A Theorem Proving Based
Methodology for Software Verification

Mark Aagaard
Miriam Leeser

TR 93-1335
March 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

Contents

A Theorem Proving Based Methodology for Software Verification

Mark Aagaard Miriam Leeser

March 17, 1993

Abstract

We have developed an effective methodology for using a proof development system to prove properties about functional programs. This methodology includes techniques such as hiding implementation details and using higher order theorems to structure proofs and aid in abstract reasoning.

The methodology was discovered and refined while verifying a logic synthesis tool with the Nuprl proof development system. The logic synthesis tool, *Pbs*, implements the weak division algorithm. *Pbs* consists of approximately 1000 lines of code implemented in a functional subset of Standard ML. It is a proven and usable implementation of a hardware synthesis tool. The program was verified by embedding the subset of SML in Nuprl and then verifying the correctness of the implementation of *Pbs* in the Nuprl logic.

1 Introduction

This paper describes a methodology for using a proof development system to prove properties about functional programs. The methodology was discovered while verifying a logic synthesis tool. The tool, *Pbs*: Proven Boolean Simplification, is based on Brayton and McMullen's weak division algorithm [BM82]. *Pbs* represents a case study in verifying large, functional programs: the implementation of *Pbs* contains approximately 1000 lines of Standard ML code and the verification required over 500 theorems.

The complete formalization of *Pbs* consists of a description of the properties to be proved, a semantics for the implementation language (a functional subset of SML), and a mechanized proof showing that the implementation satisfies the properties claimed by the weak division algorithm. The proof was done in the Nuprl proof development system [C$^+$86], and involved embedding a subset of SML in Nuprl, verifying the implementation of *Pbs* in Nuprl using that subset of SML and showing the equivalence of the semantics of this subset of SML in Nuprl and in the SML Definition [RH90].

We developed our methodology out of necessity as the verification of *Pbs* progressed. It quickly became obvious that we needed a uniform way to hide implementation details and prevent duplicated reasoning. Through trial and error we developed techniques for verifying properties of functions which met these goals, and from these techniques an overall methodology evolved. We have continued to use the methodology in several areas, including a set of functions for describing the structure of digital hardware circuits and a package for manipulating matrices. It is quite clear that for large verification efforts to be successful, it is vital that they employ methodologies which aid in organizing and structuring the proof process.

Because powerful proof systems are a relatively recent development, there is not a lot of collected knowledge about how to use these systems effectively. Knowledge about what techniques to use in different circumstances and the best overall approach to choose in tackling a problem come almost exclusively from personal experience with using the tools and with advice from mentors who are already experts at using the system. One of the goals of this paper is to share our experiences with using Nuprl. Our methodology is relevant to almost anyone using a proof system to verify software, but it will be most useful to those users whose systems are based on a higher order logic. This is because a significant portion of the methodology exploits higher order logic to structure proofs and capture abstract reasoning about functions.

While we agree that it is not yet feasible to rely solely on theorem proving based methods to verify software, the development of an efficient methodology will allow theorem proving based methods to tackle more substantial projects. Without a guiding methodology, verification engineers are forced to decide both what proof strategy they should use at the abstract level (i.e. why is the theorem true) and what steps are needed to implement their strategy in the proof system. By following a methodology, the low level details of interacting with the proof system are consistent, uniform and known ahead of time. Thus the user is free to concentrate on more abstract reasoning.

A proof effort of the magnitude of *Pbs* requires a significant amount of time. Nevertheless, there are many cases when the time invested in verification will be well spent. When a program will be used over and over again, it is often more effective to invest effort in verifying the software, rather than verifying the output of that software each time the program is used. Proving properties about a program forces an improved understanding of the program, which increases the opportunity to perform safe optimizations. Theorem proving based verification is well suited for cases where there is a large abstraction gap between the level of the specification and implementation. Higher order logic allows for natural and expressive specifications and proof systems provide mechanisms for automatically maintaining the consistency of large proofs. Theorem proving techniques work best for programs written in a functional style. This is because it is much easier to reason about functional code than code which relies on imperative features such as side effects and pointers.

