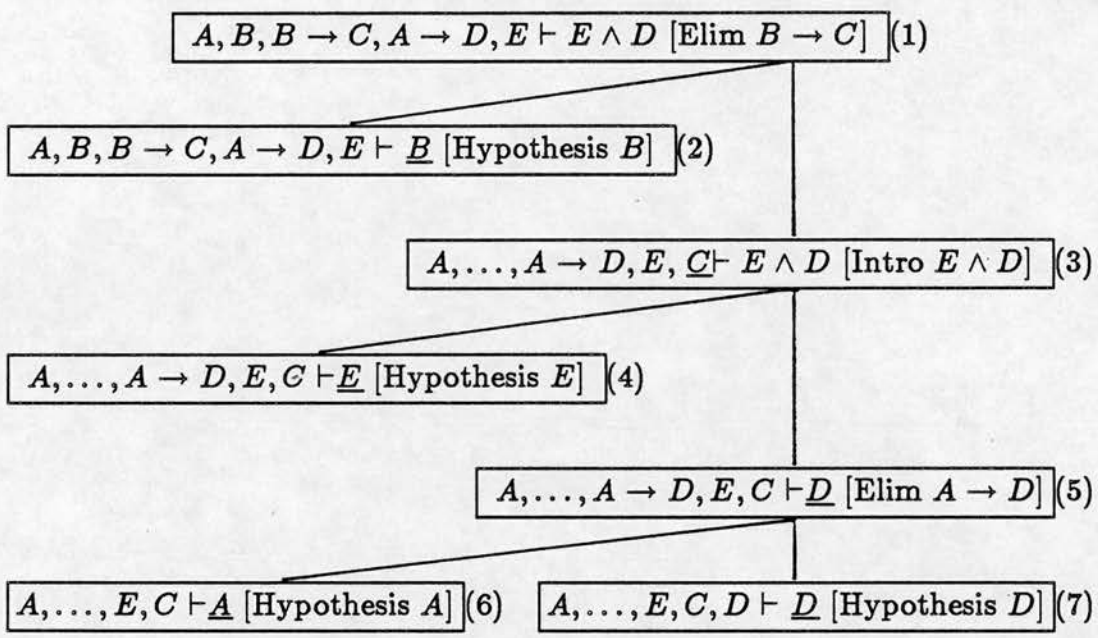
We write each node in the proof-tree [Sub-goal sequent [Rule applied]]

The part of each sequent that differs from its parent is underlined.

"Elim" denotes application of a connective elimination rule.

"Intro" denotes application of a connective introduction rule.

"Hypothesis" denotes the application of a hypothesis.



The hypothesis C that first appears in goal (3) is idle because it is not used in any part of the proof for (3), and hence the proof can be completed without it.

The proof step in the sub-proof whose root is marked (2) is therefore redundant because the rule-application in its parent node (1) only serves to introduce the idle hypothesis C. Goal (3) which differs from its ancestor (1) only by the idle hypothesis C is thus a throwback to (1).

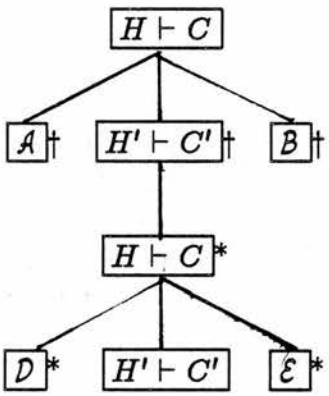
The hypothesis B is not in any required set for the top level goal (and hence is idle) because it is only used in an idle proof-step (2). If we apply the proof of the throwback to the goal (3) the rule application in which B is used is eliminated.

The hypothesis $B \to C$ is idle for the same reason.

These can be worked out from the P-required sets of the goal's children, which are passed up from the recursive calls for these children.

2. It works out *throwb_sets* – the P-required sets for the current goal that correspond to proofs that do not use the existing rule applied to the goal. That is, the proofs based on the proofs for throwbacks to the current goal. These P-required sets can be worked out from the information on throwbacks passed up from the recursive calls to the goal's children.

   NOTE: The P-required sets *req'd_here* are still useful when throwbacks exist because eliminating proof steps with one throwback may prevent the elimination of other proof steps with another throwback. Consider, for example, a proof of the form:



   We can either eliminate the proof steps marked * or those marked † but not both. Therefore, there is no guarantee that the hypotheses required for proofs of $H \vdash C$ based on the throwback $H \vdash C$ are all the P-required sets for $H \vdash C$. The sets of hypotheses required for proofs based on applying the existing rule and then the proof for the throwback *H'vdashC'* could also be minimal.

   We must therefore collect both the required sets for proofs based on throwbacks and those based on applying the existing rule, and combine them.

3. The algorithm combines of *req'd_here* and *throwb_sets* to give *req'd_sets* – the P-required sets for the current goal.

4. It works out if the current goal is a throwback to any of its ancestors. If it is the procedure works out the P-required sets for the current goal whose associated proofs can prove those ancestors, and adds those to the throwback information to be passed up the recursion.

5. It returns the throwback information and the P-required sets for the current goal to its caller.

Figure 4–4 gives a table containing the various sets computed by our algorithm for the nodes of the proof-tree in figure 4–3.

Once the P-required sets for $H \vdash C$ are known, we can readily work out the P-idle hypotheses in $H \vdash C$. Thus, by keeping track of the hypotheses corresponding to particular pre-conditions in the proof of an induction lemma we can identify the distinct alternative sets of redundant pre-conditions.

We prove the correctness of our algorithm for constructing P-required sets as follows:

Theorem

Let P be a sub-proof in some proof whose root node is the goal $H \vdash C$ and whose ancestor goals are A. The procedure call required(P, A) returns a pair $(throwb'\_pairs, req'd\_sets)$ such that:

$req'd\_sets$ are the P-required sets for $H \vdash C$.

$throwb'\_pairs$ is the set of pairs (C', M') such that C' is the conclusion of a throwback in P with P-required sets M' that is a throwback to $H \vdash C$, or a descendant, or a goal A. We term these pairs – $throwb'\_pairs$ – the *throwback pairs* for $H \vdash C$.

Proof

We proceed by induction on the size of P.

Base-case: We assume P has only one node.

Figure 4–4: Trace of Computation of Idle Set for Example Proof

Node	Children	*req'd_here*	*throwb_sets*
		req'd_sets	*throwb_pairs*
(1)	2,3	$\{\{B, E, A \rightarrow D, A\}\}$	$\{\{E, A \rightarrow D, D\}\}$
		$\{\{E, A \rightarrow D, A\}\}$	$\{(B \rightarrow C, \{\{E, A \rightarrow D, D\}\})\}$
(2)		$\{\{B\}\}$	$\emptyset$
	B	$\{\{B\}\}$	$\emptyset$
(3)	4,5	$\{\{E, A \rightarrow D, A\}\}$	$\emptyset$
		$\{\{E, A \rightarrow D, A\}\}$	$\{(B \rightarrow C, \{\{E, A \rightarrow D, A\}\})\}$
(4)		$\{\{E\}\}$	$\emptyset$
		$\{\{E\}\}$	$\emptyset$
(5)	6,7	$\{\{A \rightarrow D, A\}\}$	$\emptyset$
		$\{\{A \rightarrow D, A\}\}$	$\emptyset$
(6)		$\{\{A\}\}$	$\emptyset$
		$\{\{A\}\}$	$\emptyset$
(7)		$\{\{D\}\}$	$\emptyset$
		$\{\{D\}\}$	$\emptyset$

$$\begin{aligned}\text{Idle set} &= \{A, B, B \rightarrow C, A \rightarrow D, E\} \setminus \{E, A \rightarrow D, A\} \\ &= \{B, B \rightarrow C\}\end{aligned}$$

Figure 4–5: Procedure to Find Required Sets of Hypotheses given a Proof

PROCEDURE required(P, A)

$H \vdash C$:= root node of P
req'd_in_rule := Hypotheses required by rule applied to $H \vdash C$
new_hyps := New Hypotheses in root nodes of children of $H \vdash C$
n := Number of children of $H \vdash C$

IF $n = 0$
THEN
    req'd_sets := *req'd_in_rule*
    throwb's := $\emptyset$
ELSE
    FOREACH $i \in \{1, \ldots, n\}$
    DO $throwb's_i, req'd_i$:= required$(child_i, cons(H \vdash C, A))$

    $req'd\_in\_kids := \{nr_1 \cup \ldots \cup nr_n \mid nr_1 \in req'd_1, \ldots, nr_n \in req'd_n\}$
    $req'd\_here := \{(rb \cup req'd\_in\_rule) \setminus new\_hyps \mid rb \in req'd\_in\_kids\}$

    $throwb's := \bigcup_i throwb's_i$
    $req'd\_in\_throwb's := \{r \in R \mid (C, R) \in throwb's\}$

    $req'd\_sets := \{r \in (req'd\_here \cup req'd\_in\_throwb's) \mid$
        $\neg\exists r' \in (req'd\_here \cup req'd\_in\_throwb's).r' \subset r\}$

ENDIF

IF $\exists\, (H' \vdash C) \in A.\exists\, r \in req'd\_sets.r \subseteq H'$
THEN
    $throwb'\_pairs := throwb's \cup \{(C, req'd\_sets)\}$
ELSE
    $throwb'\_pairs := throwb's.$
ENDIF
RETURN $(throwb'\_pairs, req'd\_sets)$

The P-required sets for $H \vdash C$ are therefore the single set of hypotheses required by the inference rule applied $H \vdash C$. Therefore, by its construction, *req'd_sets* is the P-required sets for $H \vdash C$.

Similarly, if $H \vdash C$ is a throwback to a goal in A then the throwback set for $H \vdash C$ is the singleton $\{(C, req'd\_sets)\}$, otherwise the throwback set must be empty. Therefore, by its construction, *throwb'_pairs* is the throwback pairs for $H \vdash C$.

Step-case: We assume P has N nodes (rule applications) and that our theorem holds for all proofs with fewer than N rule applications.

The proofs built by deleting idle proofs steps in P that prove goals $H' \vdash C$ where $H' \subseteq H$ divide into two disjoint classes: those whose root is the same as P's and those whose root is different. We will denote members of the first class P' and members of the second class P''.

Consider the first class - those P' whose root is identical to the root of P.

The proofs of the children of $H' \vdash C$ in any P' proof must, by the definition of P', be built from the proofs for the corresponding children of $H \vdash C$ in the proof P by deleting idle proof steps. Any P'-required set for the ith child of $H' \vdash C$ in a P' proof must thus be a P-required set for the ith child of $H \vdash C$ and vice-versa.

Thus, we can deduce that the hypotheses in the children of $H' \vdash C$ in any P' proof must be a superset of some $\bigcup_i r_i$ where r_i is a P-required set for the ith child of $H \vdash C$. However, by the induction hypothesis, the P-required set of the ith child of $H \vdash C$ is $req'd_i$. Thus, the hypotheses in the children of $H' \vdash C$ in all P' proofs are a superset of some member of *req'd_in_kids*.

Therefore: H' in any $H' \vdash C$ provable by a P' proof must be a superset of some member of *req'd_here*.

Consider the second class - those P'' whose root is not the root of P.

> Trivially, any P'' that proves $H' \vdash C$ where $H' \subseteq H$ must also prove $H \vdash C$. Further, since any P'' is P with idle proof-steps eliminated, its root must be a rule in P applied to a *throwback* to $H \vdash C$ in P.
>
> Hence, by the induction hypothesis, for any $H' \subseteq H$ that permits $H' \vdash C$ to be proved by a P'' there must be some $((H \vdash C), M') \in$ *throwb's* such that H' is a superset of some member of M'.

Therefore: any $H' \subseteq H$ that allows $H \vdash C$ to be proved by a P'' must be a superset of some member of *req'd_in_throwb's*.

We now combine our results for P' and P'' proofs.

> P' and P'' proofs cover all the possible proofs for $H' \vdash C$ where $H' \subseteq H$ built by deleting idle proof-steps from P. Thus, any $H' \subseteq H$ that allows $H' \vdash C$ to be proved by a basis of P must be a superset of some member of *req'd_in_throwb's* $\cup$ *req'd_in_throwb's*. Thus, any P-required set for $H \vdash C$ must be a member of *req'd_here* $\cup$ *req'd_in_throwb's*.

Therefore: *req'd_sets* are the P-required sets for $H \vdash C$.

Finally, by the induction hypothesis, *throwb's* is the union of the throwback pairs for the children of $H \vdash C$. Thus, if $H \vdash C$ is a throwback to a goal in A,*throwb's* $\cup \{(C,$ *req'd_sets*$)\}$ must be the throwback pairs for $H \vdash C$. If $H \vdash C$ is not a throwback to a goal in A then *throwb's* must be the throwback pairs for $H \vdash C$.

Therefore: *throwb'_pairs* is the throwback pairs for $H \vdash C$.

QED.

Throwback and Idle Proof-Steps (revised)

We have in fact simplified the notions of idleness, algorithm, and proof somewhat in the discussion above so as to provide a clear presentation of the underlying

ideas. The actual algorithm used in the rational reconstruction is based on a slightly more effective (but considerably more complicated) extended notion of idle proof-steps. The definition it uses is that:

A proof sub-goal $H \vdash C$ with proof P is a *throwback* if it has an ancestor $H' \vdash C'$, and a descendant (including itself) $H'' \vdash C'$ with a P-required set r such that:

- No proof step between $H \vdash C$ and $H'' \vdash C'$ required to introduce a member of r depends on the goal conclusion.
- $(r \cup r') \subseteq H'$ where r' is the set of hypotheses required in proof steps between $H \vdash C$ and $H'' \vdash C'$ needed introduce the members of r not in H.

We term $H'' \vdash C'$ a *throwback basis*, since it provides the basis for a throwback proper.

A proof step is *idle* if it is a descendant of $H' \vdash C$ that is not in the proof for $H'' \vdash C'$ and not required to introduce members of r not in H.

The intuition behind this extended notion of idleness is that we can prove $H' \vdash C'$ by applying the rules required to introduce members of r not in H followed by the proof of $H'' \vdash C'$ corresponding to the P-required set r.

The extended algorithm is listed in appendix F.

4.3.3 Practical Considerations

Obviously, the details of the discussion so far are only applicable to logics formalised as sequent calculi. The basic principles involved however appear to be quite universal. The only real requirement is that the structure of proofs is open to analysis. For example, in resolution based system the notion of hypothesis redundancy is simply that of not appearing in a clause taking part in a resolution that led to the derivation of the empty clause. Similar criteria for redundancy can be formulated for logics presented in the form of natural deduction proof systems.

It should again be emphasised, however, that our procedure does not detect redundant hypotheses per se, but only hypotheses that can be shown to be redundant in a particular proof. For example, consider the induction lemma:

$$\neg l = nil \wedge member(x, l) \rightarrow length(delete(x, l)) < length(l)$$

The proof of this lemma is based on an induction on l to give the goals:

$$l = nil$$
$$\vdash \neg l = nil \wedge member(x, l) \rightarrow length(delete(x, l)) < length(l)$$

and

$$\neg l = nil,$$
$$(\neg tl(l) = nil \wedge member(x, tl(l)) \rightarrow length(delete(x, tl(l))) < length(tl(l)))$$
$$\vdash \neg l = nil \wedge member(x, l) \rightarrow length(delete(x, l)) < length(l)$$

The most direct proof of the base-case then uses $\neg l = nil$ to derive a contradiction with $l = nil$. Strictly speaking, however, $\neg l = nil$ is redundant since it can be deduced from $member(x, l)$.

It is because of situations like this that it is also worthwhile analysing the proofs of lemma applicability for redundancy. In order to apply an induction lemma $P \rightarrow T' < T$ in a well-foundedness proof we have to prove that $G \vdash P$ where G are the governing conditions of a step-case whose well-foundedness we wish to prove using the induction lemma. If G' - the set of non-redundant governing conditions in the proof of $G \vdash P$ - is a subset of P then the pre-conditions $(P \setminus G')$ must be redundant. For example, if we prove

$$member(x, l) \vdash \neg l = nil \wedge member(x, l)$$

in some well-foundedness proof using the induction lemma above we can deduce that $\neg l = nil$ is redundant.

It is, of course, also possible to apply a theorem prover to the task of checking whether any pre-conditions follow from the remainder. This solution is, however, currently rather impractical because of the many useless attempts that would be made to show non-redundant pre-conditions redundant. Unless very efficient

criteria for giving up on hopeless proofs could be formulated the costs involved would more than likely far outweigh the benefits of eliminating the weak base-case from induction proofs. Furthermore, since the proofs involved are only semi-decidable, it could still not guarantee to find all redundant pre-conditions.

4.3.4 Further Work - Automated Induction Lemma Construction

In addition to tackling the problem of redundant pre-conditions in user-supplied induction lemmas, we can in fact also go some considerable way towards constructing induction lemmas automatically. Recent work by Walther [Wal88] provides an extremely efficient proof procedure for a large subset of well-foundedness proofs based on measures of the structural size of recursive data-objects. The structural size measure for lists is, for example, the *length* function. The structural size measure for trees is the function that counts the number of nodes in a tree.

The heart of Walther's procedure is a syntactic characterisation for defined functions which return values whose structural size is bounded above by the structural size of one or more arguments of the same type. Walther terms such functions N-bounded functions where N is the number of the argument position that bounds the structural size of the function's result. For example, the function *reverse* is 1-bounded because it never returns a list longer than the one it was applied to:

$$\forall l.length(reverse(l)) \leq length(l)$$

This procedure permits efficient proofs of well-foundedness because as well as recognising the arguments a function is bounded by, it also constructs the conditions under which the function's result is strictly smaller than its bounding arguments. That is, as well as finding argument positions i for which a function f can be shown to be i-bounded, Walther's algorithm also constructs a set of conditions C_f^i such that:

$$\forall v_1, \ldots, v_n . C_f^i \rightarrow size(f(v_1, \ldots, v_n)) < size(v_i)$$

where *size* is the structural size measure for objects with the same type as v_i. For example, the function $delete(x, l)$ which deletes all occurrences of x from l is 2-bounded because it never returns a list longer than l. The constructed condition C^2_{delete} under which $length(delete(x, l)) < length(l)$ holds is $member(x, l)$.

Once the C_f^i's are known well-foundedness proofs can be mechanised quite straightforwardly. An induction schema with step-substitutions S that is dual to a function bounded by an argument v is well-founded if:

$\forall \sigma \in S . \exists f_1, \ldots, f_n$ function symbols appearing in $v\sigma$ such that:

$v\sigma$ has the form $f_1(\ldots, f_2(\ldots, f_n(\ldots, v, \ldots), \ldots), \ldots)$

and $\forall i < n . f_{i+1}$ is the outermost function symbol of f_i's p_ith argument.

and f_n's p_nth argument is v

and $\forall i \leq n . f_i$ is p_i-bounded

and $\vdash \Phi \rightarrow \bigvee_{i=1}^{n} C_{f_i}^{p_i}$

where Φ is the condition governing the step-case in which the step substitution σ appears.

The first four conditions ensures that $v\sigma$ it is bounded in size by v. The f_i are constrained so that $size(v) \geq size(f_n(v)) \geq size(f_{n-1}(f_n(v))) \geq \ldots \geq size(v\sigma)$ where *size* is the structural size measure for objects of the same type as v. The last condition then proves that σ reduces v by showing that under σ's governing conditions one of the $\geq$'s in the chain can be replaced with a $>$.

The overall effect is that of a procedure for automatically constructing induction lemmas of the form:

$\Phi \rightarrow size(v\sigma) < size(v)$

Clearly, given the very restricted form of the induction lemmas this procedure can "construct", Walther's work can only partly alleviate the need for user-supplied induction lemmas. Nevertheless, those that it can construct are in practice the vast majority of those required. Generally speaking it is only exceptional recursive function definitions that cannot be proved well-founded by such induction lemmas. For example, out of around 60 recursive functions defined in [BM79] only two ("normalize" pp.355, and "gopher" pp.359) cannot be shown well-founded using a simple measures of structural size or tuples of such measures. This is not to say, however, that Walther's procedure is perfect. There are at least two important situations where it fails to find relatively "obvious" proofs of well-foundedness based on simple structural measures.

The first is when a function increases a structural measure up to a limit rather than decrease it down to a minimum. For example, Walther's procedure cannot prove the well-foundedness of the function:

$factors(i, n) \equiv$
if $i > n$ then nil
else if i divides n then $cons(i, factors(i, n/i))$
else $factors(i + 1, n)$

The second situation is where the argument boundedness of a function is known through a lemma, but cannot be deduced by a syntactic check of its definition. Walther's procedure does not attempt to make any use of appropriate lemmas when finding a sequence of f_i nested in $v\sigma$ that can be shown to be bounded above by v. If it cannot find such a sequence using its syntactic criterion alone it gives up. It is this omission that leads Walther to incorrectly suggest that the function "samefringe" (pp.359 [BM79]) cannot be proved well-founded using a simple structural size measure. The proof of well-foundedness for "samefringe" requires the use of a lemma that shows the function "gopher" is 1-bounded. Gopher fails Walther's syntactic criteria for 1-boundedness.

Unfortunately, although it seems likely that Walther's procedure could be readily extended to deal with these situations, time pressure has not permitted

a proper investigation of the issue. We therefore leave the design and implementation of such an extended procedure as a topic for further work.

The final flaw in Walther's approach is that it provides no help in eliminating redundant pre-conditions. The inductions lemmas it "constructs" simply take the governing conditions of the step-substitution they apply to as their pre-conditions. In order to use Walther's approach in recursion analysis we must apply the procedure outlined in the preceding section, or some similar technique, to identify (and eliminate) redundant pre-conditions. Sadly, as with the extension of Walther's procedure, time-pressure has not allowed us to accumulate experimental evidence of the efficiency – or otherwise – of this procedure. Results from the manual analysis of well-foundedness proofs suggested by Walther's procedure are encouraging, but we are forced to leave more rigorous empirical testing for further work.

4.4 Template Instantiation

The first proof-time step in recursion analysis is the construction of the inductions dual to the recursive function instances in the goal to be proved. For each recursive function instance ("recursive term") the induction templates dual to its function are looked up and instantiated to produce an induction that will deal with that term. This is an induction that will produce instances of that function in the induction hypotheses that will match those produced by symbolic evaluation in the induction conclusion. For example, in the goal:

$$\vdash even(a) \wedge even(b) \rightarrow even(a + b)$$

The induction schema dual to + from +'s induction template

$$\frac{\vdash G}{x \neq 0, G[p(x)/x, y/y] \vdash G}$$ is instantiated to $$\frac{\vdash G}{a \neq 0, G[p(a)/a, b/b] \vdash G}$$

to match the instance of + in the goal.

In its original context, there is relatively little to criticise in Boyer and Moore's approach to instantiating the induction templates. The procedure is

certainly sound: its informal justification in section 3.4 can be readily expanded into a more formal proof (cf. page 188 [BM79]). The only real improvement possible is to slightly refine the treatment of templates with identically instantiated changing variables. Such template instantiations are not, as is assumed in the soundness check, *always* unsound. Soundness is preserved in the special case when the two variables are always substituted identically in induction hypotheses. A few extra induction templates are retained if this exception is taken account of in the soundness check. For example, it would become possible to instantiate the induction template for *merge*:

$$\frac{\vdash G}{\begin{array}{l}\neg l = nil, G[tl(l)/l, r/r] \vdash G \text{ justified by } (length(l), length(r)) \\ \neg r = nil, G[l/l, tl(r)/r] \vdash G\end{array}}$$

to suit the goal

$$\vdash ordered(x) \rightarrow merge(x, x) = double\_each\_element(x)$$

The only other likely refinement would be to check that the template instantiations failing the soundness check really are ill-formed induction schemes. Those that were in fact still well-formed could then be retained. The benefits of such a refinement would however be unlikely to outweigh the cost of the necessary well-formedness proofs. The only class of induction that could be rescued in this way would be those still well-formed despite substituting for unchangeables. Such inductions would however, be relatively unlikely to be of much use. The substituted unchangeables would be guaranteed *not* to match between induction hypotheses and the (symbolically evaluated) induction conclusion.

The real flaw appears when we attempt to apply the technique, outside its original context, to logics other than Boyer and Moore's "Computational Logic". The instantiation procedure is quite unable to deal with quantified or otherwise bound variables. These do not appear in the "Computational Logic" so Boyer and Moore do not suggest how to deal with them.

At first sight the issue looks trivial, bound variables cannot appear outside the scope of the construct that binds them. Therefore:

- If any substitution for a measured variable is instantiated so as to contain bound variables, the induction should be scrapped as it cannot possibly be proved well-founded.

- Any other substitutions that are instantiated to contain bound variables should be deleted as ill-formed.

Unfortunately, to apply this admirably simple rule we need a considerable amount of pre-processing. The problem is that, the rule does not distinguish between irretrievably bound variables and variables that, if necessary, can be made free. For example, consider the goal:

$$\ldots \vdash (\forall x.P(x,y)) \wedge P'(y)$$

where P is a recursive term and x is one of its measured arguments.

The quantification on x can trivially be eliminated by moving it out to cover the whole goal and then applying quantifier elimination. This allows x to be induced over, yet the rule outlined above excludes this possibility. Clearly, when instantiating inductions, we need to distinguish between bound variables and variables that are freeable - variables whose quantifiers we can eliminate without introducing new subgoals. The actual goal rewriting needed to free these variables need, of course, only be performed if any are actually present in the induction schema finally chosen by recursion analysis.

It should be emphasised that a procedure for recognising freeable variables /freeing freeable variables is *not* a goal normalisation procedure. In any given logic there may be combinations of connectives within which quantifiers cannot be eliminated without introducing additional sub-goals. A procedure for finding freeable variables would say nothing at all about the normalisation of goals containing such combinations of connectives. For example, in NuPRL type-theory it is not possible to eliminate the quantification in a goal to the effect:

$$\vdash (\forall x \in A.\, B) \wedge (\forall y \in C.\, D)$$

without splitting it into two separate subgoals by eliminating the $\wedge$ connective.

The details of actual procedures to recognise or free freeable variables of course depend greatly on the logics they are intended to deal with. In a system mechanising a classical logic the we would simply apply the standard algorithm for bringing sentences into prenex normal form. In classical logic variables are freeable if their quantifiers can be transformed into outermost universal quantifiers. Thus, since prenex normal form brings all quantifiers as far out as possible, freeable variables are those universally quantified in the prenex normal form of a goal. For example in the goal:

$$\vdash \forall x.\forall y.\exists z.x < y \rightarrow x + z = y$$

it is easy to show that the only freeable variables are x and y.

In other logics where no normal form exists the construction of a procedure to recognise /free freeable variables would be more difficult. That said, however, it is difficult to conceive of a logic where we cannot derive a freeing procedure in a fairly systematic way from the logic's rules. The aim would be to find all equivalence preserving transformations that eliminated quantifiers or moved them "nearer" to where they could be eliminated. The freeable variables could then be (in-efficiently) found simply by enumerating the distinct goals reachable via the transformations.

The procedure for finding freeable variables in the NuPRL type-theory logic used in the test implementation of the rational reconstruction is presented in appendix E. The Prolog implementation of the rationally reconstructed template instantiation procedure is listed in appendix D.1.

4.5 Merging and Subsumption

Superficially, merging and subsumption checking are very different procedures. The job of subsumption checking is to strip redundant induction schemata from the list of induction schemata an induction appropriate to the goal must subsume. Merging, by contrast, constructs the set of non-subsumed induction schemata dual to terms which are in the goal to be proved which, as far as possible, avoid side-effects in other terms. We were therefore somewhat surprised to discover that the key to improving both phases lay in recognising that both, in fact, implemented the same basic operation.

4.5.1 Flaws in Merging and Subsumption

Merging a pair of overlapping induction schemata replaces them with a single induction schema that deals with all the recursive terms dealt with by either. However, as we noted in section 3.5.2, what this really means is that it constructs a common subsuming schema for the schemata being merged. That is, merging two induction schemata replaces them with a single induction schema that *subsumes* both. For example,

$$\frac{\vdash G}{\neg x = nil, \neg y = nil, G[tl(x)/x, tl(y)/y] \vdash G}$$

and

$$\frac{\vdash G}{\begin{array}{c}\neg y = nil, z = 0, G[tl(y)/y, z/z] \vdash G \\ \neg y = nil, \neg z = 0, G[tl(y)/y, p(z)/z] \vdash G\end{array}}$$

merge to give the induction schema

$$\frac{\vdash G}{\begin{array}{c}\neg x = nil, \neg y = nil, z = 0, G[tl(x)/x, tl(y)/y, z/z] \vdash G \\ \neg x = nil, \neg y = nil, \neg z = 0, G[tl(x)/x, tl(y)/y, p(z)/z] \vdash G\end{array}}$$

which subsumes both.

This, however, is exactly the function performed by subsumption checking for pairs of induction schemata that overlap completely. If one subsumption schema subsumes the other, the other is discarded. Thus, since all induction schemata subsume themselves, the subsumption checking procedure in effect replaces the pair of schemata check with a common subsuming schema. A separate subsumption check is thus quite superfluous, as it is merely a special case of merging. For example, the schema

$$\frac{\vdash G}{\neg x = 0, \neg y = 0, G[p(x)/x, p(y)/y] \vdash G}$$

subsumes the induction schema

$$\frac{\vdash G}{\neg x = 0, G[p(x)/x] \vdash G}$$

and if we apply the merging procedure to these schemata it simply returns the subsuming schema. In effect, the first schema is deleted.

The reason this bug is not glaringly obvious in the original system is that it is masked by a second flaw, this time in the merging procedure. The problem is that the procedure, as presented by Boyer and Moore, is in fact incomplete. It cannot cope with the situation where, although two overlapping induction's cannot be directly merged, their repeated forms (see 3.5.1) can. That is, it cannot construct common subsuming schemata in situations where these need to be extensions of repeated forms of the schemata merged. The subsumption check thus appears useful because it deals with a common situation that the faulty merging procedure does not. This is the situation where the common subsuming schema is (an extension of) a repeated form of one schema, but is identical to the other. A typical example of this situation is the goal:

$$\vdash even(x) \wedge even(y) \rightarrow even(x + y)$$

The common subsuming schema of the induction schemata dual to $x + y$ and $even(x)$ is:

$$\frac{\vdash G}{\neg x = 0, \neg p(x) = 0, G[p(p(x))/x] \vdash G}$$

which is identical to the induction schema dual to $even(x)$.

It is only in the (rarer) situations where the common subsuming schema is an extension of a repeated form of one or both schemata being merged, that the flaw is exposed. The following example illustrates the problem:

$$even(x) \wedge x < y \wedge y < z \wedge \neg even(z) \rightarrow x + 2 < z$$

Boyer and Moore's recursion analysis procedure fails to find a schema appropriate to this goal because the only merges it finds are those between inductions dual to the $<$ and $+$ terms. The induction that results, which replaces x, y, z with $x - 1, y - 1, z - 1$ in the induction hypothesis is fatally flawed because it does not deal with the *even* terms[4].

If, however, repeated form common subsuming schemata are found the situation is transformed. The repeated forms of the $<$ and $+$ inductions merge with the unrepeated schemata dual to the *even* terms giving an induction replacing x, y, z with $x-2, y-2, z-2$. This induction deals with the goal perfectly. Clearly, if we are to deal with problems like this example, we must extend the Merging procedure.

4.5.2 Merging Repeated Forms

A merging algorithm that finds merges between the repeated forms of induction schemata as well as their unrepeated forms must tackle two problems in addition to the actual construction of a merged schema. It must decide which repeated forms (if any) will merge to give the simplest common subsuming schema, and once the repeated forms required (if any) have been decided it must construct them to allow the common subsuming schema to be built.

---

[4]In recent versions of the BMTP the proof, in fact, succeeds in spite of the poor choice of induction. The proof is - eventually - rescued by a powerful arithmetic decision procedure. In similar examples outside arithmetic reasoning, however, the proof still fails completely.

The latter task - the construction of the repeated forms of induction schemata - is relatively straight-forward. The instantiations of recursive term instances produced by repeatedly unfolding a recursive term are the compositions of the instantiations of the recursive term instances produced by a single unfold. For example, if we repeatedly unfold the function $areverse(l, a)$ where *areverse* is defined:

$$areverse(l, a) \equiv \text{ if } l = nil \text{ then } a \text{ else } areverse(tl(l), cons(hd(l), a)$$

we obtain

if $l = nil$ then a
else $areverse(tl(l), cons(hd(l), a))$

then

if $l = nil$ then a
else if $tl(l) = nil$
then $cons(hd(l), a)$
else $areverse(tl(tl(l)), cons(hd(tl(l)), cons(hd(l), a)))$

then

if $l = nil$ then a
else if $tl(l) = nil$
then $cons(hd(l), a)$
else if $tl(tl(l)) = nil$
then $cons(hd(tl(l)), cons(hd(l), a)))$
else if $areverse(tl(tl(tl(l))), cons(hd(tl(tl(l))), cons(hd(tl(l)), cons(hd(l), a))))$

and so on

The step-substitutions of the $n + 1$ repeated form of the induction schema are thus the compositions of those of the n repeated form schema with those of the unrepeated form. The governing conditions of each step-substitution of the $n + 1$ repeated form follow from those governing the step-substitutions it was

built from. They are constructed as the governing conditions of the unrepeated form step-substitution prepended to the conditions produced by applying the unrepeated form step-substitution to the governing conditions of the repeated form step-substitution.

For example, the unrepeated form of the induction dual to *areverse* is:

$$\frac{\vdash G}{\neg l = nil, G[tl(l)/l, cons(hd(l), a)/a] \vdash G}$$

while the doubled form of the induction dual to *areverse* is

$$\frac{\vdash G}{\neg l = nil, \neg tl(l) = l, G[tl(tl(l))/l, cons(hd(tl(l)), cons(hd(l), a)))/a] \vdash G}$$

The algorithm listed below uses these rules to build the $(n+1)$-repeated form of an induction schema given its n-repeated form and its un-repeated form.

1. For each step case u of the unrepeated form:

2. For each step case r of the nth repeated form:

3. Accumulate a new case constructed by applying the substitution of u to r's substitution and governing conditions and adding u's pre-conditions to those of r.

4. The Step cases of the $n+1$th repeated form are the set of distinct cases accumulated.

The other problem an extended merging procedure must solve - finding exactly which repeated forms (if any) are mergeable, and ensuring those found are the simplest possible - is theoretically very much more complicated. This is because it is not at all clear whether the existence of a common repeated form is even decidable.

If we can decide whether a common repeated form of an induction schema exists we can then decide whether there are positive integers m, n such that:

$m \otimes \sigma_1 = n \otimes \sigma_2$

where σ_1, σ_2 are substitutions and $n \otimes \sigma$ denotes the result of composing σ with itself n times.

This substitutions problem, although we have so far been unable to prove its undecidability, is suspiciously similar to several undecidable problems in the theory of formal languages. Certainly, we have found no straight-forward procedure for finding m, n that can be proved to terminate.

Fortunately, there are few, if any, practical consequences of this gap in our theoretical understanding. Non-contrived merges that required the construction of repeated forms higher than the 5th or 6th can to all intents and purposes be ignored. Indeed, the only merges involving repeated forms that seem to arise in practice are merges between the repeated form of one schema and the unrepeated form of another. For example, consider merging the schemata dual to $even(x)$ and $x < y$ as in the example above:

DUAL TO: *even*	DUAL TO: $<$
G	G
(base cases)	(base cases)
$x \neq 0, p(x) \neq 0, G[p(p(x))/x] \vdash G$	$x \neq 0, y \neq 0, G[p(x)/x, p(y)/y] \vdash G$

The simplest schema combining the effects of both is in fact

$$\frac{G}{\text{(base cases)}}$$

$$x \neq 0, p(x) \neq 0, y \neq 0, p(y) \neq 0, G[p(p(x))/x, p(p(y))/y] \vdash G$$

which is in fact the repeated form of the schema dual to $x < y$ merged with the unrepeated form of that dual to $even(x)$.

Thus, as a heuristic, we simply set an arbitrary limit on the repeated forms considered by merging. The algorithm for finding common subsuming schemata that results is listed in figure 4–6. The Prolog implementation used in the rational reconstruction can be found in Appendix D.2. It is worth noting that this enhanced merging algorithm is not in fact significantly more expensive to apply than the original. In the vast majority of cases a merge is either found or rejected in a single iteration of the algorithm. The number of cases where the

algorithm iterates more than once, but eventually rejects a merge is extremely small.

Figure 4–6: Algorithm to Construct Common Subsuming Schemata

1. *oa*,*ob* := overlapping induction schemata

   $a := oa$

   $b := ob$

   $count := 0$

2. $sa :=$ step-substitutions a
   $sb :=$ step-substitutions b
   $count := count + 1$

   IF $\exists\sigma \in sa.\neg\exists\sigma' \in sb.\ \sigma'$ and σ unify
   OR $\exists\sigma \in sb.\neg\exists\sigma' \in sa.\ \sigma'$ and σ unify
   THEN STOP no merge is possible

3. APPLY original merge algorithm to a b
   IF successful
   THEN STOP replace *oa*,*ob* with merged schema

4. IF $\forall\sigma \in sa.\exists\sigma' \in sb.\ \sigma'$ can be instantiated to match σ
   THEN $b \leftarrow$ next repeated form b
   ELSE $a \leftarrow$ next repeated form a

5. IF $count < 7$
   THEN GOTO 2
   ELSE STOP no sensible merge could be found

4.6 Final Selection

The last phase of recursion analysis, final selection, chooses which of the fully merged induction schemata remaining should actually be applied to the goal in hand. There are two basic criteria on which this choice can be based on. These are, in order of precedence: the likelihood an induction will fail (leave uneliminable recursive terms), and the amount of progress it will make if it succeeds. As we have seen in section 3.6, Boyer and Moore spread the implementation of these two criteria through two distinct processing steps. The first, flaw checking, eliminates induction schemata likely to fail. The second, score selection, scores induction schemata according to the number of recursive terms they deal with. The induction schema chosen for application is the highest scoring schema left after flaw checking has been applied. If no schemata survive flaw checking then the highest scoring flawed induction schema is chosen.

4.6.1 Flaws in the Flaw Checking

Boyer and Moore's flaw checking procedure is based on the observation that induction schemata with side-effects - inappropriate substitutions for variables in recursive terms they do not deal with - often fail. It is intended to recognise and discard induction schemata that produce uneliminable side-effects. Unfortunately it does not in fact realise this intent. The test it applies to recognise when an induction schema has side-effects, is incorrect.

Boyer and Moore justify their flaw checking procedure using two facts about the induction schemata produced by merging. Namely that:

> s_1 shares variables v_i with a schema s_2 then some v_i in a recursive term t dealt with by s_2 is inappropriately substituted by s_1; and

> an inappropriate substitution by a schema s for a variable v causes a side-effect if and only if v is a *changing* variable - a variable that is changed by application of s's step-substitutions.

They also implicitly apply the (very effective) heuristic that side-effects produced by an induction's substitutions for unmeasured variables can be eliminated.

On the basis of these observations Boyer and Moore treat as flawed any induction schema s_1 that shares some measured changing variable v with another schema s_2. The flaw in this line of reasoning is that it fails to take into account situations where s_1 also shares some unmeasured or unchanging variable v' with s_2. In such situations it is quite possible that the only inappropriate substitution s_1 makes is that for v'. Yet, if v' is unchanging it cannot produce a side-effect, and if it is unmeasured it is most unlikely to produce one that cannot be eliminated. Thus, in situations where a schema shares changing and unchanging/unmeasured variables with another, Boyer and Moore's flawing test may find the schema as flawed when, in fact, it is not.

It is precisely this bug that causes the problem that Boyer and Moore (incorrectly) attribute to a bug in merging on page 317 of [BM79]. The goal they illustrate the problem with is:

$$x \neq 0 \wedge y \neq 0 \wedge x \neq 1 \wedge \neg prime1(x, y) \wedge y < x$$
$$\rightarrow remainder(x, greatest\_factor(x, y)) = 0$$

The correct induction (that dual to $greatest\_factor(x, y)$ and $prime1(x, y)$) shares both a changing variable (y) and an unchanging variable (x) with the induction dual to $y < x$. This means that although this schema is not flawed (its only inappropriate substitution is for the unchanging x) Boyer and Moore's system discards it as flawed. This allows recursion analysis to choose the wholely inappropriate induction dual to $<$, which is also flawed but scores higher, which eventually causes the proof to fail.

The cure for this bug is trivial once we have correctly identified it. We simply extend the test for flawing to exclude inductions all of whose inappropriate substitutions are in fact identity:

> An induction schema s_1 is flawed if there exists an induction schema s_2 such that some measured changing variable v of s_1 is common to s_2 and s_1's step-substitutions for v differ from those of s_2.

If this corrected flawing test is applied in place of Boyer and Moore's original, recursion analysis will find the correct induction for the goal shown above. The other examples in [BM79] are unaffected – no new failures are introduced. The Prolog implementation of the rationally reconstructed flaw checking procedure is listed in appendix D.3.

4.6.2 Other Causes of Failure

However, even if the flaw check works perfectly, it is still a long way from providing a reliable means of recognising induction schemata that are likely to fail. The problem is that the mechanism for induction failure that it deals with – side-effects – is only one of several. There are at least two other significant causes of induction failure in addition to side-effects:

Failed Match

As we have seen in section 3.4 it is not always possible to build exact dual inductions for the recursive terms in the goal. The "dual" inductions produced for terms with arguments that are not distinct variables are in fact only approximations. Inductions constructed from such duals often produce step-cases in which the recursive terms being dealt with only partially match between the induction hypotheses and the (symbolically evaluated) induction conclusion. These inductions will fail unless the step-cases can be rewritten to complete these partial matches. For example, consider attempting a direct proof of the goal:

$\vdash palin(areverse(x, x))$

where *palin* is a predicate to recognise palindromic lists.

We cannot construct an exact dual for $areverse(x, x)$ because that would imply simultaneously substituting $tl(x)/x$ and $cons(hd(x), x)/x$ in the induction hypothesis. The induction schema produced by recursion analysis gives the induction step case:

$$\neg x = nil, palin(areverse(tl(x), tl(x))) \vdash palin(areverse(x, x))$$

which, after symbolic evaluation of the conclusion, becomes

$$\neg x = nil, palin(areverse(tl(x), tl(x))) \vdash palin(areverse(tl(x), cons(hd(x), x)))$$

The proof fails because there is no possible way we can match

$areverse(tl(x), tl(x))$ with $areverse(tl(x), cons(hd(x), x))$.

Failed ripple out

Even if an induction does produce matching instances of the recursive terms it deals with, it is still not guaranteed to eliminate these recursive terms. As we noted in section 3.2 if the recursive terms are nested inside other terms, then a process of "rippling-out" has to occur. The rewriting of one recursive term to match the induction hypothesis has to enable the rewriting of the function surrounding it until the whole literal matches. Consider, for example, the proof of the goal

$$\vdash remainder(times(plus(a, b), c), c) = 0$$

where *remainder* and *times* are defined

$remainder(x, y) \equiv$ if $\neg x > y$ then x
else $remainder(minus(x, y), y)$

$times(x, y) \equiv$ if $x = 0$ then 0 else $plus(y, times(p(x), y))$

If we apply the induction dual to *plus* and symbolically evaluate *plus* in the induction conclusion we obtain the induction step-case:

$$\neg a = 0, remainder(times(plus(p(a), b), c), c) = 0$$
$$\vdash remainder(times(s(plus(p(a), b)), c), c) = 0$$

In order to match *plus* and the other recursive terms with the induction hypothesis we have to eliminate the mis-matching term - s. We do this by applying the lemma $times(s(x), y)$=$plus(y, times(x, y))$ to give the goal

$$\neg a = 0, remainder(times(plus(p(a), b), c), c) = 0$$
$$\vdash remainder(plus(c, times(plus(p(a), b), c)), c) = 0$$

in which the mis-match is "rippled-out" one level. We eliminate the new mis-matching term *plus* by applying the lemma

$$remainder(plus(y, x), y) = remainder(x, y)$$

to give the goal:

$$\neg a = 0, remainder(times(plus(p(a), b), c), c) = 0$$
$$\vdash remainder(times(plus(p(a), b), c), c) = 0$$

in which the mis-match ripples out completely, so that we can prove the goal by hypothesis.

Thus, since the mis-matches that need to be eliminated depend on the induction schema applied, the success or failure of rippling out depends crucially on the choice of induction schema. An induction schema that produces mis-matches that cannot be rippled out is unlikely to provide a basis for a successful proof.

4.6.3 Further Work - A better Score

We use the analysis of induction failures outlined above to propose a more principled replacement for Boyer and Moore's scoring function. We compute the score of an induction schema s as follows:

Let $T(s)$ be the set of recursive terms in the goal to be proved that s substitutes for or that have terms s substitutes for nested within them. Thus, informally, $T(s)$ the set of terms that s forces us to rewrite to obtain a match between the induction conclusion and the induction hypotheses.

We then construct $T'(s)$ - the subset of terms in $T(s)$ that we can expect to be able to match by rippling-out s. For example, in the rippling-out example above:

$T' = T = \{remainder(\ldots), times(\ldots), plus(\ldots)\}$

Associated with each t in $T'(s)$ is $RO(t)$ – the terms t will appear nested within when the ripple-out passes through t. For example, in the rippling-out example above $RO(times(\ldots)) = \{plus(c, \ldots)\}$ and $RO(plus(ldots)) = \{s(\ldots)\}$.

$T'(s)$ and $RO(t)$ are constructed as follows:

$t \in T'(s)$ if:

either There is no t' such that $t' \in T(s)$ is nested within t **and** t can be rewritten to match the induction hypothesis by symbolic evaluation and/or the application of a lemma.

or The least nested terms in $T(s)$ nested within t, namely $t_1, \ldots, t_n$, are all in $T'(s)$ **and** $\exists ro_1 \in RO(t_1), \ldots, ro_n \in RO(t_n)$ such that $t[ro_1/t_1, \ldots, ro_n/t_n]$ matches the induction hypothesis, or is such that any mis-match can be rippled-out through symbolic evaluation or the application of a lemma.

In both cases $RO(t)$ is constructed from the set of alternative rewritings that match t with the induction hypothesis. That is, $RO(t)$ is the set of terms of the form $f[t']$ where t' is the term corresponding to t in an induction hypothesis, and some $t[ro_1/t_1, \ldots, ro_n/t_n]$ can be rewritten to $f[t']$ through symbolic evaluation or the application of a lemma.

An exception to this rule for computing $T'(s)$ occurs if t is of the form $t_1 = t_2$ and symbolic evaluation or the application of a lemma cannot complete the rippling-out. If, in this situation the terms in $RO(t_1)$ and $RO(t_2)$ are all applications of constructor or destructor functions to the corresponding terms in the induction hypotheses then all subterms of t are eliminated from $T'(s)$. This is because an application of a destructor or constructor function to a term, by definition, denotes a different object from the term itself. The elimination of the

mis-match between t and the induction hypotheses in the situation described would thus, in all probability, be impossible. It would be tantamount to proving non-identical terms identical. The subterms of t would thus almost certainly not match the induction hypotheses, and thus do not belong in $T'(s)$. Consider, for example, the situation where induction and rippling-out leads to the goal:

$$t_1 = t_2 \vdash t_1 = s(t_2)$$

We cannot possibly match the induction hypothesis and conclusion since, by definition, $s(t_2) \neq t_2$ holds.

Once $T'(s)$ has been constructed the score associated with s is computed as $|T'(s)| + |U(s)|$, where $U(s)$ are the number of terms left unchanged by the induction. That is, we score on the basis of the terms an induction can be expected to match, rather than on the number of terms with dual inductions that might match. As with Boyer and Moore's scoring procedure it is intended that the unflawed induction schema with the highest score is chosen for application. In the event of a tie we prefer inductions that deal with the most recursive terms/terms with non-structural inductions.

4.6.4 Initial Results

There are several situations where our scoring procedure demonstrably succeeds where that of Boyer and Moore does not. A typical example is the goal:

$$unique(b) \rightarrow perm(append(a, append(b, c)), append(append(a, reverse(b)), c))$$

where *unique* and *perm* are defined by:

$$\begin{aligned} unique(l) \equiv\ & \text{if } l = nil \text{ then } true \\ & \text{else if } member(hd(l), tl(l)) \text{ then } false \\ & \text{else } unique(tl(l)) \end{aligned}$$

$$\begin{aligned} perm(l, r) \equiv\ & \text{if } l = nil \text{ then } r = nil \\ & \text{else if } member(hd(l), r) \text{ then } perm(tl(l), delete(hd(l), r)) \\ & \text{else } false \end{aligned}$$

If we apply the Boyer-Moore theorem prover to this goal, it scoring scheme prefers the induction on b dual to $append(b, c)$, $unique(b)$ and $reverse(b)$ to that on a dual to $append(a, \ldots)$. The induction on a is, however, much the superior choice. It leads to a proof almost immediately, whereas the induction on b appears to cause the Boyer-Moore theorem prover to run forever.

Under our scoring schema, the induction on a scores 6 even if no appropriate lemmas are known: induction on a is guaranteed to match the two $append(a, \ldots)$ terms along with $unique(b)$ and $reverse(b)$, and will ripple-out through $append(append(\ldots), c)$ by symbolic evaluation. Only the ripple-out through *perm* requires the application of a lemma. The inappropriate induction on b, by contrast scores at most 4: it is guaranteed to match the $append(b, c)$, $reverse(b)$, and $unique(b)$ terms, but will not ripple-out through the other *append*'s, and hence will not ripple-out through *perm*.

An analysis of a sizable sample of proofs produced by the Boyer-Moore theorem prover has produced no examples where our scoring scheme might choose an inappropriate induction. The proofs we have analysed include all those listed by Boyer and Moore in [BM79], plus around a hundred from the standard library of proofs supplied with the Boyer-Moore theorem prover. There are therefore no known situations where our scoring function fails and the Boyer-Moore scheme succeeds.

4.7 Conclusion and Summary

4.7.1 An Explanatory Theory for Recursion Analysis

On the basis of the analysis in this chapter and chapter 3 we are now in a position to formulate an explanatory theory for recursion analysis.

The theoretical roots of the recursion analysis technique lie in the meta-theoretic properties of successful inductive proofs. In a successful proof, induction introduces induction hypotheses that enable literals containing recursive

terms in the induction conclusion to be eliminated by hypothesis. An appropriate induction (an induction likely to lead to a successful proofs) can therefore be characterised as one that produces literals in the induction hypotheses and conclusions that can be rewritten to match each other. An appropriate induction must thus, at the very least, produce reduced instances of recursive terms in the induction hypotheses that match those that can be produced by rewriting recursive terms in the induction conclusion.

In general, however, the only possible rewritings of recursive terms into reduced instances of the same recursive terms are those produced by symbolic evaluation. In special situations lemmas can be applied to appropriately rewrite recursive terms, but only the rewritings produced by symbolic evaluation are always available. Thus, heuristically, an induction must produce induction hypotheses reduced in the same way as the instances of recursive terms produced by symbolic evaluation of the induction conclusion. Hence, heuristically, an induction must subsume the dual inductions of the recursive terms in the goal to be proved if it is to be appropriate for that goal.

This condition although heuristically necessary, is not sufficient to ensure that an induction schema is appropriate. It merely guarantees that the induction schema will enable a match between induction hypothesis and conclusion on the recursive terms that it *deals with.* That is, it will enable a match on the recursive terms whose dual induction schemata it subsumes. If the induction schema is to be appropriate it must also avoid *failure* – it must avoid introducing an uneliminable mis-match on terms it does not deal with. There are three distinct mechanisms by which an induction that subsumes inductions dual to recursive terms in a goal may introduces such mis-matches and fail:

1. The induction may produce uneliminable *side-effects* in recursive terms whose dual inductions it does not subsume.

2. The induction may produce only partial matches between induction hypothesis and conclusion on recursive terms it is only approximately dual to.

3. The induction may fail to enable a *ripple-out* from recursive terms whose duals it subsumes that are nested within other recursive terms.

In all three cases the match between induction hypothesis and conclusion will be imperfect and the elimination of recursive terms by hypothesis unlikely. Thus, heuristically, an induction is appropriate if and only if:

- It subsumes some subset of inductions dual to recursive terms in the goal.
- It is free of uneliminable side-effects in recursive terms whose dual inductions it does not subsume.
- It is likely to enable exact matches between induction hypothesis and conclusion on recursive terms they are only approximately dual to.
- It is likely to enable rippling-out from the recursive terms whose dual induction it subsumes that are nested within other recursive terms.

Recursion analysis successfully produces appropriate inductions because it constructs inductions that precisely meet these criteria. Rationally reconstructed, the algorithm operates in four main steps:

1. It constructs the induction schemata dual to the recursive terms in the goal to be proved.
2. It then merges these induction schemata. The result is the the set of distinct induction schemata dual to terms in the goal that have minimal side-effects.
3. It then, if possible, discards schemata whose non-empty side-effects, are unlikely to be eliminable.
4. It picks from the induction schemata left, the schema whose successful application requires the smallest proportion of steps in which a failed ripple-out or failed match is possible.

4.7.2 Specific Improvements

In addition to this explanatory theory we have also proposed specific improvements to Boyer and Moore's published algorithm for recursion analysis. Each phase of our procedure, we claim, performs better than its equivalent(s) in Boyer and Moore's procedure, and is more solidly based in the theory outlined above. Specifically:

- Definition Time Analysis

  Boyer and Moore's procedure for constructing the induction schema dual to a recursively defined function has major flaws (see section 4.2). The imprecise notion of duality it realises means that it is unable to construct the well-founded induction schemata dual to two significant classes of function. Our replacement procedure derives from a formal definition of duality, and successfully constructs dual inductions in those situations where the Boyer-Moore procedure fails.

  We also saw in section 4.3 that Boyer and Moore's procedure for eliminating redundant pre-conditions from induction schemata is vulnerable to user-errors. The procedure relies on the user formulating induction lemmas without redundant pre-conditions. In section 4.3.1 we presented a method by which a significant proportion of redundant induction lemma pre-conditions can be detected mechanically. In combination with Walther's procedure for automating proofs of well-foundedness [Wal88] this procedure provides the basis for the mechanised construction of a large proportion of induction lemmas.

- Template Instantiation

  Boyer and Moore's template instantiation procedure (see section 4.4) is restricted to quantifier-free logics. Our replacement procedure allows quantification in the goal to be dealt with properly.

- Subsumption and Merging

In section 4.5 we saw that Boyer and Moore's treatment of overlapping induction schemata deviates widely from that required by theory. The requirement is for a procedure to construct a common subsuming schema for pairs of overlapping induction schemata whenever this common subsuming schema exists. Boyer and Moore's system, by contrast, has two procedures for dealing with overlap whose combination fails to find common subsuming schemata in several important situation. The rational reconstruction procedure, however, uses an extension of Boyer and Moore's merging algorithm that[5] satisfies the theoretical requirements. This enhancement allows our rational reconstruction to successfully find inductions for a class of goals where Boyer-Moore recursion analysis demonstrably constructs inappropriate inductions.

- Flaw Checking and Score Selection

  Boyer and Moore's flawing test - intended to detect inductions likely to fail due to side-effects - suffers from two serious deficiencies. Firstly, it fails to detect induction schemata flawed due to side-effects caused by their substitutions for unmeasured variables. Secondly, it can incorrectly identify unflawed inductions as flawed if they share multiple variables with other induction schemata. The rational reconstruction substitutes a sound replacement for this buggy flawing test that eliminates both these deficiencies. This, once again, allows the rational reconstruction to find appropriate inductions in situations where Boyer-Moore recursion analysis fails. We have also proposed, and tested on paper, an improved scheme for scoring induction schemata in the event of choice between several unflawed alternatives.

The overall result is a reconstructed procedure that finds appropriate inductions in situations where the Boyer-Moore original fails, but still succeeds in

---

[5] At least, for all practical purposes

situations where the Boyer-Moore procedure succeeds. In all fairness we must, however, point out that recent *implementations* of the Boyer-Moore theorem prover have modifications that partly solve some of the problems we have tackled:

- The procedure for extracting dual inductions is modified so that it collects all governing conditions (see section 4–1) for the step-substitutions of a function's dual induction. The Boyer-Moore algorithm thus no longer excludes governing conditions that are relevant, but only at the expense of including all irrelevant governing conditions.

  The new procedure is still incapable of automatically extracting dual inductions for function definitions whose well-foundedness depends on conditionals that make recursive calls.

- The problem of schemata being incorrectly rejected as flawed is partly fixed. If all schemata are marked flawed, only those that substitute for other schemata's unchanging variables are rejected. This solves the particular example outlined in 4.6.1, but is not a general solution. Failure can still occur in situations where an unflawed induction schema that shares unchanging variables with others, and some flawed schemata do not substitute for other schemata's unchanging variables.

These partial fixes are, however, neither published nor even documented. Indeed, in the case of the algorithm for extracting dual inductions it is by no means clear precisely what the algorithm is. The relevant procedure is heavily optimised, makes frequent calls to other sub-systems in the theorem-prover, and is written in an uncommented and heavily customised style of LISP. Appendix ?? summarises the results obtained with our algorithm compared with those achieved by the Boyer-Moore theorem prover.

Chapter 5

Implementation in NuPRL

5.1 Introduction

So far, our analysis of recursion analysis has been purely abstract. We have presented a generic design for an enhanced version of recursion analysis suitable for automatic theorem provers based on a wide variety of logics. This is not to say, however, that the implementation of recursion analysis is equally difficult in every logic. Inevitably, some logics are likely to be better and others worse as bases for an automatic induction theorem prover. In this chapter we therefore focus on the problems we encountered in implementing our rational reconstruction on top of the NuPRL type-theory logic[CAB*86].

These problems and the solutions we adopted are of some interest because NuPRL type-theory belongs to a relatively novel class of logics – constructive type-theories – that have been widely proposed as suitable frameworks for formal program development [Mar79]. In attempting to use NuPRL type-theory as a basis for an automatic theorem prover we have, at least in part, put this proposition to the test. As far as we are aware there is no existing literature on the use of these constructive type-theories in automatic theorem provers outside the project of which this thesis is a part.

5.2 NuPRL Type-Theory

NuPRL type-theory is an intuitionistic logic based primarily on the work of Martin-Löf [Mar84] and Curry [CFC58] and de Bruijn [dB70]. It, like the logics on which it is based, is constructed around the idea of representing propositions by the type of witnesses to their truth[1]. Instead of truth-values, NuPRL type-theory deals with judgements - the types of witnesses to propositions. The notion of truth is thus replaced with that of inhabitation - a judgement is true if and only if it is inhabited.

5.2.1 Type-constructors

As might be expected, the construction of composite judgements in type-theory is rather different from that of sentences in a classical logic. Judgements are combined not by logical connectives but by the type-constructors of the kind usually associated with the typed lambda calculus. The core of NuPRL type-theory that realises its "logical" machinery comprises the following seven type constructors:

– Conjunction and Existential Quantification.

A conjunctive proposition $A \wedge B$ is intuitionistically true if and only if there is a witness to the truth of A and a witness to the truth of B. Thus, the type of witnesses to such propositions is the Cartesian product of witnesses to A and witness to B: $A \times B$ is inhabited if and only if A is inhabited and B is inhabited. In NuPRL this type is denoted by the form $A \# B$.

A proposition $\exists\, x \in A.\ B$ intuitionistically true if and only if we can exhibit some object, o, of type A such that there is a witness to the truth of $B[o/x]$.

---

[1]The so called propositions as types approach - see [How80]

The type of witnesses to existentially quantified judgements is thus the Cartesian product of A and B with B dependent on the particular element A. $A \times B$ with B dependent on A is inhabited if and only if A is inhabited by some element o for which $B[o/x]$ is inhabited. In NuPRL this type is denoted by the form $x : A \# B$.

The canonical objects of $\#$ type (pairs) are denoted in NuPRL using the usual form $(Left, Right)$. The operation of decomposing a pair is denoted by the form $spread(P; V_1, V_2.Tm)$ where P is a term that denotes a pair and V_1, V_2 bind its components in Tm. For example, the term

$spread(x; fst, snd.snd)$

denotes the operation of taking the second element of a pair.

Spread obeys the normalisation rule:

$$spread((Left, Right), Var_1, Var_2.Term) \rightarrow Term[Left/Var_1, Right/Var_2]$$

| – Disjunction.

A disjunctive proposition $A \vee B$ is intuitionistically true if and only if there is a witness to the truth of A or there is a witness to the truth of B, and it is decidable which case holds. Thus, the type of witnesses to such disjunctive judgements is the disjoint union of witnesses to A and witnesses to B: the disjoint union $A \cup B$ is inhabited if and only if A is inhabited or B is inhabited. In NuPRL this type is denoted by the form $A \mid B$.

Canonical objects of disjoint union type (disjuncts) are denoted in NuPRL using the forms $inl(Term)$ and $inr(Term)$. The operation of discriminating on a disjunct is denoted by the form $select(Dis; V_1.Tm; V_2.Tm')$ where a left disjunct is bound to V_1 in Tm, and a right disjunct is bound to V_2 in Tm'. *Select* obeys the normalisation rules:

$$select(inl(Term), Var_1.Term_1, Var_2.Term_2) \rightarrow Term_1[Term/Var_1]$$
$$select(inr(Term), Var_1.Term_1, Var_2.Term_2) \rightarrow Term_2[Term/Var_2]$$

-> – Implication and Universal Quantification.

An implication between two propositions $A \supset B$ is intuitionistically true if and only if there is an effective procedure for turning a witness to the truth of A into witness to the truth of B. Thus, the type of witnesses to implications is the type of functions from witnesses to A to witnesses to B: the function type $A \rightarrow B$ is inhabited if and only if there is a function mapping objects of type A into objects of type B. In NuPRL this type is denoted by the form A–> B.

A proposition $\forall x \in A.\ B$ is intuitionistically true if and only if there is an effective procedure that for all objects o in A will construct a witness to $B[o/x]$. The type of witnesses to universally quantified judgements is thus the type of functions from A to B with B dependent on the particular element of A. $A \rightarrow B$ with B dependent of A is inhabited if only if there is a function every element in A, say o, into an element of $B[o/x]$. In NuPRL this type is written $x : A$–> B - *Variable* : *Type*–> *Type* is the NuPRL logic's notation for the dependent function constructor / universal quantification.

Functions are denoted in NuPRL with the usual form $\lambda Var.Tm$ where Var binds the functions argument in Tm. The operation of applying a function to an argument is denoted by the form $Func(Tm)$ where $Func$ denotes a function. A function application obeys the normalisation rule[2]:

$(\lambda Var.Term_1)(Term_2) \rightarrow Term_1[Term_2/Var]$.

U – The type universes.

The first universe of types (the small types) is denoted in NuPRL by the form $U1$. The second universe types - the types formed from types in U1 and U1 itself, is denoted by $U2$, and so on in a cumulative hierarchy.

---

[2]I.e. beta reduction!

***void* – The empty type and negation.**

The false judgement, or the empty type is denoted by the form *void*. Negation in type theory is expressed by the form T–> *void* where T denotes a judgement.

$=\in$ – Equality and Membership

A judgement of equality between two terms within a type is denoted in NuPRL by the form $A = B \in T$ where T denotes a type. This type is inhabited if and only if $A = B$ in the type T, the witness to such equalities is denoted by the form *axiom*. This type serves double duty as a type membership judgement in that the form $A = A \in T$ is inhabited if and only if A inhabits T. The form $A \in T$ is treated as a short-hand for $A = A \in T$.

It is important to note that in NuPRL type well-formedness is defined by type inhabitation. A judgement is well-formed if and only if it can be shown to inhabit one of the universes. Thus, since type-membership is undecidable, a proof in NuPRL type-theory contains numerous well-formedness sub-goals in which terms involved in the proof have to be shown to inhabit the appropriate types. For example, in NuPRL the –> introduction rule actually has the form:

$$\frac{\vdash A \text{-> } B}{A \vdash B \qquad\qquad \vdash A \in Ui}$$

where Ui is a universe that has to be chosen when the rule is applied. The problem of dealing with well-formedness subgoals and the other membership subgoals that frequently arise in NuPRL proofs is a subject we will return to in later sections.

5.2.2 Proof in Type-theory

A theorem (judgement) is proved in type-theory by the construction of a term that inhabits it - a witness term for the judgement. Each rule of inference of the logic is in fact a construction step that uses the witnesses to the rule's antecedents to build a witness to its consequent.

The principles for associating a witness term with a proof are quite straightforward. Introduction rules introduce the form that constructs objects of the type introduced. Elimination rules introduce the form that decomposes such objects. The hypothesis rule introduces the variable that binds the witness corresponding to the hypothesis used. These principles are directly reflected in the conventional notation used to represent proof sub-goals in NuPRL type-theory.

- In the hypothesis list of a NuPRL type-theory sequent each hypothesis is named by the variable that binds its witness in the witness term. A sequent is thus written

  $V_1 : Hyp_1, \ldots, V_n : Hyp_n \vdash Conclusion$

  rather than

  $Hyp_1, \ldots, Hyp_n \vdash Conclusion$

- The structure introduced into the witness term by a rule application is written after the sequent identified by the token **ext**. This convention has its basis in the conventional terminology used in [CAB*86] in which a proof's witness is termed its *extract term.*

Figure 5–1 illustrates these notational conventions and the principles underlying them with the outline of a simple proof in NuPRL type-theory.

Figure 5–1: Example NuPRL Type-theory Proof

$$\frac{\dfrac{\dfrac{\dfrac{\vdash x : A \text{-> } y : (B \# C) \text{-> } A \# B \textbf{ ext } \lambda x.W_1}{x : A, \vdash y : (B \# C) \text{-> } A \# B \textbf{ ext } W_1 = \lambda y.W_3}}{x : A, y : (B \# C) \vdash A \# B \textbf{ ext } W_2 = (W_3, W_4)}}{x : A, y : (B \# C) \vdash A \quad \textbf{ext } W_3 = x} \qquad \frac{x : A, y : (B \# C) \vdash B \quad \textbf{ext } W_4 = spread(y, l, r.W_5)}{x : A, l : B, r : C \vdash B \textbf{ ext } W_5 = l}}$$

Witness term $= \lambda x.\lambda y.(x, spread(y, l, r.l))$

The key advantage of this propositions as types approach over classical logics is its integration of computation (the lambda calculus) and deduction within a single formalism. The combination of lambda calculus, type-system and logic makes it an extremely elegant formalism for expressing programs and proofs about programs. The advantages are perhaps most clearly shown in the way elegant basis type-theory provides for deductive program synthesis.

If a program is specified by an input/output relation $R(input, output)$ then any witness term to the theorem:

$$input : InputType \text{-}\!> (output : OutputType \# R(input, output))$$

must be a function that given an *input* computes an *output* along with a witness that *input* and *output* satisfy the relation R. Thus, any proof of the theorem provides a correct realisation of the specification R. Since type-theory is a fully higher order logic there are no restrictions on the kind of functions that can be synthesised this way. Complex, polymorphic higher-order functionals can be dealt with as cleanly as simple first order functions. This compares very favourably with frameworks for deductive program synthesis based on first order logics [Biu88] [MW80].

An further point worth noting is that the propositions as types approach puts the duality between well-founded induction and recursion treated informally in recursion analysis on a completely formal footing. Consider an arbitrary well-founded induction schema:

$$\frac{\vdash G \textbf{ wit W}}{\ldots, IH : G[f(x)/x] \vdash G}$$

The induction hypothesis produced by such an induction is a judgement identical to the induction conclusion except for the replacement of some terms with smaller terms. The witness to the induction hypothesis IH must thus bind a recursive call to the witness of the conclusion with terms replaced in the same way. Hence, the witness W associated with an induction rule is, by definition, a recursion combinator for the recursion scheme dual to the induction.

Well-founded induction is, furthermore, available within type-theory through ordinary object-level rule of inference. Each inductive type - a type whose canonical members are defined recursively - introduces a decomposition form denoting a recursive decomposition operation for canonical objects of that type. The elimination rules for inductive types are thus rules for structural induction over those types[3]. Well-founded induction schemata can be derived from these structural induction principles using the higher-order facilities of the logic[4]

5.2.3 Induction, Recursion, and Theories

A further important characteristic of the type-theory approach is that it is not possible to make the usual distinction between logic and axiomatic theories expressed in it. Since a witness with the appropriate computational behaviour is required for each theorem, additional types (e.g. integers, groups etc) can only be meaningfully introduced as part of the logic[5]. The axioms of a theory become the introduction and elimination rules for the appropriate type - with the relevant constructor or destructor form as witness. It is because of this constraint that NuPRL type-theory provides the types necessary for the typical program proofs built in. We will focus on just three: *int* the integers, *nat* the natural numbers, and *rec* the recursive data-types.

---

[3]An interesting result in this area due to Backhouse [Bac86] is that the decomposition form and elimination rule can be derived mechanically from a type's introduction rules and constructor forms.

[4]However, not *all* well-founded inductions can be derived - just any that are likely to be useful! All provably well-founded inductions on *nat* can be derived using structural induction and higher-order reasoning, but there is always some ordering that cannot be proved to be a well-ordering using step-wise induction. See, for example, chapter 8 [Sch79]

[5]Or encoded using existing types.

This constraint does not, however, mean that it is impossible to build "theories" in the sense of sets of interdependent lemmas within Type-theory. In the final part of this section we consider the machinery used in NuPRL type-theory to deal with lemmas.

int — The type of integers

The type of integers is denoted in NuPRL by the form *int*. An unusual (and convenient) feature of the NuPRL logic is that the elements of *int* are structureless and the arithmetic operations +,*,/ directly built into the logic. Thus, the canonical elements of *int* are denoted $0, +1, -1, +2, \ldots$ and properties of *int*'s such as the fact that:

$$\frac{\vdash a + b = b + a \in int}{\vdash a \in int \qquad \vdash b \in int}$$

are primitive rules of inference within the NuPRL logic. There is no need to define a + function and prove the desired properties. Inductive definition and proof over the *int*'s is denoted by the form

$$ind(I; I_1', R_1.Neg; Zero; I_2', R_2.Pos)$$

where I denotes an *int*

Neg denotes the computation in the case when $I < 0$. R_1 is bound to a recursive call with $I + 1$ replacing I and I_1' is bound to I.

Zero denotes the computation in the case when $I = 0$

Pos denotes the computation in the case when $I > 0$. R_2 is bound to a recursive call with $I - 1$ replacing I and I_2' is bound to I

For example, the operation of multiplying to *ints* x and y can be written:

$ind(x; n, r.r - y; 0; n, r.r + y)$ using *ind*.

ind normalises as follows:

$$
\begin{array}{ll}
X < 0 \text{ —} & ind(X; I', R.Tm_1; Tm_2; I'', R'.Tm_3) \rightarrow \\
& Tm_1[X/I', ind(X+1; I', R.Tm_1; Tm_2; I'', R'.Tm_3)/R] \\
X = 0 \text{ —} & ind(X; I', R.Tm_1; Tm_2; I'', R'.Tm_3) \rightarrow \\
& Tm_2 \\
X > 0 \text{ —} & ind(X; I', R.Tm_1; Tm_2; I'', R'.Tm_3) \rightarrow \\
& Tm_3[X/I'', ind(X-1; I', R.Tm_1; Tm_2; I'', R'.Tm_3)/R']
\end{array}
$$

The elimination rule for *int* associated with *ind* provides simple induction over the magnitude of an *int*:

$$
\frac{i : int \vdash G \qquad \text{ext } ind(i; i'_1, ih_1.W_1; W_2; i'_2, ih_2.W_3)}{
\begin{array}{lll}
i'_1 : int, ih_1 : G[i'_1/i] & \vdash G[i'_1 + 1/i] & \text{ext } W_1 \\
H, i : int, H' & \vdash G[0/i] & \text{ext } W_2 \\
i'_2 : int, ih_2 : G[i'_2/i] & \vdash G[i'_2 - 1/i] & \text{ext } W_3
\end{array}
}
$$

rec — The Inductive Data Type Constructor

Inductive data-types (loosely, the types of recursive data-structures) are defined in NuPRL by the form $rec(Var, Type)$ which denotes the type with the recursive definition $Var = Type[Var]$[6]. For example, the inductive type of lists of integers that would normally be defined with the equation:

$intlist = NILTYPE \mid (int \# intlist)$ is denoted in NuPRL by the term:
$rec(r, NILTYPE \mid (int \# r)$

The destructor and constructor forms for a *rec* type are built using the constructors and destructors of its component types. For example, the elements of $rec(r, NILTYPE \mid (int \# r)$ include

$$inl(NIL),\ inr(0, inl(NIL)),\ inr(42, inr(42, inl(NIL))), \ldots$$

---

[6]There are some restrictions of the occurrence of Var in $Type$, but these are not significant for the purposes of this thesis

The only form directly associated with *rec* is $rec\_ind(O; Rec, O'.Comp)$ which denotes recursion over the structure of an object, O, of inductive type. In this term *Comp* denotes the recursive computation *Rec* binds the recursive call and *Val* binds *Struct*. For example, the operation of appending two *intlist* objects x and y would be expressed:

$$rec\_ind(x; r, v.decide(v; z.y; ht.spread(ht; hd, tl.inr((hd, r(tl))))))$$

Rec_ind obeys the normalisation rule:

$$rec\_ind(Str; Rec, Val.Tm) \rightarrow Tm[\lambda Z.rec\_ind(Z; Rec, Val.Tm)/Rec, Str/Val]$$

The associated elimination rule for *rec* has the form:

$$\frac{O : rec(V.T) \vdash G \text{ ext } rec\_ind(O; IH, O'.W)}{\begin{array}{l} T' : Ui, TH : (X : T' \text{-> } X \in rec(V.T)), \\ \quad IH : (X : T' \text{-> } G[X/O]), O' : T[T'/V] \quad \vdash G[O'/S] \text{ ext W} \end{array}}$$

The rule is messy because there is no direct way of characterising the recursive components of a canonical object of *rec* type, and building induction hypotheses for each. An awkward type-theoretic trick has to be used instead. The *rec* type is expanded out by replacing the recursive references in it (V) with a new type (T') subsumed by the *rec* eliminated ($X : T' \text{-> } X \in rec(V.T)$). The required induction hypotheses are thus equivalent to (and replaced by) the universal closure of the original goal over the new type ($X : T' \text{-> } G[X/O]$).

unary — The unary type

When defining inductive types using *rec* it is often useful to have a type that has only a single inhabitant to act as the type of empty elements. For example, when defining the type *intlist* above we require a type with just one element $NILTYPE$ for the type of the NIL object. Such types can be defined in the original NuPRL logic, but are rather awkward to use. In the Oyster version of NuPRL type-theory we therefore extend the logic with a type *unary* that has just one canonical inhabitant *unit*.

nat – The natural numbers

A significant disadvantage of the *int* type is that it is rather awkward to use when defining well-founded measures for induction. The presence of negative *int*s introduces the obligation to show an *int* measure is bounded below. In order to avoid this complication the Oyster version of NuPRL type-theory is extended with a type of natural numbers in Peano format *nat*. The canonical elements of *nat* are denoted 0 and $s(X)$ where X denotes an *nat*. The operation of recursively decomposing a canonical member of *nat* is denoted by the form $nat\_ind(Nat; B; V, R.S)$. *nat_ind* obeys the computation rules:

$nat\_ind(0; Base; V, R.Step) \rightarrow Base$
$nat\_ind(s(N); Base; V, R.Step) \rightarrow Step[s(N)/V, nat\_ind(N; Base; V, R.Step)/R]$

term_of — The witness to a lemma

An important practical extension in the NuPRL logic is its support for a library of named theorems. Each theorem proved in the logic introduces a new named lemma and its associated witness term. Witness terms are introduced in the form of the *term_of* term written $term\_of(Thm)$ where *Thm* is the name of an already proved theorem. A *term_of* term inhabits the type expressing the top-level goal of theorem it names and normalises into that theorems witness term. For example, if we prove a theorem named "*a_thm*" whose top-level goal is

$Intro == notation$

$a : int \text{-}\!> b : int \text{-}\!> c : int \# c + a = b \in int$

the logic is automatically extended with the term $term\_of(a\_thm)$ which normalises to the witness of *a_thm*, the lemma rule

$$\frac{\vdash G \text{ ext } (\lambda V.W)(term\_of(a\_thm))}{V : (a : int \text{-}\!> b : int \text{-}\!> c : int \# c + a = b \in int) \vdash G \quad \text{ext } W}$$

and the equality introduction rule

$$\vdash term\_of(a\_thm) \in (a : int \text{-}> b : int \text{-}> c : int \# c + a = b \in int)$$

5.3 Defining Inductive Types

One of the first facilities we need to implement in a logic when dealing with program proofs is a mechanism for the definition of recursive data-types. Unless we can introduce types corresponding to the kinds of objects referenced in programs we cannot expect to be able to prove theorems about programs. As we have seen in section 5.2.3 recursive data-types are defined in NuPRL as inductive types – recursive combinations of the disjoint union and cross-product type constructors. Unfortunately although this approach perfectly adequate from a proof theoretic point of view, it is somewhat flawed from the perspective of human users and/or automatic theorem provers:

- The recursive data-types are anonymous and difficult to distinguish – those with identical structures are indistinguishable.
- Destructors and constructors for recursive types appear as messy anonymous combinations of the constructors or destructors for disjoint union and Cartesian product.
- The elimination rule for recursive data-types is very messy, and can lead to very difficult well-formedness proofs.

In the rational reconstruction we avoid these problems by introducing recursive data-types through a "shell" that hides their implementation inside a series of definitions and lemmas. Recursive type definitions are submitted in the syntax:

$$\begin{aligned} TypeDef &::= DefHead = DefBody \\ DefHead &::= TypeName\{(TVar : Type, TVar : Type \ldots)\} \\ DefBody &::= CoName\{(DeName : CType, DeName : CType \ldots)\} \mid \ldots \end{aligned}$$

where {} denotes optional components

... denotes zero or more repetitions of a component

TypeName is the name of the type to be defined.

Type is an arbitrary NuPRL type.

TVar is a variable binding a type the recursive type is parameterised over.

CoName is the name of a constructor function for recursive type to be defined.

DeName is the name of the destructor function to access the corresponding component of the recursive type to be defined.

CType is the type of the corresponding component of the recursive type - it is either *TypeName* one of the *TVar*'s or a *Type*.

For example, the type of lists would be defined as:

$list(t : U1) = nil \mid cons(hd : t, tl : list)$

which the shell would then realise by adding a definition to the effect that

$list = \lambda t.rec(r.t \# r \mid unary)$ along with appropriate definitions for *cons*, *hd*, *tl* and *nil* in terms of *inl*, *spread* etc. The mechanism by which these definitions are introduced is however very different from that which would be used in a conventional logic. Instead of simply adding a notational definition the terms are introduced as the witness terms of proofs of suitable theorems. The *list* type shown above is, for example, introduced as the witness term of a proof of the theorem:

$list\_thm == t : U1 \text{-> } U1$

The motivation for this unusual approach is best illustrated with an example. Consider the destructor function *hd* defined for the *list* type defined above. If *hd* is introduced in the conventional way by adding the a definition like

$$hd = \lambda arg.select(arg; il.\text{“}anyoldjunk\text{”}; ir.spread(ir; hd, tl.hd))$$

then *hd* is a partial function. It is well-formed only when its argument is a *cons* construction. This would make well- formedness proofs involving *hd* extremely difficult. In order to prove any instance of *hd* well-formed directly the system would need to generalise *hd*'s argument into a variable and perform a case-analysis on its structure: *nil* or *cons*. The *cons* case could then be proved by normalising *hd* out. The *nil* case would require an (expensive) proof.by contradiction with the hypothesis to the effect that *hd*'s argument was a *cons*.

Even applying an appropriate lemma, though cheaper, would still be costly: the appropriate instantiation of the lemma would need to be worked out and introduced for each different application in a well-formedness proof. Furthermore, in more complex proofs it would be very easy to normalise terms like *hd* into forms where the appropriate lemmas could not be applied. Their well-formedness could then only be shown with using the expensive induction procedure. In either case, the extra costs would be amplified by the need to check for the presence of conditionally well-formed terms through-out well-formedness proofs. In large theories with many conditionally well-formed terms this would be singularly expensive. It would also be impossible to deal with *hd* as an object in its own right - only applications of *hd* could appear.

These problems are avoided if *hd* is introduced as a witness term for the theorem

$$t : U1 \text{-> } l : list(t) \text{-> } w : CONSP(l) \text{-> } t$$

where $CONSP(l)$ is the judgement true only if l is a cons-pair object. In this case *hd* is a total function taking three arguments: the type of the components of the list it is applied to, the list itself, and a witness that the list is of *cons* form. Its well-formedness can be proved with nothing more complicated than repeated application of equality introduction rules.

The alternative approach of making destructor functions return exceptions when applied inappropriately – typically used in programming languages (for example ML [HMM86]) – was rejected as too unwieldy. In a programming lan-

guage exception handling is, by default, dealt with by the interpreter. If no exception handling is explicitly programmed, the interpreter applies a default rule for propagating the exception. In NuPRL type-theory this machinery for propagating exceptions would have to be explicitly introduced in the definition of each and every function. This would grossly complicate both function definitions and proofs.

If destructor functions are introduced in the (admittedly less flexible) way outlined above the only major additional complication is the initial derivation of appropriate witness terms, which is in fact quite straight forward. The theorems required are readily derived from the type-definition provided to the shell. The theorem's for non-destructor functions are nothing more than the types of the appropriate functions. For example, the theorem for *list* is $t : U1 \text{-> } U1$

which makes the type for *cons*

$$t : U1 \text{-> } hd : t \text{-> } tl : list(t) \text{-> } list(t)$$

The proofs of these theorems consist of little more than the elimination of -> connectives and the direct introduction of the appropriate type-theory constructors as a witness to the goal produced. These latter can be mechanically derived from the definition of the recursive data-type. For example, the constructors for *list* are specified to be $nil \mid cons(hd, tl)$ giving the proof witnesses $inl(unit)$ for *nil* and $inr((hd, tl))$ for *cons*.

The theorems and proofs needed for destructor functions are only marginally more complicated. The judgement types needed to turn destructors into total functions are trivially derived from the witness introduced in the constructor's proof. For example, $CONSP$ is derived by introducing a boolean function *consp* defined as:

$$select(l; il.FALSE; ir.spread(ir; hd, tl.TRUE))$$

where $FALSE$ and $TRUE$ are defined terms inhabiting a defined type boolean $BOOL$. $BOOL$ is defined as the disjoint union $unary \mid unary$, whose left and right inhabitants are treated as "true" and "false" respectively. $TRUE$ is defined as $inl(unit)$ whilst $FALSE$ is $inr(unit)$. In order to allow boolean functions

to be be used as predicates in type-theory proofs we introduce a function j that returns *void* if its argument is $FALSE$ and *unary* (or any other trivially inhabited type) if its argument is $TRUE$. The type of "hd" thus becomes:

$$t : U1 \text{-}\!> l : list(t) \text{-}\!> w : j(consp(t, l)) \text{-}\!> t$$

Boolean functions and j are used in preference to defining $CONSP$ (etc) directly because it makes proving the decidability of predicates such as $CONSP$ much simpler. We need only show once and for all that

$$x : BOOL \text{-}\!> (j(x) \mid j(x) \text{-}\!> void)$$

rather than have to prove each such predicate decidable individually.

The witnesses introduced in the proofs deriving the destructor functions themselves are constructed in the much same way as the recognisers. For example, the witness term introduced to derive *tl* is:

$$select(l; il.void; ir.spread(ir; hd, tl.tl))$$

The actual proofs for deriving destructor terms differ from those of non-destructors only in the application of a case-analysis on the construction of the destructors argument. In the cases where the destructor is undefined the proof is completed by contradiction with the hypothesis that the argument's is a construction on which the destructor is defined. In the case of *tl* this a contradiction derived from the hypothesis that $consp(l)$.

Figure 5–2 lists the theorems the shell constructs to introduce *intlist* along with the terms introduced as witnesses to prove these theorems. The Prolog proof procedure necessary to implement the shell in the Oyster system [Hor88] is listed in appendix B along with a sample run. In the remainder of this chapter we use the notation $Name(T_1, \ldots, T_n)$ to represent the terms of the form $term\_of(Name)(T_1) \ldots (T_n)$. This is to allow applications of functions introduced as proof witnesses to be conveniently represented.

The introduction of appropriate definitions is, unfortunately, by no means the only task the shell needs to accomplish. Several additional lemmas and

Figure 5–2: Theorems used to Introduce *list* Type

THEOREM	DEFINITION BODY	
list	$t : U1 \text{-> } U1$	$rec(r.t \# r\bar{u}nary)$
cons	$t : U1 \text{-> } a : t \text{->}$	$inl(a, b)$
	$b : list(t) \text{-> } list(t)$	
consp	$t : U1 \text{-> } l : list(t) \text{-> } BOOl$	$select(l; n.void;$
		$c.spread(c; hd, tl.int)$
nil	$t : U1 \text{-> } list(t)$	$inl(unit)$
nilp	$t : U1 \text{-> } l : list(t) \text{-> } BOOL$	$select(l; n.int; c.void)$
hd	$t : U1 \text{-> } l : list(t) \text{->}$	$select(l; n.n; c.spread(c; h, t.h))$
	$w : j(consp(t, l)) \text{-> } t$	
tl	$t : U1 \text{-> } l : list(t) \text{->}$	$select(l; n.n; c.spread(c; h, t.t))$
	$w : j(consp(t, l)) \text{-> } list(t)$	

definitions also required, in order to allow the definitions to be conveniently used.

- An elimination lemma.

  It is frequently necessary in proofs to eliminate inductive types to perform a case-analysis on their structure. In order to hide the messy details of the elimination rule for *rec* types, the shell proves a lemma that expresses the effect of the elimination rule in terms of the type's constructors. In the case of *list* the lemma is

  $$t : U1 \text{-> } l : list(t) \text{-> } g : (list(t) \text{-> } U1) \text{->}$$
  $$\text{-> } ((hd : t \text{-> } tl : list(t) \text{-> } g(cons(t, hd, tl)))$$
  $$\quad \# g(nil(t)))$$
  $$\text{-> } g(l)$$

- A structural size measure.

  In order to apply well-founded inductions over a recursive data-type we require a size measure for the elements of that type. Thus, the type introduction shell also introduces a function counting the number of non-leaf

constructors of an object of the type defined. In the case of *list* this measure function is corresponds to the length of a *list*. It is introduced as the witness term of the theorem:

$len\_thm == t : U1 \text{->} \; l : list(t) \text{->} \; nat$

- Induction Lemmas.

  Recursion analysis requires the known situations in which well-founded measures are reduced to be enumerated as "induction lemmas". The type introduction shell there automatically proves those that follow trivially from the definition of a recursive data-types size measure. In the case of *list* there is just one such lemma:

  $t : U1 \text{->} \; l : list(t) \text{->} \; w : j(consp(t,l)) \text{->} \; len(t, tl(t,l,w)) < len(t,l)$

- Mutual exclusion lemmas.

  A key property of the constructor recognition predicates associated with a defined type is that they are decidable, and mutually exclusive. The type introduction shell therefore automatically introduces and proves the lemmas capturing this fact. In the case of *list* this gives four lemmas:

  $t : U1 \text{->} \; l : list(t) \text{->} \; j(consp(t,l)) \text{->} \; j(nilp(t,l)) \text{->} \; void$

  $t : U1 \text{->} \; l : list(t) \text{->} \; (j(consp(t,l)) \text{->} \; void) j(nilp(t,l))$

  $t : U1 \text{->} \; l : list(t) \text{->} \; j(nilp(t,l)) \text{->} \; j(consp(t,l)) \text{->} \; void$

  $t : U1 \text{->} \; l : list(t) \text{->} \; (j(nilp(t,l)) \text{->} \; void) \text{->} \; j(consp(t,l))$

Clearly, NuPRL's facilities for the use and introduction of new data-types is considerably less than ideal. The use of *rec* to introduce recursive data-types is complicated and messy. A much better solution, given the derivability of elimination and computation rules from a type's formation and introduction rules[Bac86], would be proper mechanism for the introduction of new recursive data-types. This would at least eliminate the complications inherent in realising

new types as combinations of old types. The problems of dealing with exception conditions would, however, remain. Unless type-theory's type-system can be brought in line with the type-systems of functional programming languages it is likely to remain, at best, an awkward environment for reasoning about real (rather than toy) functional programs.

5.4 Defining Functions

NuPRL type-theory's facilities for defining functions, like those for defining types, are theoretically quite powerful but very awkward to use. The system of constructors and destructors for types though Turing-complete, is a long way from providing a convenient notation for user-defined functions. Thus, as with the definition of types, the rational reconstruction defines functions through a "shell" that hides the details of their realisation within the NuPRL logic. The function definition notation used has the syntax:

$$Definition ::= FuncName(Var : Term\{, Var : Term \ldots\}) \equiv DTerm$$

$$\begin{aligned} DTerm ::= \quad & \text{if } DTerm \text{ then } DTerm \text{ else } DTerm \quad | \\ & FuncName(DTerm\{, DTerm \ldots\}) \quad | \\ & ArgVar \quad | \\ & Term \end{aligned}$$

where $FuncName$ is the name of the function to be defined.

Var is a variable of NuPRL type-theory.

$ArgVar$ is a variable appearing in the left hand side of the definition.

and $Term$ is an arbitrary NuPRL type-theory term.

A typical example of such a definition is:

$$\begin{aligned} count(t : U1, l : list(t), p : (t \text{-}\!> U1)) \equiv \quad & \text{if } nilp(t,l) \text{ then } 0 \\ & \text{else if } p(hd(t,l) \text{ then } 1 + count(t, tl(t,l), p) \\ & \text{else } count(t, tl(t,l), p) \end{aligned}$$

As with the functions realising recursive types, user defined functions are introduced as the witness terms of suitable proofs. The proof procedure begins with the deduction of the function's type from the function definition in order to provide the right theorem to prove. The function's domain types are provided by the head of its definition, its range type is deduced by looking up the type of the dominant terms in the bodies of the function's base-cases. The *count* function defined above would, for example, be introduced through a proof of the theorem:

$$\vdash t : U1 \text{-> } l : list(t) \text{-> } int$$

The proof step is to move the function's domain types into the hypothesis list by repeated application of -> introduction. In the case of *count* this gives the sub-goal:

$$t : U1, l : list(t) \vdash int$$

The function's recursion scheme (if any) can then be introduced into the witness by applying its dual induction. The well-founded measure required to justify the induction being found through the procedure outlined in section 3.3.2. In the case of *count* the sub-goal that results is:

$$t : U1, l : list(t), ih : (l' : list(t) \text{-> } len(l') < len(l) \text{-> } int) \vdash int$$

Once the recursion is in place the function's conditional case structure is introduced into the witness term through similarly nested case-analyses on the conditions tested. The proof is then completed by explicit introduction: the body of each conditional case in the function definition is introduced as the witness to the corresponding sub-goal in the proof. The proof of *count*, for example, has the form:

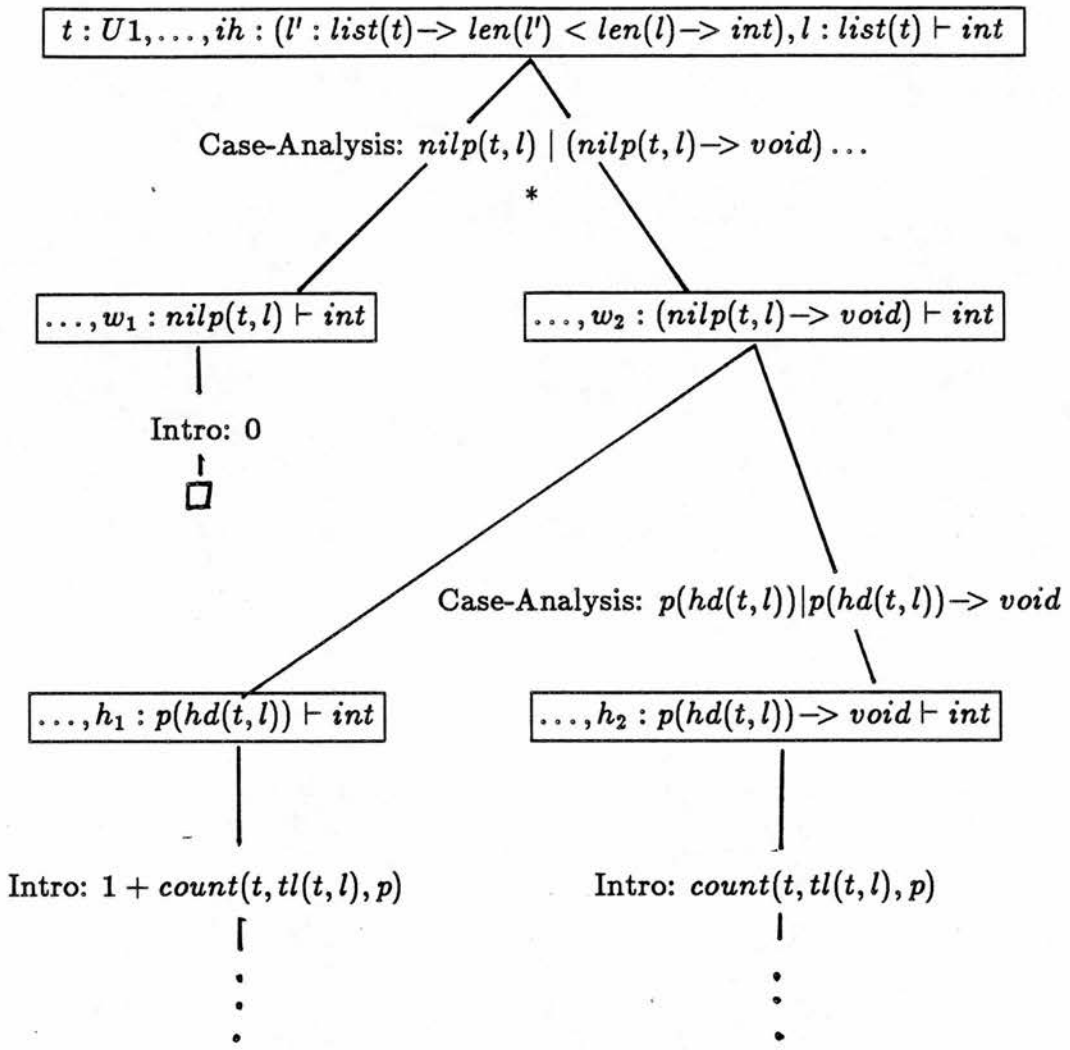


Figure 5–3 outlines the overall proof procedure applied. The simplicity of this procedure and the proof it builds is deceptive. The most difficult part of realising a function definition is in fact embedded within the "Intro" and "Case-Analysis" procedures. The key problem is that the function definition notation omits the well-formedness arguments of conditionally well-formed functions, and permits recursive references to the function being defined. For example, the case-body of the step-case in the definition of *count* contains a reference to *count*, and an instance of *tl* without the required well-formedness argument. The well-formedness argument of *tl* is missing in the $p(hd(t,l))$ conditional. Thus, the main task of Intro and Case-Analysis is not so much the application of the relevant inference rules, but the construction of well-formedness arguments and well-formed recursive references.

Figure 5–3: Procedure to realise function definition

1. Introduce theorem $\vdash$ *TypeOfFunction*.

2. Introduce all –>.

3. Apply dual induction.

4. *term* := body of definition.
   APPLY DeriveFunc(*term*)

5. PROCEDURE DeriveFunc(*term*)

   IF *term* matches if *pred* then *thencase* else *elsecase*
   THEN
   APPLY Case-Analysis(*pred* | *pred*–> *void*)
   IN *pred* SUBGOAL APPLY DeriveFunc(*thencase*)
   IN *pred*–> *void* SUBGOAL APPLY DeriveFunc(*elsecase*)
   RETURN
   ELSE
   APPLY Intro(*term*)
   RETURN

The omitted well-formedness arguments are dealt with first. At each call to Intro or Case-Analysis any absent well-formedness conditions required for the corresponding case-body or condition are cut in. The well-formedness arguments absent in the term are then filled in with the variables binding the witness to the appropriate hypothesis. For example, in the proof steps marked * in the case-analysis for *count* we cut in $consp(t, l)$

$$\frac{w_2 : (nilp(t,l) \text{–>} void) \vdash int}{\vdash consp(t,l) \qquad w_3 : consp(t,l) \vdash int}$$

and in the subsequent steps realise $hd(t, l)$ and $tl(t, l)$ with $hd(t, l, w_3)$ and $tl(t, l, w_3)$.

Ideally, the proof of these cut-in well-formedness conditions would be dealt with by an appropriate automatic theorem prover. Unfortunately, time-pressure

has not permitted this option to be explored in the rational reconstruction. Instead, an interim manual solution is used. The system simply searches for (and applies) lemmas that allow the required conditions to be derived directly from the available hypotheses. If appropriate lemmas cannot be found, the system aborts the definition introduction and indicates to the user the form of those required. It is the user's responsibility to introduce the "condition lemmas" required for the definitions (s)he introduces. The only exceptions are the lemmas capturing the mutual exclusion of different types of constructor. These are automatically made available when a new induction types are introduced. A typical example is the lemma

$$t : U1 \text{-}\!> l : list(t) \text{-}\!> (nilp(t,l) \text{-}\!> void) \text{-}\!> consp(t,l)$$

which is introduced automatically when the *list* type is defined. It is this lemma that we require to deduce *consp* in the proof constructed to derive *count*. We propose to investigate a proper solution using an automatic theorem prover in future work. The work of Plummer [Plu85] and Walther [Wal88] appears to be relevant in this context.

Recursive references are dealt with in a very similar way. The variable binding the witness to the induction hypothesis in the proof binds the recursive reference in the witness term. Recursive calls to the function being defined are realised by applications of this variable to the arguments of the recursive call. The additional argument required which carries the witness to the well-foundedness of the recursive call is supplied by applying the appropriate induction lemma. This lemma having already been found during the search for measures to justify the well-foundedness of the definition described in section 3.3.2. For example, in the proof used to derive *count* the lemma

$$t : U1 \text{-}\!> l : list(t) \text{-}\!> w : conps(t,l) \text{-}\!> len(t, tl(t,l,w)) < len(t,l)$$

is applied after the proof steps marked * to give

$$\dfrac{\dfrac{w_3 : consp(t,l) \vdash int}{\ldots}}{\dfrac{\text{APPLY:} \ldots \text{->} \; len(t, tl(t,l,w)) < len(t,l)}{\ldots}}$$

$$w_4 : len(t, tl(t,l,w_3)) < len(t,l) \vdash int$$

which allows $count(t, tl(t,l,w_3), p)$ to be realised with the term $ih(tl(t,l,w_3)), w_4)$ in the subsequent applications of Intro.

It is worth noting at this point that the introduction of definitions as proof witnesses is no more costly than the direct introduction of definitions in more conventional systems. The only non-trivial proof steps required are those that simply re-appear in separate proofs of well-foundedness and well-formedness when the function is introduced directly. The Prolog implementation of the function definition procedure used in the rational reconstruction is listed in appendix C, along with some sample runs.

5.5 Induction

In the discussion so far we have assumed that we can straightforwardly apply well-founded induction within the NuPRL logic. Unfortunately, this is not the case - the only induction schemes directly available in NuPRL are simple structural inductions. Thus, the final component required to allow recursion analysis to be implemented in NuPRL type-theory is a proof procedure to realise well-founded induction in terms of structural induction.

We illustrate the procedure used with the well-founded induction needed for the derivation of the *merge* function.

$$\begin{aligned} merge(l,r) \equiv\ & \text{if } l = nil \text{ then } r \\ & \text{else if } r = nil \text{ then } l \\ & \text{else if } hd(l) < hd(r) \text{ then } cons(hd(l), merge(tl(l), r)) \\ & \text{else } cons(hd(r), merge(l, tl(r))) \end{aligned}$$

This induction scheme is justified by the lexicographic ordering on the pair of measures $len(l), len(r)$ and has the form

$$\frac{l : list(t), r : list(t) \vdash G[l, r]}{l' : list(t) \text{-->} r' : list(t) \text{-->} len(l'), len(r') \ll len(l), len(r) \text{-->} G[l', r'] \vdash G[l, r]}$$

where G is some arbitrary judgement on l and r, and $\ll$ is the lexicographic ordering on pairs based on $<$.

The procedure for implementing a well-founded induction begins by generalising the conclusion of the goal into its universal closure over the variables appearing in the induction's justifying measure. The quantification is constructed so that variables are bound in the order of their left-most instances in the justifying measure. In our example induction this gives the goal:

$$\ldots \vdash l : list(t) \text{-->} r : list(t) \text{-->} G[l, r]$$

The expansion procedure then recurses left to right over the measures in the justifying tuple. In each recursion step, any quantifiers binding variables appearing in the current left-most measure are eliminated, and the well-founded induction based on the left-most measure applied. This latter step being accomplished through the application of lemmas of the form

$$j : (nat \text{-->} U1) \text{-->} (m : nat \text{-->} (n : nat \text{-->} n < m \text{-->} j(n)) \text{-->} j(m)) \text{-->} l : nat \text{-->} j(l)$$

which has a straightforward proof requiring only structural induction[7].

The end result, after some tidying up, is a sub-goal equivalent to that produced by well-founded induction. The only difference is the (trivial) replacement of the single induction hypothesis in terms of $\ll$ with an equivalent list of induction hypotheses in terms of $<$. In the case of the *merge* induction we obtain the goal:

---

[7]The "trick" is to prove the theorem indirectly by proving the stronger theorem $\ldots \text{-->} l : nat \text{-->} l \leq m \text{-->} j(l)$

$$
\begin{array}{l}
\quad l : list(t), r : list(t), \\
\quad ih_1 : (l' : list(t) \text{-> } len(l') < len(l) \text{-> } r' : list(t) \text{-> } G[l', r']), \\
\quad ih_2 : (r' : list(t) \text{-> } len(r') < len(r) \text{-> } G[l, r']) \\
\vdash G[l, r]
\end{array}
$$

5.6 Well-formedness Proofs

The most obvious difference between proofs in NuPRL type-theory and proofs in classical logics is the presence of membership sub-goals in which terms have to be shown to inhabit some appropriate type. For example, when we apply the rule for -> introduction:

$$\frac{\vdash x : A \text{-> } B \text{ ext } \lambda x.W}{x : A \vdash B \text{ ext } W \qquad\qquad \vdash A \in Ui}$$

we obtain the well-formedness sub-goal $A \in Ui$ in addition to the goal $x : A \vdash B$ which we might expect.

Fortunately, despite the undecidability of type membership, the well-formedness sub-goals that arise in all but the most exotic proofs can be proved quite trivially. If conditionally well-formed terms do not appear in a proof, well-formedness sub-goals can almost always be proved by exhaustive application of the term normalisation, equality introduction, and hypothesis rules. The replacement of conditionally well-formed terms with well-formed equivalents is, as we have seen in section 5.3, relatively straight-forward.

The trivial nature of almost all well-formedness proofs does not, however, imply that they inexpensive to perform. The real problem is simply the sheer size of the well-formedness proofs. If we view the term whose type-membership is being proved as a tree, each application of equality introduction eliminates just a one node - the root. The remainder re-appear in the sub-goals introduced by the rule application. The time-complexity of well-formedness proof for a term is thus at least linear in the size of a term. In practice, it usually increases rather more This is because NuPRL applies inference rules in reverse so that a

good number of the equality introduction rules requires "guessed" types to be supplied. For example, the equality introduction rule for function applications:

$$\frac{\vdash F(A) \in FT}{\vdash F \in AT \text{-}> FT \qquad \vdash F \in AT}$$

requires us to supply a guess as to the type of the argument object in the function application. This guess can only be supplied through an expensive look-ahead procedure - trying out the proof with an arbitrary type to see which type is required, and then rebuilding the proof once the right type is found.

Clearly, this need for large numbers of trivial but lengthy well-formedness sub-proofs is a serious drawback of the NuPRL type-theory logic. A partial solution would be to extend the logic with an efficient partial decision procedure for "trivial" well-formedness sub-goals. User-written proof-procedures for well-foundedness proofs – as are currently used – require the use of considerable computational resources in the construction of explicit proof trees for well-formedness sub-goals. This effort, however, is entirely redundant since the extract term for well-formedness sub-goals is always *axiom*, and hence requires no reference to the proof-tree. A partial decision procedure for trivial well-formedness integrated into the logic would be considerably more efficient since it could use whatever data-structures were most efficient, rather than an explicit proof-tree. For example, the need for look-ahead to deduce guessed types might be eliminated through judicious use of meta-variables and unification.

Exactly this approach is already used in NuPRL to minimise the cost of arithmetic reasoning. Instead of many low-level rules about the arithmetic operations the logic implements an arithmetic decision procedure that captures their overall effect. Only "non-trivial" steps are left for the user/automatic theorem prover.

5.7 Computation

The final major difference between proofs in NuPRL type-theory and in classical logics is the distinction the NuPRL logic makes between term normalisation (computation) rules and rules of inference. In a classical logic all axioms whether "computational" or not are used in exactly the same way. They are applied as lemmas within a proof. In contrast the term normalisation rules of NuPRL behave very differently from the rules of inference. The application of a normalisation rule simply changes the form used to denote a particular object in a proof. The object - and hence the witness term - are unaffected. No well-formedness sub-goals are produced since if the object denoted is unchanged so is its type-inhabitation. In contrast, the application of a rule of inference introduces some structure into witness term that constructs or decomposes an object. In many cases well-formedness sub-goals appear because of the need to show that the objects referenced by an application of the rule are, in fact, of the appropriate types. For example, when we apply –> introduction

$$\frac{\vdash A \text{–>} B \text{ ext } \lambda V.W}{V : A \vdash B \text{ ext } W \qquad\qquad \vdash A \in Ui}$$

we are constructing a witness term $\lambda V.W$ from W a witness to the first sub-goal, and have to show that A denotes a type in the second sub-goal. If, however, we apply the computation rule for λ

$$\frac{\vdash (\lambda X.F)(A) \text{ ext W}}{\vdash F[A/X] \text{ ext W}}$$

we leave the witness term unaffected and have no well-formedness sub-goal to show.

The crucial consequence of this distinction between computation and inference rules is that it makes it impossible to extend computation with lemmas. Consider, for example, the goal

$$\vdash spread((2,1); h, t.h)) = 1 + 1 \in int$$

If we apply the normalisation rules directly to this goal we obtain $\vdash 2 = 1 + 1 \in$ *int* and can immediately prove the goal. However, if we apply the lemma

$$\vdash x : int \text{-}> y : int \text{-}> spread((x, y); h, t.h)) = x \in int$$

then we obtain the goal

$$h : (x : int \text{-}> y : int \text{-}> spread((x, y); h, t.h)) = x \in int)$$
$$\vdash spread((2, 1); h, t.h)) = 1 + 1 \in int$$

which we can prove only after we have: instantiated h with 2 and 1; proved 2 and 1 inhabit *int*; substituted for *spread* in the goal; and finally proved the goal is well-formed with an integer in the position of *spread*.

This is clearly a significant flaw in the NuPRL logic given that the application of computation often plays a central role in induction proofs. If computation cannot be extended with lemmas every computation sub-proof always has to be constructed from scratch even if it is very long, and has already been performed many times before.

The solution would be to modify NuPRL type-theory to allow derived rules of normalisation – *computation lemmas* – to be introduced. The basic scheme is uncomplicated. If a term t_1 can be rewritten to a term t_2 through a series of normalisation rule applications we introduce $t_1 \rightarrow t_2$ as a derived normalisation rule. We treat any free variables in t_1 as schema variables for the purpose of applying the lemma to particular terms.

The only real complication is in ensuring computation lemmas do not permit violations of NuPRL type-theory's restrictions that on the application of normalisation rules in proofs.

If T is a goal or hypothesis and $t(\preceq T)$ is a term properly occurring in T or identical then we may only apply a normalisation rule to t if $\exists u, v.t \preceq u \preceq v \preceq T$ such that:

1. No free variables of u are bound in v

2. $\forall r.v \prec r \preceq T \Rightarrow r$ is a canonical term

3. $\forall r.t \prec r \preceq u \Rightarrow r$ is a non-canonical term such that:

   - If r is a recursive decomposition term then the variables binding the recursive call(s) are not referenced.
   - r is not a λ term.

This means that if we are to use computation lemmas exactly as other normalisation rules we must apply the following restriction on the term normalisations used to derive a computation lemma:

If u is a term to be rewritten in the derivation of a computation lemma we may only apply a normalisation rule to $t \preceq u$ provided $\forall r.t \prec r \preceq u \Rightarrow r$ is a non-canonical term such that:

- If r is a recursive decomposition term then the variables binding the recursive call(s) are not referenced.
- r is not a λ term.

Once this restriction is applied, it is entirely straightforward to show that the addition of a computation lemma is a conservative extension of the logic. Consider rewriting a goal conclusion or hypothesis $T[t'_1]$ to $T[t'_n]$ using a computation lemma $t_1 \rightarrow t_n$.

This means that there exists a series of rewritings $t_1 \rightarrow \ldots \rightarrow t_n$ such that the term $r_i(\preceq t_i)$ normalised at each step satisfies the condition above. Therefore, since $t'_1 \preceq T$ must be in a context where it may be normalised there exists a sequence of rewritings

$$T[t'_1] \rightarrow \ldots \rightarrow T[t'_n]$$

such that the terms normalised at each step $r'_i \preceq t'_i \preceq T$ satisfy the restriction on the contexts in which normalisation rules apply.

Therefore $T[t'_1]$ can be rewritten to $T[t'_2]$, without the computation lemma, using the existing normalisation rules.

QED

Given the uncomplicated nature of this extension, and the obvious benefits it brings, it is somewhat surprising that it was not incorporated in the original logic.

5.8 Summary

In this chapter we have presented an over-view of NuPRL type-theory and the characteristics it presents when used as the basis for an automatic theorem prover for program proofs. In this role NuPRL type-theory has two main advantages over the classical first-order logic conventionally used in automatic theorem provers:

- It is a much more expressive formalism. The built-in programming language provided by its term normalisation system provides a very flexible mechanism for expressing procedures Since it is a higher-order logic it can deal with higher order programs as readily as first order programs.

- It provides a very elegant and flexible basis for deductive program synthesis. The propositions-as-types principle on which the logic is based allows a correct program to be synthesised as a witness to a theorem trivially constructed from the program's specification.

These advantages are, however, not without a price. NuPRL type-theory has several serious weaknesses as the basis for an automatic theorem prover for program proofs.

- The introduction of user-defined data-types is somewhat messy and inelegant. Recursive data-types can only be realised as combinations of existing types rather than as types in their own right. The lack of an efficient mechanism for dealing with exceptions causes serious difficulties in proofs dealing with non-trivial programs.

- Proofs in NuPRL type-theory are considerably more bulky than their equivalent in classical logics because of the need to repeatedly prove well-formedness sub-goals. Although these well-formedness proofs are generally trivial, their sheer length means that they adversely affect the performance of an automatic theorem prover based on type-theory.

- In its original form, the NuPRL logic also makes proofs based on the application of computation rules very expensive. Each proof based on computation always has to be performed in full using low-level computation rules because lemmas cannot be effectively used to capture common sub-proofs based computation.

  This problem can, however, be solved by a simple conservative extension of NuPRL type-theory.

- The facilities for reasoning about partial functions – non-terminating programs – are currently rather in something of a state of flux. No generally accepted approach has yet been defined.

- As basis for an automatic theorem prover NuPRL type-theory also suffers from the absence of unification and a very high branching rate in the search-space for proofs. There are a large number of inference rules, some of which can be applied at any time, and all terms must be fully instantiated when they appear in a proof, there is no provision for binding meta-variables later in a proof to suit constraints as they emerge.

  These problems can however be dealt with using meta-level reasoning techniques to plan proofs before constructing them at the object level [BvH*88, Plu85,SBB*82].

NuPRL type-theory also mandates a somewhat unusual approach to the introduction of defined terms. Defined functions are best introduced indirectly as the *witnesses* of proofs showing the inhabitation of their types, rather than directly by defining terms. In effect, the introduction of a function definition is merged with the proof that it is well-formed.

Chapter 6

Conclusion

In this chapter we summarise the work we have presented in this thesis and reflect on possible directions for further work in the same area.

6.1 Summary

If we wish to construct software this is dependable and remains dependable after maintenance then we must guarantee that programs are correct. That is, we must show they satisfy their specifications expressed in some well-defined logical language. Unfortunately, the development methods that can ensure correctness are difficult to apply in practice because of the need for extremely lengthy formal proofs whose manual construction is prohibitively time-consuming. The practical development of dependable software is therefore predicated on the development of powerful automatic theorem provers capable of mechanising these proofs. An important problem such automatic theorem provers must solve is the mechanisation of induction. The theorem prover must somehow construct an induction schemata appropriate to the theorems to be proved. That is, it must construct inductions that produce induction sub-goals that allow the proof to be completed once induction has been applied. It is this problem that we have tackled in our thesis.

The starting point for our work has been recursion analysis – the procedure for constructing appropriate inductions developed by Boyer and Moore for their automatic theorem prover for pure LISP. Our methodology, since Boyer and Moore's work does not include a well-defined theory to explain their procedure, has been one of rational reconstruction. We initially constructed a provisional theory for recursion analysis on the basis of a simple meta-theory of inductive proofs and Boyer and Moore's informal justification of their procedure. We then used this provisional theory in an analysis of the recursion analysis procedure to identify its major flaws. Finally, we employed the insights gained through this analysis to develop an improved recursion analysis procedure based in a well-defined, testable, theory of appropriate inductions. It this improved procedure, and the improved understanding captured in our theory of appropriate inductions that form the major contribution of this thesis.

Our procedure improves on the performance of the Boyer-Moore original, in several significant respects:

- It is capable of dealing with goals that contain any function with a definition that can be proved well-founded. Boyer and Moore's technique has serious flaws that mean it can often fail on goals referencing functions whose definitions have a slightly unusual conditional structure (See section 4–1).

- It is capable of dealing with goals that contain explicitly quantified variables.

- It deals correctly with goals that require inductions that are *repeated forms* of the inductions dual to terms in the goal.

- It uses a sound test to eliminate from consideration inductions likely to fail due to side-effects, and uses a more accurate scoring function to choose the induction to apply from the available alternatives.

- Additionally, we have proposed a procedure that allows Walther's method for mechanising well-foundedness proofs to be used to construct the bulk of

the "induction lemmas" mechanically. These induction lemmas currently have to be formulated manually. Our procedure furthermore, promises to considerably reduce the sensitivity of recursion analysis to ill-formed induction lemmas introduced manually (see sections 4.3 and 4.3.1).

On the basis of an implementation of this rationally reconstructed algorithm for the NuPRL type-theory logic, we have identified several weaknesses in this (and similar logics) that complicate their use in automatic theorem provers. Specifically: the inordinate cost of well-formedness subproofs, and the lack of facilities for defining and applying derived computation rules. In the case of the latter problem we have proposed a conservative extension to the NuPRL type-theory logic that allows it to be eliminated.

In the longer term, however, it is the theory of appropriate inductions that we have developed through this thesis that provides our main contribution. A serious problem in Artificial Intelligence research has been the difficulty encountered in building on existing work. This has made progress sporadic and slow. This difficulty, we suggest, has in large part been due to the lack of well-defined refutable theories in AI research. If a theory is only ever presented in implicit form encoded in the design of a complex algorithm it is very difficult to refute in a precise way. The precise flaws that cause the program (and hence the theory) to break down become very hard to pin down, thus greatly complicating the development of a theories (and hence AI programs) that are definitively better.

The theory of appropriate inductions we have developed (briefly summarised below) though undoubtedly imperfect, is refutable and thus provides a solid foundation for further work.

6.2 A Summary of a Theory of Appropriate Inductions

In successful inductive proofs, induction introduces induction hypotheses that enable recursive terms to be eliminated from the induction conclusion by hypothesis (see any of the example proofs in this thesis). An appropriate induction must therefore enable the recursive terms in the induction hypotheses and conclusion to be rewritten so they match.

Induction hypotheses are, furthermore, *reduced* instances of the induction conclusion. An appropriate induction must therefore produce reduced instances of recursive terms in the induction hypotheses that match those that can be introduced into the induction conclusion. In general, this means that the recursive terms in the induction hypotheses must match those that can be produced in the induction conclusion by symbolic evaluation. Thus, heuristically, an induction must *subsume* the dual inductions of some set of recursive terms in a goal to be proved if it is to be appropriate to that goal.

This heuristically necessary condition is not, however, sufficient to ensure that an induction is appropriate. There are three distinct ways in which an induction that subsumes inductions dual to recursive terms in a goal may fail to enable recursive terms to be eliminated:

1. The induction may produce uneliminable *side-effects* in recursive terms whose dual inductions it does not subsume.

2. The induction may produce only partial matches between induction hypothesis and induction conclusion on recursive terms it is only approximately dual to.

3. The induction may fail to enable a *ripple-out* from recursive terms whose duals it subsumes that are nested within other recursive terms.

Thus, heuristically, an induction is appropriate if and only if:

- It subsumes some subset of inductions dual to recursive terms in the goal.
- It is free of uneliminable side-effects in recursive terms whose dual inductions it does not subsume.
- It is likely to enable exact matches between induction hypothesis and conclusion on recursive terms they are only approximately dual to.
- It is likely to enable rippling-out from the recursive terms whose dual induction it subsumes that are nested within other recursive terms.

In the remainder of this chapter we consider the weaknesses in our version of recursion analysis (theory and implementation) and propose directions for further work.

6.3 Limitations and Further Work

The recursion analysis procedure we have presented in this thesis is significantly improved over the Boyer-Moore original. It is based on a well-defined theory of appropriate inductions, and will find appropriate inductions in a significantly wider variety of situations than the original. It is, however, still a long way from providing a complete solution to the problem of constructing appropriate induction schemata. Two key limitations – inherent in the theory of appropriate inductions upon which recursion analysis is based – remain.

6.3.1 Non Dual Inductions

The most obvious weakness of the recursion analysis approach is that it is inherently restricted in the appropriate inductions it can find. Recursion analysis is built on the heuristic that inductions appropriate to a particular goal are, in

general, based on the dual inductions of the recursive terms in that goal. This heuristic allows it to construct an important class of appropriate inductions – inductions that produce eliminable recursive terms through the application of symbolic evaluation and rippling out – very efficiently. Although this underlying heuristic is very effective and is solidly based in the meta-theory of induction, it is still only a useful approximation. There are some situations where the appropriate induction is *not* dual to any of the recursive terms in the goal to be proved and requires proof steps other than symbolic evaluation and rippling out to succeed. In these situations recursion analysis will either fail to find an appropriate induction at all, or will find only a second-best appropriate induction. Consider, for example, the goal:

$$\vdash even(x) \rightarrow even(times(x, y))$$

If we apply recursion analysis to this goal it suggests the induction schema dual to $even(x)$. This, unfortunately, is quite inappropriate as symbolic evaluation gives the step-case conclusion:

$$even(p(p(x)) \rightarrow even(times(p(p(x)), y))$$
$$\vdash even(p(p(x)) \rightarrow even(plus(y, plus(y, times(p(p(x)), y))))$$

which is extremely difficult to ripple-out. The appropriate induction is in fact that dual to $times(y, x)$. This gives the induction step-case:

$$even(x) \rightarrow even(times(x, p(y))) \vdash even(x) \rightarrow even(times(x, y))$$

which, by applying the commutativity of *times* can be rewritten to:

$$even(x) \rightarrow even(times(x, p(y))) \vdash even(x) \rightarrow even(times(y, x))$$

Symbolic evaluation rewrites this goal to:

$$even(x) \rightarrow even(times(x, p(y))) \vdash even(x) \rightarrow even(plus(x, times(p(y), x)))$$

which gives the goal

$$even(x) \rightarrow even(times(x, p(y))) \vdash even(x) \rightarrow even(plus(x, times(x, p(y))))$$

which ripples out immediately through the application of the lemma $even(a) \wedge even(b) \rightarrow even(plus(a, b))$

6.3.2 Existential Quantifiers

The other major limitation in the recursion analysis approach is that it is prone to fail when applied to goals that contain existentially quantified variables. Consider, for example, the goal:

$$\vdash sorted(l) \wedge sorted(r) \rightarrow \exists m.sorted(m) \wedge permutation(m, append(l, r))$$

in which we prove the existence of a sorted merge of two sorted lists. We assume *sorted* is defined:

$$\begin{aligned} sorted(l) \equiv\ & \text{if } l = nil \text{ then } true \\ & \text{else if } tl(l) = nil \text{ then } true \\ & \text{else if } hd(l) \leq hd(tl(l)) \text{ then } sorted(tl(l)) \\ & \text{else } false \end{aligned}$$

If we apply recursion analysis, the induction schema suggested is that dual to *sorted* – step-wise induction on the length of the list l. This, however, is quite inappropriate. Once connectives have been eliminated we obtain the step-case goal:

$$\begin{aligned} & \neg l = nil, sorted(l), sorted(r), \\ & sorted(m'), permutation(m', append(tl(l), r)) \\ \vdash\ & \exists m.sorted(m) \wedge permutation(m, append(l, r)) \end{aligned}$$

Any further progress on this sub-goal is blocked because there is no workable solution term we can supply for m. The appropriate induction schema for this goal is step-wise induction on l or r, which gives us the induction sub-goals:

$$\begin{aligned} & l = nil, sorted(r) \\ \vdash\ & \exists m.sorted(m) \wedge permutation(m, append(l, r)) \end{aligned}$$

$$\begin{aligned} & \neg l = nil, r = nil, sorted(l) \\ \vdash\ & \exists m.sorted(m) \wedge permutation(m, append(l, r)) \end{aligned}$$

$$\begin{aligned} & \neg l = nil, \neg r = nil, sorted(l), sorted(r), \\ & sorted(m'), permutation(m', append(tl(l), r)), \\ & sorted(m''), permutation(m'', append(l, tl(r))) \\ \vdash\ & \exists m.sorted(m) \wedge permutation(m, append(l, r)) \end{aligned}$$

The base cases of this induction can be easily proved with the solution term r and nil respectively. The step-case requires a case-analysis on $hd(l) \leq hd(r)$. The $hd(l) \leq hd(r)$ case can then proved with the solution term $m = cons(hd(l), m')$, whilst the $\neg hd(l) \leq hd(r)$ can be proved using $m = cons(hd(r), m'')$.

The underlying flaw in the recursion analysis procedure that causes failures of this type is that it treats existentially quantified variables as constants, when in fact they act as place-holders for *functions*. The solution terms for an existentially quantified variable are, in effect, a definition for a function to replace that variable. This function is *recursive* in situations where the solution terms make use of the induction hypotheses. The function's recursion scheme is the dual of the induction applied. Consider, for example, replacing m in the proof above with the function $m(l, r)$. The solution terms provided for m given the following equational definition for $m(l, r)$:

$$
\begin{array}{l}
l = nil \rightarrow m(l, r) = r \\
\neg l = nil \wedge r = nil \rightarrow m(l, r) = l \\
\neg l = nil \wedge \neg r = nil \wedge hd(l) \leq hd(r) \rightarrow m(l, r) = cons(hd(l), m(tl(l), r) \\
\neg l = nil \wedge \neg r = nil \wedge \neg hd(l) \leq hd(r) \rightarrow m(l, r) = cons(hd(r), m(l, tl(r))
\end{array}
$$

which, in our usual notation, may be written:

$$
\begin{array}{ll}
m(l, r) \equiv & \text{if } l = nil \text{ then } r \\
& \text{else if } r = nil \text{ then } l \\
& \text{else if } hd(l) \leq hd(r) \text{ then } cons(hd(l), m(tl(l), r)) \\
& \text{else } cos(hd(r), m(l, tl(r)))
\end{array}
$$

Thus, in general, if an induction is to be appropriate for a goal with existentially quantified variables, it must be dual to the definitions for skolem functions for these variables. Recursion analysis – and our theory of appropriate inductions – tend to fail on goals with existentially quantified variables because they do not take account of this constraint on the choice of inductions.

6.3.3 Improving on Recursion Analysis

The characteristics of these remaining flaws in our theory suggest that significant improvements over recursion analysis require the use of a rather different approach. The recursion analysis procedure is a fixed, special-purpose algorithm that operates as a induction selection "black-box". It finds an appropriate induction when required, largely on the basis of heuristics about the way inductions typically succeed. It reasons only in a very rigid and superficial way about the proof steps needed to eliminate recursive terms after an induction has been applied.

A theorem prover that deals with the situations we have identified above would, however, have to that reason in a flexible way about the interaction between induction with the proof steps performed after induction. If we are to find inductions that succeed through proof steps other than symbolic evaluation and rippling out, we must take into account how a proof might proceed after induction. Similarly, if we wish to find inductions that allow solution terms to be found, we need to take into account how the proofs steps relating to solution terms might proceed after induction.

The idea of an *procedure* to find appropriate inductions is therefore, we feel, something of a dead end. The development of a procedure performing significantly better than recursion analysis would involve the construction of ever more complex, ad hoc data-structures and algorithms. These would be awkward to develop and very difficult to relate to an improved overall understanding of induction proofs.

What is required instead, we suggest, is a theory of appropriate inductions suitable for application in a framework for guiding proofs through mechanised meta-level reasoning [SBB*82,Ste82,Sil85]. The problem of successfully applying induction is essentially one of plan-formation. A sequence of inference rule applications has to be found that satisfies a set of constraints expressing a particular theory of appropriate inductions. The most sensible approach to solving this

problem would therefore appear to be the development of a deductive planning system (see for example [Sil85,Plu85]), rather than a fixed algorithm.

- The use of a planner would allow the necessary flexibility required to deal with the problem of solution terms etc. The choice of induction could be driven by planning the rippling-out and solution terms needed to allow the induction to succeed. The selection of an induction could thus be based on the actual goal to be achieved – the elimination of recursive terms – rather than imprecise heuristics such as those applied in the recursion analysis procedure.

- The constraints to be satisfied by an induction and the proof operations available to satisfy them could be expressed in a clean declarative way. The criteria for the planning of an induction would thus be explicitly available for inspection and modification rather than hard-wired into an opaque algorithm.

  This would allow the planning of induction steps to be integrated into meta-level deduction based schemes for planning proofs as whole, and/or learning new proof strategies [Sil85,Des87].

The key problems that remain to be solved are therefore an improved meta-theory of induction steps to cover solution terms etc, and a planning formalism capable of mechanising the use of such a theory to plan induction steps. Promising initial results in these area have already been demonstrated in work by Bundy et al [BvH*88] in which we have collaborated.

6.4 Other Further Work

In additional to the major area for further work outlined above, there are several important topics related to (or arising from) this thesis that require further work.

6.4.1 Automated Induction Lemma Construction

In section 4.3.1 we proposed a procedure for the mechanical recognition of redundant hypotheses in proofs, and hence the mechanical recognition of redundant pre-conditions in induction lemmas. Initial analysis suggests that this procedure, in combination with Walther's procedure for mechanising well-foundedness, proofs provides a effective method for mechanising the construction of induction lemmas for recursion analysis. Unfortunately, time-pressure (and the lack of a suitable automatic theorem prover) have prevented proper empirical testing of this proposal. A useful area for further work would thus be the implementation, testing, and development of this proposal within a practical automatic theorem prover. The further development of Walther's well-formedness proof procedure to eliminate the flaws outlined in section 4.3.4 would also provide a worthwhile topic for further research.

6.4.2 Generalisation

An important question that we have not addressed at all in this thesis is that of the relationship between generalisation and induction. The success of an induction very often depends on the application of an appropriate generalisation before induction is invoked [Pra71,Kre65].

Consider, for example, the theorem

$$\forall x, y : BinaryTree, z : List.append(flatten(x), append(flatten(y), z)) = append(append(flatten(x), flatten(y)), z)$$

where *flatten* is a recursively defined function returning the list of nodes in a tree.

$$flatten(t) \equiv \text{ if } t = emptytree \text{ then } nil \text{ else } cons(node(t), append(flatten(left(t)), flatten(right(t))))$$

This theorem can be proved quite easily if we generalise $flatten(x)$ to x' and $flatten(y)$ to y'. The goal we obtain is simply the associativity of *append*:

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

which we can prove by induction of x'.

However, if we attempt to prove this goal directly without the use of generalisation the proof fails. Induction on z gets us nowhere, and induction on x or y just piles up ever more deeply nested instances of *append*.

An important topic for further work is thus the further development of mechanised techniques for generalising goals to allow inductions to succeed.

6.4.3 When Induction should be Applied

A final issue that we have only touched on in passing is that of deciding when to apply induction in a proof. The Boyer-Moore theorem prover applies the simple heuristic of applying induction as a last resort after all other proof methods have failed to make further progress. Induction is applied last because it is the only proof method in the Boyer-Moore theorem prover that produces sub-goals more complicated than the goal it was applied to. It therefore makes sense to keep it last until any possible simplifications that can be achieved without induction have been accomplished. However, there is no reason to believe a priori that this approach would be appropriate in all automatic theorem provers, or even that it is the best solution for theorem provers based on Boyer and Moore's design. The problem of when to apply induction requires further investigation.

Bibliography

[Aub75] R. Aubin. Some generalization heuristics in proofs by induction. In *Actes du Colloque Construction: Amelioration et verification de Programmes*, Institut de recherche d'informatique et d'automatique, 1975.

[Aub76] R. Aubin. *Mechanizing Structural Induction.* PhD thesis, University of Edinburgh, 1976.

[Bac86] R. Backhouse. *On the Meaning and Construction of the Rules in Martin-Löf's Theory of Types.* Tech. Report CS 8606, University of Groningen, 1986.

[BD77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BGM82] R.S. Boyer, M.W. Green, and J.S. Moore. *The Use of a Formal Simulator to Verify a Simple Real Time Control Program.* Technical Report ICSA-CMP-29, University of Texas at Austin, 1982.

[BHHW88] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The karlsruhe induction theorem proving system. In *9th International Conference on Automated Deduction*, 1988.

[Biu88] S. Biundo. Automated synthesis of recursive algorithms as a theorem proving tool. In *European Conference on Artificial Intelligence*, 1988.

[BJ78] D Bjorner and C.B. Jones. *The Vienna Development Method: The Meta-language.* Springer, 1978. Lecture Notes in Computer Science.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[BM84] R.S. Boyer and J.S. Moore. Proof checking the RSA public key encryption algorithm. *American Mathematical Monthly*, (91(3)):181–189, 1984.

[BM88a] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Perspectives in Computing, Vol 23.

[BM88b] R.S. Boyer and J.S. Moore. Integrating decision procedures into heuristic theorem provers: a case study with linear arithmetic. In D. Michie, editor, *Machine Intelligence 11*, Oxford University Press, 1988.

[BMS86] R.M. Burstall, D.B. MacQueen, and D.T. Sannella. *Hope: An Experimental Applicative Language.* Technical Report XX, Dept. of Computer Science, Univ. of Edinburgh, 1986.

[Bun84] A. Bundy, editor. *Catalogue of Artificial Intelligence Tools*, Springer-Verlag, 1984. Second Edition.

[BvH*88] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. *A rational reconstruction and extension of recursion analysis.* Research Paper 419, Dept. of Artificial Intelligence, Edinburgh, 1988. In the proceedings of IJCAI-89.

[CAB*86] R.L. Constable, S.F. Allen, H.M. Bromley, et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[CFC58] H.B. Curry, R. Feys, and W. Craig. *Combinatory Logic.* North-Holland, 1958.

[CS77] K.L. Clark and S. Sickel. Predicate Logic: a calculus for deriving programs. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 419–420, IJCAI, 1977.

[dB70] G. de Bruijn, N. *The Mathematical Language AUTOMATH and some of its extensions. Lecture Notes in Mathematics*, Springer, 1970.

[dBS69] J.W. de Bakker and D. Scott. *A Theory of Programs.* Technical Report Unpublished Notes, IBM seminar, Vienna, 1969.

[Des87] R.V. Desimone. *Learning Control Knowledge within an Explanation-Based Learning Framework.* Research Paper 321, Dept. of Artificial Intelligence, Edinburgh, April 1987. Published in Progress in Machine Learning, (eds) Bratko & Lavrač, Sigma Press 1987.

[ea87] Bauer et al. *The Program Transformation System CIP-S. Lecture Notes in Computer Science*, Springer-Verlag, 1987.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics. EATCS Monographs on Theoretical Computer Science*, Springer-Verlag, 1985.

[Fea79] M.S. Feather. *A System for Developing Programs by Transformation.* PhD thesis, Dept. of Artificial Intelligence, Edinburgh, 1979.

[FK86] T. Fujita and H. Kanamori. Formulation of induction formulas in verification of prolog programs. In *8th Conference on Automated Deduction*, 1986.

[GMW79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer Verlag, 1979.

[Gog80] J.A. Goguen. How to prove algebraic inductive hypotheses without induction. In *5th Conference on Automated Deduction*, 1980.

[HH82] G. Huet and J.M. Hullot. Proof by induction in equational theories with constructors. *JCSS*, 25(2), 1982.

[HMM86] Robert Harper, David MacQueen, and Robin Milner. *Standard Ml.* Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[Hog81] C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.

[Hor88] C. Horn. *The Nurprl Proof Development System.* Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[How80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Academic Press, 1980.

[Jon79] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1979. PH International Series in Computer Science.

[Jou86] J.P. Jouannaud. Proofs by induction in equational theories without constructors. In *Proc. of IEEE Conf. on Logic in Computer Science*, 1986.

[Kan86] H. Kanamori, T. asd Seki. *Verification of Prolog Programs Using and Extension of Execution.* Technical Report TR-093, ICOT, 1986.

[KB70] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebra. In J. Leech, editor, *Computational problems in abstract algebra*, pages 263–297, Pergamon Press, 1970.

[KNZ86] D. Kapur, P. Narendran, and H. Zhang. Proof by induction using test sets. In *8th Conference on Automated Deduction*, 1986.

[Kre65] G. Kreisel. Mathematical logic. In T.L. Soaty, editor, *Lectures on Modern Mathematics vol. 3*, pages 95–197, Wiley, 1965.

[Mar79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover, August 1979. Published by North Holland, Amsterdam. 1982.

[Mar84] P. Martin-Löf. *Intuitionistic Type-theory. Studies in Proof Theory Lecture Notes*, BIBLIOPOLIS, 1984.

[Mit88] B. Mitchell. *A Tutorial on Inductive Completion.* Technical Report Unpublished Notes, SERC Recursion Worshop, Glasgow 1988, 1988.

[MNV71] Z. Manna, S. Ness, and J. Vuillemin. *Inductive methods for proving properties of programs.* Tech. Report AIM-145/STAN-CS-71-243, Stanford University, 1971.

[MW80] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[NY83] R. Nakajima and T. Yuasa. *The IOTA programming system: a modular programming environment. Lecture Notes in Computer Science*, Springer-Verlag, 1983.

[Per86] N. Perry. *Hope+ Draft Specification.* Technical Report IC/FPR/LANG/2.5.1/7, Imperial College, 1986.

[Plu85] D. Plummer. *Gazing: Using the Structures of the theory in theorem proving.* Working Paper 180, Dept. of Artificial Intelligence, Edinburgh, May 1985.

[Pra71] D. Prawitz. Ideas on result of proof theory. In J. Fensted, editor, *Proc. 2nd Scandinavian Logic Symposium*, pages 235–307, North Holland, 1971.

[San88] D. Sannella. *A survey of Formal Software Development Methods.* LFCS Report ECS-LFCS-88-56, University of Edinburgh, 1988.

[SBB*82] L. Sterling, A. Bundy, L. Byrd, R. O'Keefe, and B. Silver. Solving symbolic equations with PRESS. In J. Calmet, editor, *Computer Algebra, Lecture Notes in Computer Science No. 144.*, pages 109–116, Springer Verlag, 1982. Also available from Edinburgh as Research Paper 171.

[Sch79] A.H. Schoenfield. Personal communication. 1979. Letter expressing interest in PRESS as a teachable model of expert problem solving.

[Sil85] B. Silver. *Meta-level inference: Representing and Learning Control Information in Artificial Intelligence.* North Holland, 1985. Revised version of the author's PhD thesis, DAI 1984.

[Spi89] J.M. Spivey. *The Z notation, A Reference Manual.* Prentice-Hall, 1989. PH International Series in Computer Science.

[ST86a] D. Sannella and A. Tarlecki. Extended ml: an institution independent framework for formal program development. In *Proc. Workshop on Category Theory and Computer Programming*, Springer, 1986.

[ST86b] D. Sannella and A. Terlecki. *Specifications in an arbitrary institution.* Tech. Report CSR-184-85, Dept. of Computer Science, Univ. of Edinburgh, 1986.

[Ste82] L. Sterling. *IMPRESS - Meta-level concepts in theorem proving.* Working Paper 119, Dept. of Artificial Intelligence, Edinburgh, August 1982.

[Tam84] T. Tamaki, H.and Sato. Unfold/fold transformation of logic programs. In *2nd International Logic Programming Conference*, 1984.

[Toy86] Y. Toyama. How to prove equivalance of term rewriting systems without induction. In *8th Conference on Automated Deduction*, 1986.

[Vui73] J. Vuillemin. *Proof Techniques for Recursive Programming.* Tech. Report AIM-218/STAN-CS-73-293, Stanford University, 1973.

[Wal88] C. Walther. Argument-bounded algorithms as a basis for automated termination proofs. In *9th Conference on Automated Deduction*, pages 602–621, Springer-Verlag, 1988.

Appendix A

Implementational Conventions

A.1 Notation

The Oyster system within which we implemented our rational reconstruction of the recursion analysis algorithm represents NuPRL type-theory terms using Prolog's facilities for defining operator precedence grammars. As a result, the notation for NuPRL type-theory terms used in the proof-procedures and example runs listed in the following appendices differ somewhat from the conventional notation we have used so far. The differences relevant to these proof-procedures and examples are that:

- The sequent arrow, conventionally written $\vdash$, is written `==>`.
- The disjoint union type, conventionally written $A \mid B$, is written $A \backslash B$ in Oyster.
- The constructor for pairs (objects of Cartesian product type), conventionally written (A, B) is written A `&` B in Oyster.
- The function type, conventionally, written A`->` B is written A `=>` B.

  The function application term, conventionally written $A(B)$, is written A `of` B in Oyster.

The lambda term (function constructor term) conventionally written $\lambda V.A$ is written `lambda(`V, A`)` in Oyster.

- The conventional notation for a term T within which variables $V_1, \ldots, V_n$ are bound, $V_1, \ldots, V_n.T$ is replaced with the notation $[V_1, \ldots, V_n, T]$. Furthermore, commas replace semi-colons as field separators in non-canonical terms. For example, the term $p\_ind(s(0); 0; v, r.s(r))$ is written `p_ind(s(0),0,[v,r,s(r)])` in Oyster.

A.2 Primitives for Constructing Proofs

The Oyster system maintains proofs as named objects in the Prolog data-base. Each proof is stored as a tree of sub-goals, along with information distinguishing a "current sub-goal" in each proof. Theorems are simply complete proofs – proofs that have no sub-goals that remain to be proved.

Proofs are constructed by invoking various primitive procedures that allow a proof to be selected, inference rules applied to the current sub-goal, or the current sub-goal changed. Thus, despite the use of the Prolog language, proof-procedures are written in an imperative style. They are simply Prolog procedures that invoke the primitive procedures to navigate proofs and apply inference rules.

The main primitive procedures used in the code listed are:

- `add_thm(`$Name, TopLevGoal$`)`

  Add a new proof called $Name$ to the Prolog data-base. The new proof consists of a single node $TopLevGoal$, which becomes the current node for $Name$. The current proof and its current node are unaffected.

- `select(`$Name$`)`

  Make the proof called $Name$ the current proof.

- `pos(`P`)`

If P is unbound binds P to the position of the current proof sub-goal in the current proof-tree. If P is bound makes the current proof sub-goal in the current proof the sub-goal with the position P.

- `down(`N`)`

  Change the current proof sub-goal in the current proof to be the Nth child of the current proof sub-goal.

- `up`

  Change the current proof sub-goal in the current proof-tree to be the parent of the current proof sub-goal.

- `do(`*ProofProc*`)`

  Apply the proof procedure *ProofProc* to the current proof sub-goal in the current proof-tree. *ProofProc*s may take any one of the following special forms (which may be nested):

- *ProofProc* `then` *ProofProc'*

  Apply *ProofProc* to the current proof sub-goal, and then apply *ProofProc'* to each of the child sub-goals produced.

- *ProofProc* `then [`*ProofProc*$_1$`,` *ProofProc*$_2$`, ...,` *ProofProc*$_n$`]`

  Apply *ProofProc* to the current proof sub-goal, and then apply *ProofProc*$_1$ to its first child sub-goal, *ProofProc*$_2$ to its second child sub-goal, and so on. *ProofProc* must produce exactly n subgoals.

- *ProofProc* `then [`N_1`-`*ProofProc*$_1$`,` N_2`-`*ProofProc*$_2$`, ...]`

  Apply *ProofProc* to the current proof sub-goal, and then apply *ProofProc*$_1$ to its N_1th child, *ProofProc*$_2$ to its N_2th child, and so on. This fails if *ProofProc* does not produce N_ith child for any N_i.

- `apply(` *Rule* `)`

Apply the inference rule specified by *Rule* to the current proof sub-goal in the current proof-tree. For example:

`apply( intro( ...) )` applies the type introduction or formation rule appropriate to the conclusion of the current proof sub-goal.

`apply( elim(` *Hyp*`, ...) )` applies the type elimination rule appropriate to the hypothesis called *Hyp* in the current proof sub-goal.

`apply( seq(` *Type*`, ...) )` cuts the judgement *Type* into the hypothesis list.

- `apply(`*ProofProc*`)`

  Apply the proof procedure *ProofProc* to the current proof sub-goal in the current proof-tree, and then "fold" the resulting sub-proof. That is, retain only the unproved sub-goals from the resulting sub-proof, and makes these the children of the current proof sub-goal. The special *then* forms interpreted by `do` are also interpreted by `apply`.

A more complete description of the facilities available for writing proof procedures in Oyster can be found in [Hor88].

A.3 Proof Procedures

The proof-procedures used in the shells to introduce recursive data-types and recursive functions do not invoke the NuPRL logic directly. Instead, they make use of a family of low-level proof-procedures that provide a convenient interface to the NuPRL logic.

- `prove_mem` proves "trivial" well-formedness subgoals. It exhaustively applies computation rules, equality introduction rules and the hypothesis rule.

- *normalize_goal(Mask)* exhaustively applies computation rules to the conclusion of the current proof subgoal. All possible computation rule applications which do not match `Mask` are applied.

  `Mask` is a list of entries of the form `[Kind,TermPattern]`. Each such entry excludes computation rule applications of kind `Kind` on terms which match `TermPattern`. There are three different `Kinds`: `unfold` which corresponds to the expansion of definitions, `expand` which corresponds to the expansion of *term_of* terms, and `0` which corresponds any other computations.

- *normalize_hyp(Mask,Hypothesis)* acts just as `normalize_goal` except that it applies computation rules to a hypothesis in the current proof subgoal rather than its conclusion.

- *intro_hyps(Mask)* eliminates any leftmost universal quantifiers from the goal to be proved. Each hypothesis, that results is normalised through the application of `normalize_hyp`. For example, applying `intro_hyps([])` to the goal

  $\vdash x : pnat \text{-}\!> y : (lambda(x,x)ofpnat) \text{-}\!> G[x,y]$

  produces the sub-goal

  $x : pnat, y : pnat \vdash G[x,y]$

- `intro_type_hyps_upto(V)` eliminates leftmost universal quantifiers up to and including that binding the variable `V`.

- *proof_thru_lemma(Kind,Ask,Name,Witness)* applies a lemma indexed under `Kind` to prove the current proof subgoal. If the preconditions applicable lemmas do not appear in the hypothesis list of the subgoal, lemmas of type `precond_lemma` are applied to attempt to deduce them by forward chaining. If `Ask` is set to `ask`, the user will be prompted for an appropriate lemma if none can be found. The name of the lemma used is bound to `Name`. `Witness` is bound to a witness for the subgoal proved.

This procedure fails if the goal cannot be proved by applying lemmas in the manner described.

A.4 The Implementation

In addition to the conventions enforced by the Oyster system, the code listed in the following appendices also assumes several conventions of its own.

A.4.1 The Database

The various components of the recursion analysis procedure and shells communicate via the Prolog database:

- Associated with each function defined in the system is a "definition record" of the form

  `definition(Def, Args, WffArgsTypes, ArgsTypes==>Range, Kind )`

  indexed under the name of the name of the function. This record holds the information the system requires to fill in missing well-formedness arguments in function definitions:

  `Def` — the definition for the function[1]

  `Args` — the function's formal arguments.

  `WffArgsTypes` — the type(s) of the function's well-formedness arguments (if any).

  `ArgsTypes` — the type(s) of the functions other arguments.

  `Range` — the functions range type.

---

[1]Strictly speaking, this information is not required. It is only included for historical reasons.

`Kind` — The functions kind: destructor, constructor, recursive, etc.

This information is also used by the predicate `guess_type` which is used to guess a function's type given its definition.

- Each constructor function introduced into the system has associated with it a tuple of the form:

```
[ Name, Recog, RecDestrs, AllDestrs ]
```

indexed under the identifier `shell_constructor`. This tuple contains information needed by the recursive type introduction shell during the synthesis of the induction lemmas for the type the constructor function constructs:

`Name` — the name of the constructor function.

`Recog` — the name of the predicate that recognises objects constructed using the constructor function.

`RecDestrs` — the destructor functions that return components of `Name` constructions that have the same type as `Name` constructions themselves.

`AllDestrs` the destructor functions that return the components of `Name` constructions.

- The induction templates for recursively defined functions are recorded in tuples of the form:

```
[Name,Args,Measured,Measure,Schema,Bonus]
```

indexed under the identifier `template`.

  - `Name` is the name of the function the tuple records a template for.
  - `Args` are the formal arguments of the induction template.
  - `Measured` are the measured formal arguments of the induction template.

- `Measure` is the measure justifying well-foundedness of the induction template.
- `Schema` is the induction schema dual to `Name`.
- `Bonus` is the "nasty function bonus" to be added to the score of inductions that are based on the template. If `Schema` is based on a simple structural induction it is 0 otherwise it is 10.

A.4.2 Substitutions

Due to an unfortunate mistake early in its development, the Prolog implementation of the rational reconstruction, uses a representation for substitutions that differs from that used in the main text of this thesis. A substitution $[Tm_1/V_1, \ldots, Tm_n/V_n]$ is represented by the Prolog term:

$[V_1\text{-}Tm_1, \ \ldots, \ V_n\text{-}Tm_n]$

A.4.3 Induction Schemata

Induction schemata are represented as Prolog terms of the form:

[*PrecondList* ==>*StepSubs*, ...]

`PrecondList` is a list of pairs $WitName{:}Precond$ where $WitName$ is the name for a witness for the precondition $PreCond$. $StepSubs$ is a list of substitutions as described above. For example, induction schema with the abstract form

$$\frac{\vdash G}{\begin{array}{c} a \neq nil, \neg member(x,a), G[tl(a)/a, b/b], G[tl(a)/a, delete(hd(a),b)/b] \vdash G \\ a \neq nil, member(x,a), G[delete(x,a), b/b] \vdash G \end{array}}$$

is represented by the Prolog term

```
[ [w1:nilp(a)=>void, w2:member(x,a)=>void]
  ==> [[a-tl(a,w1), b-b], [a-tl(a,w1),b-delete(hd(a,w1),b)]],

  [w1:a \ne nil, w3:member(x,a)]
  ==> [[a-delete(x,a),b-b]]
]
```

in our implementation of recursion analysis for NuPRL type-theory.

Appendix B

The Recursive Type Shell

The following Prolog code implements our shell for the introduction of new recursive data-types in NuPRL type-theory. A sample run illustrating the introduction of a new recursive data-type is listed at the end of this appendix. A brief summary of the Prolog environment for which this code has been written can be found in appendix A.

B.1 Top-level Procedure

```
%**********************
%*
%*      shell_prin -   Top level module for implementing a pseudo-
%*                   "shell principle" for NuPRL.
%*
%* ENTRY:
%*    add_shell( typename( Type1, ..., Typen ),
%*                [ constructor1( dest11:Type11, ...,dest1n:Type1n1 ),
%*                  ...
%*                  constructorm( destm1:Typem1, ...,destmnm:Typemnm )
%*                ] )
%*
%*   Where Typeij = typename, or = Typei
%*
%* EXAMPLE:  (For full details see chapter 5 thesis)
%*
%*   add_shell( list(t:u(1)), [nil,cons(hd:t,tl:list)] )
%*
%* is realised as the NuPRL recursive type: rec(list,unary\ t#list)
%*
%*  cons(hd,tl) is realised as inr(hd&tl)
```

```
%*  nil is realised as inl(unit)
%*
%*  The destructor functions are straight-forwardly derived from this.
%*
%*  hd(cons) is decide(cons,[l,l],[r,spread(r,[l,r,l]))
%*  consp(l) is decide(cons,[l,{false}], [r,{true}])
%*
%* etc etc
%*
%*********************

add_shell( Type, RawConstrs  ) :-

    Type =.. [TypeName|ParmTypes],

    noisenl( 10 ),
    noise( 10, 'NEW INDUCTIVE TYPE: ' ),
    noisenl( 10, TypeName ),
    noisenl( 10 ),

    % **** Build the synthesis theorem for the term.
    any_overwrite_ok( TypeName ),

    noisenl(10),
    noise(10, Type ),
    noise(10, ' == ' ),
    noisenl(10, RawConstrs ),
    noisenl(10),

    % **** Extract type parameter variables
    ebagof( Parm, PT^member((Parm:PT),ParmTypes),  Parms ),

    % **** Build the realisation of the type as a NuPRL type-theory type
    % **** plus the terms for its constructors and destructors.
    !,
    build_ntt_type( RawConstrs, [], ConstrImpl, RawNttType ),
    NttType = rec(TypeName,RawNttType),

    % **** Introduce the type as the extract term of its type
    % **** package it in the appropriate definition.
    !,
    guess_type( pure, ParmTypes, NttType, BodyType ),
    curry_type( ParmTypes, BodyType, TypeType ),
    def_c_term( TypeName, TypeType, NttType, [type] ),

    % **** Introduce the constructors, constructor recognisers,
    % **** and destructors for the type
    def_appl( TypeName, Parms, TypeDef ),
    instantiated_list( RawConstrs, [TypeName-TypeDef], Constrs ),
    !,
    def_constrs( Constrs, ConstrImpl, TypeDef, ParmTypes ),
```

```
%**** Introduce a lemma capturing the structural induction
%**** for the new type.

!,
prove_shell_strcase( TypeDef, ParmTypes, Constrs, StrCaseLem),

%**** Synthesise the structural size measure for the defined type.
concat( TypeName, '_cnt', MeasName ),
!,
synth_struct_meas( TypeName, MeasName ),

%**** Prove the induction lemmas associated with the defined type
prove_shell_indlems( TypeDef, ParmTypes, Constrs, IndLemmas ),

%**** Prove the mutual exclusion and decidability of the
%**** recognisers.
!,
prove_shell_mutex( Constrs, LemmaNames ),

addtuple( shell, [Type,Constrs, StrCaseLem,IndLemmas, LemmaNames] ).
```

B.2 Implementation of Type in NuPRL Type-Theory

```
%*********************
%*
%*    build_ntt_type( +Constrs, +Injection, -Implementations, -NttType )
%*
%*  GIVEN:
%*    Constrs is a list of constructors (and related destructors)
%*    defined as for add_shell.
%*
%*    Injection is the injection of the constructors in Constrs into
%*    the recursive disjoint union used to implement them.
%*
%*  THEN:
%*    Implementations is the list of non-canonical terms implementing
%*    the constructors and the related destructors
%*
%*    NttType is a disjoint union containing the Constrs as implmented
%*    in Implementations
%*
%**********************

build_ntt_type( [Constr], Injection, [ConsImpl], NttType ) :-
    build_ntt_constr( Injection, Constr, ConsImpl, NttType ).
build_ntt_type( [Constr|RestConstrs], Injection,
                [Impl|RestImpl], NttType  ) :-
    build_ntt_constr( [inl|Injection], Constr, Impl, ConsType ),
    build_ntt_type( RestConstrs, [inr|Injection],
                    RestImpl, TlNttType ),
    NttType = (ConsType\TlNttType).


build_ntt_constr( Injection, Constr, Impl, ConsType ) :-
    reverse( Injection, RInjection ),
    def_appl( ConstrName, Destrs, Constr ),
    build_ntt_destrs( Destrs, RInjection, [], DestrImpl ),
    build_ntt_recog_impl( RInjection, ConstrName, RecogImpl ),
    build_ntt_const_impl( ConstrName, RInjection,
                          Destrs, ConsImpl, ConsType ),
    Impl = ConsImpl-RecogImpl-DestrImpl.

build_ntt_destrs( [Dest:_], Injection, Spread, [Impl] ) :-
    build_ntt_dest_impl( Dest, Injection, Spread, Impl ).
build_ntt_destrs( [Dest:_|RestDest], Injection, Spread,
                  [Impl|RestImpl] ) :-
    build_ntt_dest_impl( Dest, Injection, [l|Spread], Impl ),
    build_ntt_destrs( RestDest, Injection, [r|Spread], RestImpl ).
build_ntt_destrs( [], _, _,  [] ).
```

```
%******
%*
%*  build_ntt_const_impl( Name, Injection, Destrs, Impl, Type )
%*
%*    GIVEN THAT:
%*  Name is the name of a constructor function of a recursive data-type
%*
%*  Injection is the injection of Name in the disjoint union
%* of constructor types making up the recursive data-type.
%*  Destrs are the components of Name
%*
%*   THEN:
%*   Impl is the implementation of Name in NuPRL type-theory
%*   Type is the NuPRL type-theory type for Name
%*
%*****

build_ntt_const_impl( ConstrName, [inl|RestInj], Destrs,
                      inl( RestImpl), Type ) :-
    build_ntt_const_impl( ConstrName, RestInj, Destrs, RestImpl, Type ).
build_ntt_const_impl( ConstrName, [inr|RestInj], Destrs,
                      inr( RestImpl ), Type ) :-
    build_ntt_const_impl( ConstrName, RestInj, Destrs, RestImpl, Type ).

build_ntt_const_impl( _,[], [DestNm:DestType], DestNm, DestType ).
build_ntt_const_impl( ConstrName, [], [DestNm:DestType|Rest],
                      (DestNm & RestImpl), (DestType # RestType) ) :-
    build_ntt_const_impl( ConstrName, [], Rest, RestImpl, RestType ).

build_ntt_const_impl( _, [], [], unit, unary ).
```

```
%******
%*
%*  build_ntt_dest_impl( Name, Injection, Spread, Destrs, Impl, Type )
%*
%*    GIVEN THAT:
%*  Name is the name of a destructor function of a recursive data-type
%*
%*  Injection is the injection of Name in the disjoint union
%* of constructor types making up the recursive data-type.
%*
%*  Spread is the path in the the implemention of the
%* constructor (a binary tree of pairs) that defines
%* the component of the constructor that Name is supposed to access.
%*
%*   THEN:
%*   Impl is the implementation of Name in NuPRL type-theory
%*
%*****

build_ntt_dest_impl( Dest, [inl|RestInj], Spread, Impl ) :-
    Impl = decide( Dest, [l,NestImpl], [r,r] ),
    build_ntt_dest_impl( l, RestInj, Spread, NestImpl ).
build_ntt_dest_impl( Dest, [inr|RestInj], Spread, Impl ) :-
    Impl = decide( Dest, [l,l], [r,NestImpl] ),
    build_ntt_dest_impl( r, RestInj, Spread, NestImpl ).
build_ntt_dest_impl( Dest, [], Spread, Impl ) :-
    reverse(Spread,RSpread),
    build_ntt_dest_impl( Dest, RSpread, Impl ).

build_ntt_dest_impl( Dest, [l|RestSpread], Impl ) :-
    Impl = spread( Dest, [l,r, NestImpl] ),
    build_ntt_dest_impl( l, RestSpread, NestImpl ).
build_ntt_dest_impl( Dest, [r|RestSpread], Impl ) :-
    Impl = spread( Dest, [l,r, NestImpl] ),
    build_ntt_dest_impl( r, RestSpread, NestImpl ).
build_ntt_dest_impl( Dest, [], Dest ).
```

B.3 Sample Run

```
| ?- add_shell( tree(t:u(1)), [etree,node(val:t,left:tree,right:tree)]).

NEW INDUCTIVE TYPE: tree

tree(t:u(1)) == [etree,node(val:t,left:tree,right:tree)]

INTRODUCING NEW CONSTRUCTIVE NON-RECURSIVE TERM: tree

tree == t:u(1)=>u(1)
    body: rec(tree,unary\t#tree#tree)

INTRODUCING NEW CONSTRUCTIVE NON-RECURSIVE TERM: etree

etree == t:u(1)=>tree(t)
    body: inl(unit)

INTRODUCING NEW CONSTRUCTIVE NON-RECURSIVE TERM: etreep

etreep == t:u(1)=>etree:tree(t)=>{bool}
    body: decide(etree,[l,{true}],[r,{false}])

INTRODUCING NEW CONSTRUCTIVE NON-RECURSIVE TERM: node

node == t:u(1)=>val:t=>left:tree(t)=>right:tree(t)=>tree(t)
    body: inr(val&left&right)

INTRODUCING NEW CONSTRUCTIVE NON-RECURSIVE TERM: nodep

nodep == t:u(1)=>node:tree(t)=>{bool}
    body: decide(node,[l,{false}],[r,{true}])

INTRODUCING NEW DESTRUCTIVE NON-RECURSIVE TERM: val

val == t:u(1)=>val:tree(t)=>w:j(nodep(t,val))=>t
    body: decide(val,[l,l],[r,spread(r,[l,r,l])])

INTRODUCING NEW DESTRUCTIVE NON-RECURSIVE TERM: left

left == t:u(1)=>left:tree(t)=>w:j(nodep(t,left))=>tree(t)
    body: decide(left,[l,l],[r,spread(r,[l,r,spread(r,[l,r,l])])])

INTRODUCING NEW DESTRUCTIVE NON-RECURSIVE TERM: right

right == t:u(1)=>right:tree(t)=>w:j(nodep(t,right))=>tree(t)
    body: decide(right,[l,l],[r,spread(r,[l,r,spread(r,[l,r,r])])])


PROVING CASE-ANALYSIS LEMMA FOR: tree
```

```
tree_strcaselem ==
    t:u(1)=>goal:(tree(t)=>u(1))=>
    s:tree(t)=>
    a:(goal of etree(t)#
       val:t=>left:tree(t)=>
       right:tree(t)=>goal of node(t,val,left,right)
      )=>
    goal of s


SYNTHESISING STRUCTURAL MEASURE ON: tree

tree_cnt == t:u(1)=>m:tree(t)=>pnat


PROVING INDUCTION LEMMA FOR left APPLIED TO A tree

tree_cnt_left_ilem ==
    t:u(1)=>
    node:tree(t)=>
    w:j(nodep(t,node))=>
    tree_cnt(t,left(t,node,w))<*tree_cnt(t,node)

PROVING INDUCTION LEMMA FOR right APPLIED TO A tree

tree_cnt_right_ilem ==
    t:u(1)=>
    node:tree(t)=>
    w:j(nodep(t,node))=>
    tree_cnt(t,right(t,node,w))<*tree_cnt(t,node)

PROVING MUTUAL EXCLUSION OF SHELL RECOGNISER: etreep WITH [nodep]

etreep_mutex ==
     t:u(1)=>
     etree:tree(t)=>
     v0:(j(nodep(t,etree))=>void)=>
     j(etreep(t,etree))


PROVING MUTUAL EXCLUSION OF SHELL RECOGNISER: nodep WITH [etreep]

nodep_mutex ==
    t:u(1)=>
    node:tree(t)=>
    v0:(j(etreep(t,node))=>void)=>
    j(nodep(t,node))


yes
| ?-
```

Appendix C

The Recursive Function Shell and Definition Time Analysis

The following Prolog code implements our shell for the introduction of new recursive functions in NuPRL type-theory. For reasons of efficiency, the definition time analysis phase of recursion analysis is tightly integrated with this shell. A sample run illustrating the introduction of a new recursive function, and the construction of its induction templates is listed at the end of this appendix. A brief summary of the Prolog environment for which this code has been written can be found in appendix A.

C.1 Top-level Procecure

```
%************************
%*
%*
%*  def_rec_fn( Definition )
%*  -  Synthesise a new recursive function from the definition
%* Definition and construct its induction templates.
%*
%* Definition ::= Name(Name,\ldots) = Rhs
%* Rhs ::= if Body then Rhs else Rhs | Body
%* Body ::= NuPRL term possibly with some well-formedness arguments
%* omitted.
%*
%************************


def_rec_fn( Def ) :-
```

```
%**** Guess / Check the type of the recursive function being
%**** defined through an analysis of its definition.  In order to
%**** allow the type-deducer to find the type introduce a
%**** temporary definition record for the function with
%**** meta-variables in the appropriate places.

Def = (Lhs = Rhs),
Lhs =.. [Name|Args],
typing_pairs_of( Args, _DomainTypes, ArgsWithTypes ),
noisenl( 10 ),
noise( 10, 'INTRODUCING NEW (RECURSIVE) FUNCTION: ' ),
noisenl( 10, Name ),
noisenl( 10 ),
any_overwrite_ok( Name ),
record_obj( Name,
            definition( Lhs = 0, Args, Args,
                        [],
                        ArgsWithTypes ==> Range,
                        [recursive] ),
            _ ),
noise( 100, 'Guessing type of function...' ),
!,
guess_check_type( ArgsWithTypes, Rhs, Range ),
noisenl( 100, 'Done.' ),

curry_type( ArgsWithTypes, Range, FuncType ),
noisenl(10),
noise(10, Name ),
noise(10, ' == ' ),
noisenl(10, FuncType ),
noisenl(10),

%**** Fill in well-formedness arguments to give a wff definition
%**** In order to allow the necessary proofs to go through, give
%**** the defined function a temporary definition of the
%**** appropriate type.

noisenl( 10, 'Filling in well-formedness arguments...' ),
build_tmp_def( Lhs, Range, ArgsWithTypes ),
!,
wff_definition( Rhs, ArgsWithTypes, WffRhs ),
noisenl( 10, 'Done.' ),

% **** Build raw induction schema corresponding to recursive defn

noise( 10, 'Constructing dual induction schema...' ),
!,
induction_schema( Lhs, WffRhs, RawSchema ),
noisenl( 10, 'Done.' ),
noise_schema( 100, RawSchema ),
noisenl( 100 ),
```

```
% **** Collect the functions that, by type, might prove to be
% **** wf measures for the induction

noise( 10, 'Collecting possible justifying measures...' ),
!,
poss_measures( ArgsWithTypes, Measures ),
noisenl( 10, ' Done.' ),
noisenl( 100, Measures ),
noisenl( 100 ),

%**** Annotate the induction schema with the induction lemmas that
%**** might allow it to be proved well-founded (see chapter 3
%**** of thesis).

!,
noisenl( 10, 'Finding measures applicable to induction...' ),
noisenl( 10, '    (This could take a while!)' ),

cases_with_measures( RawSchema, ArgsWithTypes,
                     Measures, RawSchemaWithMeas ),
noisenl( 10, 'Done.' ),
noise_schema( 200, RawSchemaWithMeas ),
noisenl( 200 ),

%**** Find the (lexicographic combinations of) measures that will
%**** show the induction well-founded via existing induction
%**** lemmas.   Out of this subset find those whose induction
%**** lemmas can actually be shown to apply, and then keep only
%**** those that do not measure a superset of the variables
%**** measured by other measures found.

noisenl( 10, 'Finding measures justifying induction/recursion...' ),
!,
wf_measures( RawSchemaWithMeas, WfMeasures ),
!,
strip_all_supsets( WfMeasures, MinWfMeasures ),
noisenl( 10, MinWfMeasures ),
noisenl( 10, 'Done.' ),

%**** Synthesize the function using the wff definition (see
%**** synth_rec_fn below)

synth_rec_fn( Lhs, WffRhs, FuncType, MinWfMeasures ),

ctheorem( Name ) =: P,
!,
check_status( Name, P, complete ),


def_appl(Name,Args,DefHead),
```

```
curried_appl( term_of(Name), Args, DefBody ),
add_def( (DefHead<==>DefBody) ),

%**** Replace the provisional definition used to guess type,
%**** definition with proper definition based on extract term of
%**** proof.   Then record the templates corresponding to the
%**** various well-founded measures found.

record_obj( Name,
            definition( (DefHead<==>DefBody), Args, Args,
                        [],
                        ArgsWithTypes ==> Range,
                        [recursive] ),
            _ ),
classifications( Name, ArgsWithTypes ==> Range,
                 DefHead<==>DefBody ),

erasetuple( template, [Name, _, _, _, _, _] ),
(
  (
  member( Meas, MinWfMeasures ),
  T =.. [x|Meas],
  free_vars( T, Measured ),

  %**** Replacing the pre-conditions of the raw induction schema
  %**** with those of the induction lemmas used to prove it
  %**** reduces the measure Meas.  (This gives us the induction
  %**** template).

  tidy_schema( RawSchemaWithMeas, Meas, Args, Templ ),
  noise_schema( 100, Templ ),
  noisenl( 100 ),

  nasty_bonus( RawSchema, Measured, NastyBonus ),
  addtuple( template, [Name, Args, Measured,
                       Meas, Templ, NastyBonus] ),
  fail)
  ;
  true
),

%**** Finally, remind the the user of any promises he made during
%**** the definition of this function.
unproved_lemmas.
```

C.2 Construction of Dual Induction Schema

```
%*************
%*
%*   induction_schema( +Lhs, +WffDef, -Schema )
%*
%* Given that:
%*  Lhs is the l.h.s of a recursive function defintion
%*
%*  WffDef is the body of the definition with well-formedness
%* arguments filled in.
%*
%* Then:
%*  Schema is a raw induction schema dual to the recursive function
%* definition.
%*
%* Schema still contains irrelevant pre-conditions etc etc.
%*
%*  We construct Schema
%* by looking for recursive references in each case of the definition,
%* and collecting the instantiations for the formal arguments of the
%* function in these recursive references.  The set of distinct
%* instantiations in each case provides the step-substitutions for the
%* corresponding case of the induction schema.
%*
%*************

induction_schema( Lhs, WffDef, Schema ) :-
    induction_schema_cases(  WffDef, Lhs, [], [], Schema ).
```

```
induction_schema_cases( (wffwits WffWits boundin Body),
                        Lhs, Hyps, CondSubs, SchemaCases ) :-

    %**** Well-formedness witnesses deduced from hypothesis list

    pairing_of(WffHyps,_,WffWits),
    append(WffHyps,Hyps,NextHyps),
    induction_schema_cases( Body, Lhs, NextHyps,
                    CondSubs, SchemaCases ).


induction_schema_cases( (if (_-_-CCond) then CThen else CElse),
                        Lhs, Hyps, CondSubs, SchemaCases ) :-

    %**** Redundant Conditional

    instantiations_recrefs( Lhs, CThen, ThenInsts ),
    instantiations_recrefs( Lhs, CElse, ElseInsts ),
    seteq( ThenInsts, ElseInsts ),
    instantiations_recrefs( Lhs, CCond, RecInsts ),
    union( RecInsts, CondSubs, BranchCondSubs ),

    induction_schema_cases( CThen, Lhs, Hyps,
                            BranchCondSubs, ThenCases ),
    induction_schema_cases( CElse, Lhs, Hyps,
                    BranchCondSubs, ElseCases ),
    union( ThenCases, ElseCases, SchemaCases ).

induction_schema_cases( (if (TW-FW-CCond) then CThen else CElse),
                        Lhs, Hyps, CondSubs, SchemaCases ) :-
    %**** Non-Redundant Conditional

    instantiations_recrefs( Lhs, CCond, RecInsts ),
    union( RecInsts, CondSubs, BranchCondSubs ),

    induction_schema_cases( CThen, Lhs, [TW:CCond|Hyps],
                            BranchCondSubs, ThenCases ),
    induction_schema_cases( CElse, Lhs, [FW:(CCond=>void)|Hyps],
                            BranchCondSubs, ElseCases ),
    append( ThenCases, ElseCases, SchemaCases ).

induction_schema_cases( DefBody, Lhs, Hyps, CondSubs, Case ) :-
    instantiations_recrefs( Lhs, DefBody, BodySubs ),
    union( BodySubs, CondSubs, CaseSubs ),
    ( (CaseSubs = [], Case = []) ; Case=[(Hyps=>CaseSubs)] ),
    !.
```

C.3 Collecting Applicable Induction Lemmas

```
%*****************
%*
%*  cases_with_measures( +IndCases, +ArgsWithTypes,
%*                       +Measures, -CasesWithMeas )
%*
%*  - CasesWithMeas is the induction schema IndCases with each
%*  step-subsitution associated with the wf measures
%*  from Measures that that have an induction lemma that shows them
%*  reduced under the step-susbtitution.
%*
%*
%*****************

cases_with_measures( [(Pre=>Subs)|RestCases], ArgsWithTypes, Measures,
                     CasesWithMeasures ) :-
    cthm =: OldThm,
    reverse(Pre,RPre),
    append( ArgsWithTypes, RPre, Hyps ),
    curry_type( Hyps, pnat, Goal),
    add_thm( '$$prove_with_ilem$$', [] ==> Goal ),
    select( '$$prove_with_ilem$$' ),
    (apply( intro_hyps( [[unfold,_],[expand,_]] ) );true),
    down(1),
    pos(P),
    noisenl(100),
    subs_with_measures( Subs, Measures, P, Hyps, SubsWithMeas ),
    CasesWithMeasures =  [(Pre=>SubsWithMeas)|RestCasesWithMeas],
    select( OldThm ),
    ctheorem( '$$prove_with_ilem$$' ) =: _,
    cpos( '$$prove_with_ilem$$' ) =: _,
    !,
    cases_with_measures( RestCases, ArgsWithTypes, Measures,
                         RestCasesWithMeas ).

cases_with_measures( [],  _, _, [] ).
```

```
subs_with_measures( [Sub|RestSubs], Measures, P, Hyps,
                    [(Sub-SubMeasures)|RestSubWithMeas] ) :-
    pos(P),
    measures_for_sub(Sub, Measures, Hyps, SubMeasures),
    subs_with_measures( RestSubs, Measures, P, Hyps, RestSubWithMeas ).
subs_with_measures( [], _, _, _, [] ).


measures_for_sub( Sub, Measures, Hyps, SubMeasures ) :-
    D =.. [f|Sub],
    freevarsinterm( D, NeededVars ),
    ebagof( H,T^member(H:T,Hyps), Vars ),
    extend_thin( Vars, Hyps, NeededVars, ToThin ),
    thin_hyps( ToThin, Hyps, ReqdHyps ),
    bagof( Meas,
           ind_lemma_for_sub( Sub, Measures, ReqdHyps,  Meas ),
           SubMeasures ),
    !.

ind_lemma_for_sub( Sub, Meas_s, ReqdHyps, (Measure-ILemma-PreConds) ) :-
    member( Measure, Meas_s ),
    instantiated( Measure, Sub, RecMeasure ),
    Reduction = (RecMeasure <* Measure ),
    provable_with_ilemma( Reduction, ILemma, ILPreConds ),
    union( ILPreConds, ReqdHyps, PreConds ).
```

```
%******************
%*
%*  synth_rec_fn( Lhs, WffRhs, Type, JustifMeasures)
%*  - Synthesise a recursively defined function from a definition.
%*
%* Given:
%*  Lhs the l.h.s. of a recursive function definition
%*  WffRhs a r.h.s for the definition with well-formedness arguments
%* filled in
%*  Type the type for the function, and
%* JustifMeasures a list of justifying measures for the function's
%* recursion scheme
%*
%*   We synthesise the function by:
%*  (*) introducing a well-founded induction based on one of the
%* justifying measures
%*
%*  (*) introducing a case-analysis dual to the function's
%* conditional structure
%*
%*  (*) Proving the cases that result by explicit introduction of the
%* function definition's case-bodies - replacing recursive
%* references with references to the
%* appropriate induction hypothesis.  (See chapter 5 of thesis).
%*
%******************

synth_rec_fn( Lhs, WffRhs, Type, [WfMeasure|_] ) :-

    Lhs =.. [Name|Args],
    noisenl( 10 ),
    noise( 10, 'Synthesis proof for: ' ),
    noisenl( 10, Name ),
    noisenl( 10 ),

    %**** Build the synthesis theorem for the term - this is just its
    %**** Type
    any_overwrite_ok( Name ),
    add_thm( Name, [] ==> Type ),
    select( Name ),

    %**** Introduce the arguments (do not expand out definitions
    %**** as we'll want to match against lemma's about defined
    %**** constructors destructors etc.)
    noise( 100, 'Introducing arguments...' ),
    intro_hyps( [[unfold,_],[expand,_]] ),
    noisenl( 100, 'Done.' ),

    %**** Apply well-founded induction based on a measure justifying
    %**** the functions dual induction schema, and then introduce the
    %**** function's conditional structure by case analysis.
```

```
%**** Once this has bottomed-out introduce the function
%**** definition's case-bodies as witnesses to the proof cases
%**** that result.

( (testmode =: _, apply(because));
  (
    apply(do_wf_induction( Args, WfMeasure, WfMeasure_Hyps )),
    down(1),
    synth_fn_cases( WffRhs, Name, Args, WfMeasure_Hyps, [] )
  )
).
```

```
%*******************
%*
%* synth_fn_cases( Rhs )
%* - Apply the case-analyses, and cutting in off well-formedness
%* arguments required to synthesise a function whose body is Rhs
%*
%*******************

synth_fn_cases( (wffwits WffWits boundin Body),
                Name, Args, Measure, RecRefsSoFar ) :-
    noise( 100, 'Cutting in well-formedness witnesses...' ),

    %**** Cut in the well-formedness witnesses bound by the
    %**** wffwits Witnesses boundin Terms structure.   These will be
    %**** needed to allow the well-formedness of the terms introduced
    %**** later to be proved.

    do( ( apply( cut_in_wff_wits( WffWits) ),
          noisenl( 100, 'Done.' )
        )
        then
          synth_fn_cases( Body, Name, Args, Measure, RecRefsSoFar )
      ).

synth_fn_cases( (if TNm-FNm-Cond then ThenBod else ElseBod),
                Name, Args, Measure, RecRefsSoFar ) :-
    noise( 100, 'Case analysis on condition: ' ),
    noise( 100, Cond ),
    noisenl( 100, '.' ),

    hyp_list( Hyps ),
    new_free_var( 'c', [SH],[TNm:int,FNm:int|Hyps] ),

    %**** Replace any recursive references in the condition to be cut
    %**** in with references to a suitable induction hypothesis.
    ind_hyp_refs( Cond, Name, Args, Measure, IndHypCond,
                  RecRefsSoFar, NRecRefs ),

    %**** Cut in a disjunction on the truth of the condition tested in
    %**** the conditional being synthesised.  Prove the cut in formula
    %**** by application of an appropriate lemma, and then eliminate
    %**** the disjunction to perform the case-analysis.
```

```
    do( apply( seq( IndHypCond\(IndHypCond=>void ), new[SH] ))
        then
          [ apply_lemma( decide_lemma ),
            do( apply( elim( SH, new[TNm,FNm] ) then thin( [SH] ) )
                then

                  %**** Introduce the nested conditional structure in
                  %**** the cases introduced.

                  [ synth_fn_cases( ThenBod, Name, Args, Measure,
                                    NRecRefs ),
                    synth_fn_cases( ElseBod, Name, Args, Measure,
                                    NRecRefs )
                  ]
              )
          ]
      ).

synth_fn_cases( Term, Name, Args, Measure, RecRefsSoFar ) :-

    %**** Once the conditional structure has been introduced prove the
    %**** resulting cases by explicit introduction of the appropriate
    %**** case-body from the function definition.  Replace recursive
    %**** references with references to a suitable induction
    %**** hypothesis.

    noisenl( 100, 'Introducing body of case of function.' ),
    ind_hyp_refs( Term, Name, Args, Measure,
                  IndHypTerm, RecRefsSoFar, _ ),
    apply(intro(explicit(IndHypTerm))),
    down(1),
    prove_mem,
    noisenl( 100, 'Case body Done.' ).
```

C.4 Sample Runs

```
| ?- def_rec_fn(

difference(l,r) =
    (if zerop(r)
     then l
     else if zerop(l)
     then 0
     else difference(p(l),p(r))
     )
).

INTRODUCING NEW (RECURSIVE) FUNCTION: difference

Guessing type of function...Done.

difference == l:pnat=>r:pnat=>pnat

Filling in well-formedness arguments in definition...Done.
Constructing dual induction schema...Done.

Collecting possible justifying measures... Done.
[id(l),id(r)]

Finding measures applicable to induction/recursion...
    (This could take a while!)

Finding measures justifying induction/recursion...
[[id(l)],[id(r)]]

Synthesis proof for: difference

Introducing arguments...Done.

Case analysis on condition: j(zerop(r)).
Case analysis on condition: j(zerop(l)).
Cutting in well-formedness witnesses...Done.
Ind. hyp. for recursive reference: difference(p(l,w),p(r,w0))


INDUCTION TEMPLATE(S):

  [w0:j(posp(r)),w:j(posp(l))]
 => [l-p(l,w),r-p(r,w0)]

  [w0:j(posp(r)),w:j(posp(l))]
 => [l-p(l,w),r-p(r,w0)]
```

```
| ?- def_rec_fn(

minimums(l,o) =
   (if not(consp(l))
    then cons(o,l)
    else if subbagp( car(l), o)
    then minimums( cdr(l), car(l) )
    else if subbagp( o, car(l))
    then minimums( cdr(l), o)
    else if plessp( length( minimums( cdr(l), o )), length(l))
    then minimums(.minimums( cdr(l), o), car(l) )
    else minimums( cdr(l), car(l) )
    )
).

INTRODUCING NEW (RECURSIVE) FUNCTION: minimums

Guessing type of function...Done.

minimums == l:{sexp}=>o:{sexp}=>{sexp}

Filling in well-formedness arguments in definition...Done.
Constructing dual induction schema...Done.

Collecting possible justifying measures... Done.
[length(l),length(o),sexp_cnt(l),sexp_cnt(o)]

Finding measures applicable to induction/recursion...
    (This could take a while!)

Finding measures justifying induction/recursion...
[[length(l)]]
Done.

Synthesis proof for: minimums

Introducing arguments...Done.

Case analysis on condition: j(not consp(l)).
Cutting in well-formedness witnesses...Done.
Case analysis on condition: j(subbagp(car(l,w),o)).
Ind. hyp. for recursive reference: minimums(cdr(l,w),car(l,w))
Case analysis on condition: j(subbagp(o,car(l,w))).
Ind. hyp. for recursive reference: minimums(cdr(l,w),o)
Case analysis on condition:
       j(plessp(length(minimums(cdr(l,w),o)),length(l))).

Ind. hyp. for recursive reference: minimums(cdr(l,w),o)
Ind. hyp. for recursive reference: minimums(cdr(l,w),o) already built.
Ind. hyp. for recursive reference:
    minimums(minimums(cdr(l,w),o),car(l,w))
```

```
Ind. hyp. for recursive reference: minimums(cdr(l,w),car(l,w))


INDUCTION TEMPLATE(S):

  [w:j(plessp(x,y)),w:j(consp(l))]
 => [l-cdr(l,w),o-car(l,w)]
    [l-minimums(cdr(l,w),o),o-car(l,w)]
    [l-cdr(l,w),o-o]
```

Appendix D

Template Instantion, Merging, and Flaw Checking

The following Prolog code implements the template instantation, merging, and final selection phases of our rationally reconstructed recursion analysis procedure. Some sample runs that illustrate the construction of appropriate inductions for a variety of different goals are listed at the end of this appendix. A brief summary of the Prolog environment for which this code has been written can be found in appendix A.

D.1 Template Instantiation

```
%*****
%*
%*  instantiate_templ( +Funct, +PhysicalArgs, +InducibleVars,-Sugg)
%*
%*  Given that:
%*   Funct is the name of a function
%*
%*   PhysicalArgs are the arguments of Funct
%*   InducibleVars are variables that may be induced on
%*
%*  Then:
%*    Sugg is an induction template for Funct instantiated to suit
%* PhysicalArgs.   That is, Sugg is an induction dual to Funct applied
%* to PhysicalArgs.
%*
%* NOTE:  This predicate fails if Funct, applied to PhysicalArgs has no
%* dual induction.  Retrying this predicate will give the rest of
%* the distinct inductions dual to Funct applied to PhysicalArgs.
```

```
%*
%*****

instantiate_templ( Name, PhysicalArgs, InducibleVars, Sugg ) :-
    tuple( template, [Name, FormalArgs, Measured,
              Measure, Schema,Bonus] ),

    % *** Schema substitutes for those variables present as physical
    % *** arguments instantiating the recursive term.
    setof( Var, (member(Var,PhysicalArgs), ttvar(Var)), SubdVars ),

    % *** FUnchanging are the unchanging measured formals
    % *** FChanging are the changing measured formals
    pairing_of( FormalArgs, PhysicalArgs, InstSub ),
    unchanging( Measured, Schema, FUnchanging ),
    subtract( Measured, FUnchanging, FChanging ),

    % *** The unchangeable variables are those present in arguments
    % *** instantiating the unchanging measured formals
    instantiated_list( FUnchanging, InstSub, UnchangeableTerms ),
    T =.. [g|UnchangeableTerms],
    freevarsinterm( T, Unchangeables ),

    %**** The changeables are those terms instantiating
    %* *** the changing measured formals
    instantiated_list( FChanging, InstSub, RawChangeables ),

    %**** Apply the Criteria for checking template instantiation
    %**** described in section 4.5 of thesis.  The actual checking
    %**** that identically instantiated changeables have the same
    %**** substitution is done after instantation, and removal of all
    %**** clashes excluding clashes on changeables.

    \+ (member(C,RawChangeables), \+ member(C,InducibleVars)),
    list_to_set( RawChangeables, Changeables ),
    \+ intersect( Changeables, Unchangeables ),
    instantiable( InstSub, Measured, Instbles ),
    raw_instantiated_templ( Schema, InstSub, Instbles, RawInstTempl ),
    check_schema( RawInstTempl, DualSchema ),


    length( FormalArgs, NoArgs ),
    unchanging( FormalArgs, Schema, UnchangedArgs ),
    union( Instbles, UnchangedArgs, ArgsDealtWith ),
    length( ArgsDealtWith, NoDealtWith ),
    Score is integer(10000*NoDealtWith / NoArgs )+Bonus,

    %**** The instantiated schema deals with precisely one recursive
    %**** term, and is justified by the instantiated measure.
    %**** It has not so far been shown to be flawed.

    instantiated_list( Measure, InstSub, InstMeas ),
```

```
unchanging( SubdVars, DualSchema, UnchangingVars ),
Sugg = schema( DualSchema, Score, unflawed,  [InstMeas],
               vars( SubdVars, Changeables, UnchangingVars )  ).
```

```
%*********
%*
%*   check_schema( +RawSchema, -Schema )
%*
%* Given that:
%*   RawSchema is an instantiated induction template and there are
%* no clashes in the substitutions of RawSchema
%*
%* Then:
%*   Schema is RawSchema with duplicate inductions removed.
%*
%*   NOTE:  If there are clashes this predicate fails.
%*********

check_schema( [(Pre=>Subs)|RestCases], [(Pre=>TSubs)|RestTCases ] ) :-
    check_subs( Subs, TSubs ),
    check_schema( RestCases, RestTCases ).
check_schema( [],[] ).

check_subs( [Sub|RestSubs], [TSub|RestTSubs] ) :-
    check_sub( Sub, TSub ),
    check_subs( RestSubs, RestTSubs ).
check_subs( [], [] ).

check_sub( [(Var-Tm1)|RestSub], RestTSub ) :-
    member( (Var-Tm2), RestSub ),
    !,
    Tm1 = Tm2,
    check_sub( RestSub, RestTSub ).
check_sub( [(Var-Tm1)|RestSub], [(Var-Tm1)|RestTSub] )  :-
    check_sub( RestSub, RestTSub ).
check_sub( [], [] ).
```

D.2 Merging

```
%******
%*
%*     merged_form( +S1, +S2, +Bound, +CommonVars, -Merged)
%*
%*    Given that:
%* S1 and S2 are induction schemata  with a common subsuming schema
%*
%* Bound is the union of the variables substituted for by S1,S2
%*
%* CommonVars are the variables substituted for in common by S1 S2
%*
%*    Then:
%* M is the simplest common subsuming schema for S1 and S2
%*
%* NOTE: fails if no common subsuming schema can be found.
%*
%* (see chapter 4 of thesis for explanation of algorithm)
%******

merged_form( Form1, Form2, Bound, CommonVars, MdForm ) :-
    raw_merged_form( Form1, Form2, Bound, CommonVars, MdForm ).

merged_form( Form1, Form2, Bound, CommonVars, MergedForm ) :-
    extension(Form2, Form1, CommonVars ),
    extendable( Form2, CommonVars ),
    !,
    merged_iter_form2( 6, Form1, Form2, Form1, Form2, Bound,
                       CommonVars, MergedForm ).

merged_form( Form1, Form2, Bound, CommonVars, MergedForm ) :-
    extension(Form1, Form2, CommonVars ),
    extendable( Form1, CommonVars ),
    !,
    merged_iter_form2( 6, Form2, Form1, Form2, Form1, Bound,
                       CommonVars, MergedForm ).

                        Bound, CommonVars, MergedForm ).
```

```
merged_iter_form2( N, Form1, Form2, OrgForm1, OrgForm2,
                   Bound, CommonVars, MergedForm ) :-
    extension(Form2, Form1, CommonVars ),
    !,
    sequence( Form2, OrgForm2, NextForm2 ),
    merged_iter_form1( N, Form1, NextForm2,  OrgForm1, OrgForm2,
                       Bound, CommonVars, MergedForm ).

merged_iter_form2( N, Form1, Form2, OrgForm1, OrgForm2,
                   Bound, CommonVars, MergedForm ) :-
    extension(Form1, Form2, CommonVars ),
    !,
    sequence( Form1, OrgForm1, NextForm1 ),
    merged_iter_form1( N, Form2, NextForm1, OrgForm2, OrgForm1,
                       Bound, CommonVars, MergedForm ).


merged_iter_form1( _, Form1, Form2, _, _,
                   Bound, CommonVars, MergedForm ) :-
    raw_merged_form( Form1, Form2, Bound, CommonVars, MergedForm ).

merged_iter_form1( N, Form1, Form2, OrgForm1, OrgForm2,
                   Bound, CommonVars, MergedForm ) :-
    N > 0,
    SN is N-1,
    merged_iter_form2( SN, Form1, Form2, OrgForm1, OrgForm2,
```

```
%******
%*
%*    raw_merged_form( +S1, +S2, +Bound, +CommonVars, -Merged)
%*
%*  - The Boyer-Moore merging algorithm, enhanced to deal with
%* complication that in NuPRL type-theory pre-conditions can bind
%* witnesses that can appear in the terms substituted in
%* step-substitutions.
%*
%******

raw_merged_form( S1, S2, Bound, CommonVars, Merged ) :-
    length(S1,L1),
    length(S2,L2),
    \+ L1 > L2,
    !,
    new_witness_names( S1, wa, Bound, NS1 ),
    new_witness_names( S2, wb, Bound, NS2 ),
    merged_cases( NS1, NS2, notmerged, Bound, CommonVars, Merged ).

raw_merged_form( S1, S2, Bound, CommonVars, Merged ) :-
    new_witness_names( S1, wa, Bound, NS1 ),
    new_witness_names( S2, wb, Bound, NS2 ),
    !,
    merged_cases( NS2, NS1, notmerged, Bound, CommonVars, Merged ).

merged_cases( [C1|R1], S2, _, Bound, CommonVars, [M|RestMerged] ) :-
    member( C2, S2 ),
    merged_case( C1, C2, Bound, CommonVars, M ),
    delete( S2, C2, R2 ),
    merged_cases( [C1|R1], R2, merged, CommonVars, RestMerged ).

merged_cases( [_|R1], S2, merged, CommonVars, RestMerged ) :-
    merged_cases( R1, S2, notmerged, CommonVars, RestMerged ).
merged_cases( [], S2, _, _, S2 ).


merged_case( (Pre1=>Subs1), (Pre2=>Subs2),
             Bound, CommonVars, (UnionPre=>MergedSubs) ) :-
    length(Subs1, L1),
    length(Subs2, L2),
    \+ L1 > L2,
    !,
    precond_union( Pre1, Pre2, Bound, WitMatch, UnionPre ),
    merged_subs( Subs1, Subs2, notmerged, CommonVars,
                 WitMatch, MergedSubs ).
merged_case( (Pre1=>Subs1), (Pre2=>Subs2),
             Bound, CommonVars, (UnionPre=>MergedSubs) ) :-
    !,
    precond_union( Pre2, Pre1, Bound, WitMatch, UnionPre ),
    merged_subs( Subs2, Subs1, notmerged,
                 CommonVars, WitMatch, MergedSubs ).
```

```
merged_subs( [C1|R1], S2, _, CommonVars, WitMatch, [M|RestMerged] ) :-
    member( C2, S2 ),
    merged_sub( C1, C2, CommonVars, WitMatch, M ),
    delete( S2, C2, R2 ),
    merged_subs( [C1|R1], R2, merged, CommonVars,
                 WitMatch, RestMerged ).

merged_subs( [_|R1], S2, merged, CommonVars, WitMatch, RestMerged ) :-
    merged_subs( R1, S2, notmerged, CommonVars, WitMatch, RestMerged ).

merged_subs( [], S2, _, _, _, S2 ).

merged_sub( Sub1, Sub2, CommonVars, WitMatch, MergedSub ) :-
    instantiated_list( Sub1, WitMatch, ISub1 ),
    \+ (
      member( Var, CommonVars ),
      member( (Var-Repl1), ISub1 ),
      member( (Var-Repl2), Sub2 ),
      \+ Repl1 = Repl2
    ),
    union( ISub1, Sub2, MergedSub ).
```

D.3 Flaw Checking

```
%**********************
%*
%*    flawed( +S1, +S2, +OF1, +OF2, -F1, -F2 )
%*
%*    - F1, F2 are the flawing state flawed|unflawed of
%*    schemas S1 and S2 whose flawing states so far are OF1 OF2
%*
%**********************

flawed( _, _, flawed, flawed, flawed, flawed ).

flawed( S1, S2, F1, F2, F1, F2 ) :-
    S1 = schema( _, _, _, _, vars( Subd1, _IndOn1, _Unch1 ) ),
    S2 = schema( _, _, _, _, vars( Subd2, _IndOn2, _Unch2 ) ),
    intersection( Subd1, Subd2, [] ),
    !,
    fail.

flawed( S1, S2, flawed, _, flawed, F2 ) :-
    ((induction_clash( S2, S1 ),F2=flawed);F2=unflawed),
    !.

flawed( S1, S2, _, flawed, F1, flawed ) :-
    ((induction_clash( S1, S2 ),F1=flawed);F1=unflawed),
    !.
flawed( S1, S2, _, _, F1, F2 ) :-
    ((induction_clash( S2, S1 ),F2=flawed);F2=unflawed),
    ((induction_clash( S1, S2 ),F1=flawed);F1=unflawed),
    !.

%*************************
%*
%*    induction_clash( S1, S2 )
%*    -  The substitutions of schema's S1 S2 disagree somewhere
%*    on a measured variable of S1.  I.e. Their forms cannot be merged
%*    on the measured variables of S1 shared with S2.
%*
%*
%*************************

induction_clash( S1, S2 ) :-
    S1 = schema( Form1, _, _, _, vars( Subd1, IndOn1, _Unch1 ) ),
    S2 = schema( Form2, _, _, _, vars( Subd2, _IndOn2, _Unch2 ) ),
    intersection( IndOn1, Subd2, RelevantSubdVars ),
    union( Subd1, Subd2, Subd ),
    pairing_of(Subd,Subd,Binding),
    \+ RelevantSubdVars = [],
    \+ raw_merged_form( Form1, Form2, Binding, RelevantSubdVars, _ ).
```

D.4 Sample Runs

```
| ?- inductions(
    equal(append(a,append(b,c)),append(append(a,b),c))
).

Goal suggests 3 induction schemata.

  [w:j(consp(a))]
 => [a-cdr(a,w)]

  [w:j(consp(b))]
 => [b-cdr(b,w),c-c]

  [w:j(consp(a))]
 => [a-cdr(a,w),b-b]

These merge into 2 distinct induction schemata.

unflawed: 2.0000
  [wb:j(consp(a))]
 => [a-cdr(a,wb),b-b]

flawed: 1.0000
  [w:j(consp(b))]
 => [b-cdr(b,w),c-c]
```

```
| ?- inductions(
    x:pnat=>y:pnat=>z:pnat=>(j(lessp(x,y)) #
    j(lessp(y,z)))=>j(lessp(x,z))
).

Goal suggests 6 induction schemata.

  [w0:j(posp(y)),w:j(posp(x))]
 => [x-p(x,w),y-p(y,w0)]

  [w0:j(posp(y)),w:j(posp(x))]
 => [x-p(x,w),y-p(y,w0)]

  [w0:j(posp(z)),w:j(posp(y))]
 => [y-p(y,w),z-p(z,w0)]

  [w0:j(posp(z)),w:j(posp(y))]
 => [y-p(y,w),z-p(z,w0)]

  [w0:j(posp(z)),w:j(posp(x))]
 => [x-p(x,w),z-p(z,w0)]

  [w0:j(posp(z)),w:j(posp(x))]
 => [x-p(x,w),z-p(z,w0)]

These merge into 1 distinct induction schemata.

unflawed: 60000
  [wa:j(posp(y)),wb:j(posp(z)),wb0:j(posp(x))]
 => [y-p(y,wa),x-p(x,wb0),z-p(z,wb)]
```

```
| ?- inductions(
    equal(reverse_acc(l,a),append(reverse_unf(l),a))
).

Goal suggests 2 induction schemata.

  [cc:j(consp(l))]
 => [l-cdr(l,cc),a-cons(car(l,cc),a)]

  [w:j(consp(l)),w0:j(consp(cdr(l,w)))]
 => [l-cdr(cdr(l,w),w0)]

These merge into 1 distinct induction schemata.

unflawed: 2.0000
  [wb:j(consp(l)),wb0:j(consp(cdr(l,wb)))]
 => [l-cdr(cdr(l,wb),wb0),
     a-cons(car(cdr(l,wb),wb0),cons(car(l,wb),a))]
```

```
| ?- inductions(
    (j(lessp(y,x))#(j(prime1(x,y))=>void)#
     (j(zerop(x))=>void)#(j(peq(x,s(0)))=>void)#
     (j(zerop(y))=>void))=>j(peq(remainder(x, gf(x,y)),0))
    ).

Goal suggests 5 induction schemata.

  [w0:j(posp(x)),w:j(posp(y))]
 => [y-p(y,w),x-p(x,w0)]

  [w0:j(posp(x)),w:j(posp(y))]
 => [y-p(y,w),x-p(x,w0)]

  [w:j(posp(y))]
 => [x-x,y-p(y,w)]

  [w:gf(x,y)<*x]
 => [x-difference(x,gf(x,y))]

  [w:j(posp(y))]
 => [x-x,y-p(y,w)]

These merge into 3 distinct induction schemata.

flawed: 2.0000

  [wb:j(posp(x)),wb0:j(posp(y))]
 => [y-p(y,wb0),x-p(x,wb)]

unflawed: 2.0000
  [wb:j(posp(y))]
 => [x-x,y-p(y,wb)]

flawed: 1.0010
  [w:gf(x,y)<*x]
 => [x-difference(x,gf(x,y))]
```

```
| ?- inductions(
   h:{sexp}=>
   t:{sexp}=>
   w:( y:{sexp}=>x:{sexp}=>
       (  j(member(x,cons(h,t))) #
          j(member(y,cons(h,t)))
       ) =>
       j(subbagp(x,y))#j(subbagp(y,x))=>void
     ) =>
   length(minimums(t,h))=s(0) in pnat

).

Goal suggests 1 induction schemata.

  [w:j(plessp(x,y)),w:j(consp(t))]
 => [t-cdr(t,w),h-car(t,w)]
    [t-minimums(cdr(t,w),h),h-car(t,w)]
    [t-cdr(t,w),h-h]

These merge into 1 distinct induction schemata.

unflawed: 1.0010

  [w:j(plessp(x,y)),w:j(consp(t))]
 => [t-cdr(t,w),h-car(t,w)]
    [t-minimums(cdr(t,w),h),h-car(t,w)]
    [t-cdr(t,w),h-h]
```

Appendix E

Finding Freeable Variables in NuPRL Type-Theory

NuPRL Type-theory is implemented as a sequent calculus, and has 5 main connectives. The logic is intuitionistic and thus has no normal form for goals.

NuPRL Goal Syntax

NuPRLGoal ::= *var* : *type*, ... ⊢ *type*

type ::= *var* : *type*->*type* "universal quantification"

type->*type* "implication"

var : *type*#*type* "existential quantification"

type#*type* "conjunction"

type | *type* "disjunction"

NuPRL type-theory sequents can be transformed into equivalent sequents in 3 distinct ways:

Transformations Eliminating Quantifiers

$\ldots, x : A \# B \vdash \ldots \rightarrow \ldots, x : A, B \vdash \ldots$ (elim1)

$\ldots \vdash x : A \text{->} B \rightarrow \ldots, v_{new} : A \vdash B$ (intro1)

Transformations moving # 's left in a hypothesis

$\ldots, A \# B \vdash \ldots \rightarrow \ldots, A, B \vdash \ldots$ (elim2)

$(x : A \# C) | (y : A \# C) \vdash \rightarrow\ x : A \# (y : A \# (B|C)) \vdash$ (leftrule1)

Transformations moving ->'s left in the conclusion

$\ldots \vdash A \text{-> } B \rightarrow \ldots, A \vdash B$ (intro2)

$\vdash (x : A \text{-> } B) \# (y : A \text{-> } C) \rightarrow \vdash x : A \text{-> } y : A \text{-> } (B \# C)$ (leftrule2)

Rules for Finding Freeable Variables

On the basis of these transformations we obtain the following rules for exhaustively rewriting a NuPRL type-theory goal to find the set of freeable variables:

Goal	**Transformation**	**Freed Variable**
$\ldots \vdash x : A \text{-> } B$	intro1	x
$\ldots \vdash A \text{-> } B$	intro2	
$\ldots \vdash (x : A \text{-> } B) \mid (y : A \text{-> } C)$	leftrule2	
$\ldots, x : A \# B, \ldots \vdash \ldots$	elim1	x
$\ldots, A \# B, \ldots \vdash \ldots$	elim2	
$\ldots, (x : A \# B) \mid (y : A \#) \ldots \vdash \ldots$	leftrule1	

The exhaustive application of these rules is guaranteed to terminate as every transformation eliminates a connective or leaves the number of connectives unchanged and moves a # or -> left.

Appendix F

Full Procedure for Finding Required Sets of Hypotheses

F.1 An Extended Notion of Idleness

The procedure for finding required sets of hypotheses listed in section 4.3.1 is in fact somewhat simplified. The full algorithm (listed below) is based on the following, extended, notion of idle proof steps:

A proof sub-goal $H \vdash C$ with proof P is a *throwback* if it has an ancestor (including itself) $H' \vdash C'$, and a descendant (including itself) $H'' \vdash C'$ with a P-required set r such that:

- No proof step between $H \vdash C$ and $H'' \vdash C'$ that is required to introduce a member of r depends on the goal conclusion.
- $(r \cup r') \subseteq H'$ where r' is the set of hypotheses required in proof steps between $H \vdash C$ and $H'' \vdash C'$ needed to introduce the members of r not in H.
- Either $H'' \vdash C'$ or $H' \vdash C'$ is not identical to $H \vdash C$.

We term the descendant $H'' \vdash C'$ a *throwback basis* since it provides the basis for a throwback proper.

A proof step is *idle* if it is a descendant of $H' \vdash C$ that is not in the proof for $H'' \vdash C'$ and is not required to introduce members of r not in H.

The intuition behind this extended notion of idleness is that we can prove $H' \vdash C'$ by applying the rules required to introduce members of r not in H followed by the proof of $H'' \vdash C'$ corresponding to the P-required set r.

F.2 An Extended Algorithm for Finding Required Sets

PROCEDURE required(P, A)

$H \vdash C$:= root node of P
req'd_in_rule := Hypotheses required by rule applied to $H \vdash C$
new_hyps := New Hypotheses in root nodes of children of $H \vdash C$
n := Number of children of $H \vdash C$

IF $n = 0$
THEN
  req'd_sets := *req'd_rule*
  throwb's := $\emptyset$
  bases := $\emptyset$
ELSE
  FOREACH $i \in \{1, \ldots, n\}$
  DO $throwb's_i, req'd_i, bases_i$:= required$(child_i, cons(H \vdash C, ancestors))$

  $req'd\_in\_kids := \{nr_1 \cup \ldots \cup nr_n \mid nr_1 \in req'd_1, \ldots, nr_n \in req'd_n\}$
  $req'd\_here := \{(rb \cup req'd\_rule) \setminus new\_hyps \mid rb \in req'd\_in\_kids\}$

  $throwb's := \bigcup_i throwb's_i$
  $req'd\_in\_throwb's := \{r \in R \mid (C, R) \in throwb's\}$

  FOREACH $bases_i$
  DO
    IF rule applied to $H \vdash C$ depends on C
    THEN
      $bases_i := \{(C', r) \in bases_i \mid r \cap new\_hyps = \emptyset\}$
    ELSE
      $req'd\_rest := \{(\bigcup_{j \neq i} r_j \cup req'd\_rule) \mid r_i \in req'd_i\}$
      $bases_i := \{(C', (r \cup req'd\_in\_rule) \setminus new\_hyps) \mid$
                $(C', r) \in bases_i \wedge r \cap new\_hyps \neq \emptyset\} \cup$
              $\{(C', r) \in bases_i \mid r \cap new\_hyps = \emptyset\}$
  END

$bases := \bigcup_i bases_i$
$req'd\_in\_bases := \{r \mid (C, r) \in bases\}$

$req'd\_sets := (req'd\_here \cup req'd\_in\_throwb's \cup req'd\_in\_bases)$
$req'd\_sets := \{r \in \mid \neg\exists\, r' \in req'd\_sets\, .r' \subset r\}$
ENDIF
IF $\exists\, (H' \vdash C) \in A$
THEN
$bases := \{(C, r) \mid r \in req'd\_sets)\} \cup bases$

$$new\_throwb's := \{(C', R') \mid (H' \vdash C') \in A \wedge R' \neq \emptyset \wedge R' = \{r \mid (C', r) \in bases \wedge r \subseteq H'\}\}$$

throwb'_set := $throwb's \cup new\_throwb's$
$bases := \{(C', r) \in bases \mid (H' \vdash C') \in A\}$

RETURN $(throwb'\_pairs, req'd\_sets, bases)$

This algorithm, like that listed in section 4.3.1, constructs the P-required sets for the sub-goals of a proof on the way up from a depth-first traversal of the proof-tree. For each node of the proof-tree:

1. It works out *req'd_here* – the P-required sets for the current goal that correspond to proofs that make use of the existing rule applied to the goal. These are worked out from the P-required sets of the goal's children, which are passed up from the recursive calls for these children.

2. It works out *throwb_sets* – the P-required sets for the current goal that correspond to proofs that do not make use of the existing rule applied to the goal. That is, the proofs based on the proofs for throwbacks to the current goal. These P-required sets are constructed from the information on throwbacks passed up from the recursive calls on the goal's children.

3. It works out *bases* – the throwback bases. That is, descendants of the current goal whose conclusion matches that of an ancestor; but for which no ancestor has yet been found whose required hypotheses are a subset of the hypotheses present in the ancestors whose conclusions they match.

These are constructed from the information on throwback bases passed up from the recursive calls on the goal's children.

4. It constructs *req'd_in_bases* – the P-required sets for proofs of the current goal that omit proof steps not needed to derive hypotheses required in a throwback basis.

5. It combines the sets *req'd_here*, *throwb_sets*, *req'd_in_bases* to give *req'd_sets* – the P-required sets for the current goal.

6. It adds the throwback bases due to the current goal to *bases*.

7. It adds the throwbacks from the current goal that can be based on *bases* to *throwb_sets*.

8. It returns the throwback information, *throwb_sets*, the P-required sets, and the throwback bases *bases* for the current goal to its caller.

Appendix G

Results

G.1 Recursive Functions

In addition to then recursive functions defined in [BM79] our rationally reconstructed algorithm accepts, and automatically constructs dual inductions for, several classes of function definition where the Boyer and Moore's published procedure fails. The example definitions below typify these classes of definition. Definitions successfully accepted only by our rationally reconstructed algorithm are marked †. Those also accepted by recent versions of the Boyer-Moore theorem prover because of undocumented modifications are listed marked *.

† `minimums(l,o) =`

```
      if not(consp(l))
      then cons(o,l)
      else if subbagp( car(l), o)
      then minimums( cdr(l), car(l))
      else if subbagp( o, car(l))
      then minimums( cdr(l), o)
      else if plessp( length( minimums( cdr(l), o)), length(l))
      then minimums( minimums( cdr(l), o), car(l) )
      else minimums( cdr(l), car(l) )
```

The Boyer-Moore procedure is unable to prove function definitions well-founded if their well-foundedness depends on conditionals that embed recursive references to the function being defined.

* `unexpected(l,e) =`

```
      if not consp(l)
      then {nil}
      else if equal(car(l),number(nat(32)))
      then unexpected(cdr(l),e)
      else if not consp(e)
      then cons(car(l), unexpected(cdr(l),e))
      else cons(car(l), unexpected(cdr(l),cdr(e)))
```

The Boyer-Moore procedure is unable to construct dual inductions for function definitions whose well-formedness depends on conditionals nested within conditionals one whose branches has a subset of the recursive references of the other.

```
* lineparse(l,q) =
      if q
      then (if not consp(l)
            then cons( number(nat(32)), {nil} )
            else if equal(car(l),number(nat(34)))
            then lineparse(cdr(l),{true})
            else cons(car(l), lineparse(cdr(l),{false})))
      else if not consp(l)
           then {nil}
           else if equal(car(l),number(nat(92)))
           then lineparse(cdr(l),{true})
           else cons(toupper(car(l)), lineparse(cdr(l),{false}))
```

The Boyer-Moore procedure is unable to construct dual inductions for function definitions that have conditionals relevant to the function's recursion scheme nested within conditionals that are irrelevant to the recursion scheme.

G.2 Explicit Quantifiers

An important advantage of our recursion analysis procedure over the Boyer-Moore original is that it can deal with goals containing explicitly quantified variables. Our algorithm, for example, finds appropriate inductions for goals such as

```
x:pnat=>
y:{sexp}=>
w:sorted(y)=>
z:{sexp}#(sorted(z)#equal(y,delete(number(x),z)))
```

```
or h:{sexp}=>
   t:{sexp}=>
   ( y:{sexp}=>
     x:{sexp}=>
     (j(member(x,cons(h,t)))#j(member(y,cons(h,t))))=>
     ( j(subbagp(x,y))#
       j(subbagp(y,x))=>void
     )
   )=>
   length(minimums(t,h))=s(0) in pnat
```

which the Boyer-Moore algorithm is quite unable to cope with.

G.3 Appropriate Inductions

In addition to finding appropriate inductions for all the goals listed in the appendix of [BM79], our version of recursion analysis finds appropriate inductions for several types of goal for which Boyer and Moore's published procedure will fail. Typical examples of such goals include:

- `equal(reverse_acc(l,a),append(reverse_unf(l),a))`
- ```
  (even(x) # lessp(x, y) # lessp(y, z) # not(even(z))) =>
  lessp(plus(x,2), z)
  ```
- ```
  ( j(lessp(y,x)) #
    j(prime1(x,y))=>void #
    j(zerop(x))=>void #
    j(peq(x,s(0)))=>void #
    j(zerop(y))=>void
  )=>
  j(peq(remainder(x, gf(x,y)),0))

  *
  ```
- ```
  (unique(b))=>
  perm(append(a,append(b,c)), append(append(a,reverse(b)),c)))
  ```

Recent implementations of the Boyer-Moore theorem prover can also successfully prove the theorem marked * due to an undocumented modification. We have forced the Boyer-Moore theorem prover to accept these induction suggestions rather than its own. In all these cases it succeeds with our suggestion and fails on its own.
  ```

This thesis has been composed by myself
and the work is my own.