In our case, *Pbs* is an integral part of the Bedroc hardware synthesis system [LCA+93], thus it will be used over and over again. Because we verified *Pbs*, we were able to make several optimizations to the implementation. Had we not verified *Pbs*, we would have been reluctant to make the optimizations, for fear of introducing bugs into the code. The difference between the level of abstraction of the implementation of *Pbs* and the specification was quite large. The specification reasons about abstract properties of Boolean logic and properties of a division operation, while the implementation is based on simple operations on lists. Although the weak division algorithm is complicated, it lends itself to a well structured implementation: all of the operations can be efficiently written in a functional style, most of the functions are based upon simple operations on lists, and the control structures of almost all of the functions are very simple. These factors made it desirable and feasible to formally prove properties about an implementation of the weak division algorithm.

Because of the size and complexity of the weak division algorithm, it would have been infeasible to try to use paper proofs to prove properties about *Pbs*. For large verification efforts a proof system is an absolute necessity. Proof systems help deal with the complexity encountered in large verifcation efforts because they ensure that definitions are used consistently, lemmas are completely proved before they are used, and only sound inference rules are used. In addition, they provide a sound basis for describing objects and proving theorems and automate some of the mundane aspects of proofs.

Some estimates on the size of the verification effort are: there is a four to one ratio in the size of the specification (theorems) to the size of the implementation. The total size of our verification effort is over 400 pages; of these 400 pages, user input represents only eighty pages (the rest is generated by Nuprl to make the proofs readable). It is interesting to note that a team which used a non-theorem proving based methodology (Cleanroom [DK90]) to specify and implement a large program found the same four to one ratio in the size of the specification to implementation. The team emphasized that they never wrote a specification which was more formal than needed to adequately describe the desired behavior of the code. The fact that the ratio of specification to implementation is the same for both *Pbs* and the Cleanroom project illustrates that mechanized proof system may be used to efficiently describe the specifications of programs.

Our development of the proofs proceeded almost entirely within the Nuprl system. Many proofs were done entirely within Nuprl, without any need to sketch out arguments on paper before using Nuprl or any need to print out a copy of the proof while working on it. The fact that we were able to do so much of the work within Nuprl demonstrates that there are now proof development systems available which are sophisticated enough to be truly considered environments for creating proofs, as opposed to tools simply used to check proofs once they have been developed.

1.1 Related Work

The developement and verification of *Pbs* was motivated in part by past research in using mechanical proof systems for verifying digital hardware. Originally, most work in this area was done by proving the correctness of an implementation after it was designed [Coh88, Hun86]. This methodology suffers from the fact that such a *post hoc* verification process is invariably time-consuming and labor intensive. Many researchers are proving hardware design tools correct and investigating synthesis by proven transformations. For example, Martin [Mar90] uses proved correct transformations to synthesize delay insensitive circuits, Chin [Chi90] uses verified design procedures to synthesize array multipliers, and McFarland [McF91] found several errors in the System Architect's Workbench [TDW+88] while proving their transformations correct. *Pbs* is the only work that has used theorem proving based methods to verify the implementation of a synthesis tool.

Wing has written a survey article on software verification which provides an excellent introduction to the area as well as a summary of important advances in the area [Win90]. The Boyer-Moore theorem prover [BM88] has been used to verify a number of hardware and software projects, most notably the verified stack consisting of a verified microprocessor, assembly language compiler, high level language compiler, and operating system kernel [BM889]. The Process Verification Environment has been used by Fischer to check a railroad type protocol [FST92]. Both in work done using the Boyer-Moore theorem prover and using PVE, errors were found which would only occur after a sequence of events occurred in a specific order. Both groups believe that it would have been virtually impossible to find and correct the errors without the use of such tools.

Lafontaine [LLS90] has described the use of the B theorem prover and the Vienna Development Method in the design and implemenation of a software program for controlling a simple robot. One of the goals of this case study was to explore the potential for reusing results of the project. Their efforts in improving reusability concentrated on high quality, semi-automated documentation (in the style of literate programming) and structuring the verification effort so as to group common lemmas and theorems. Among the conclusions reached by the study were that support environments and tools such as theorem provers are necessary when formally developing realistic software systems and that theorem provers are useful for verifying programs several hundred lines in length.

PVE and the Boyer-Moore and B theorem provers differ from Nuprl in that Nuprl is an interactive proof development system using a higher order logic. PVE is based on process algebras and model checking and both Boyer-Moore and B are first order systems. Boyer-Moore is an automated theorem prover that uses a heuristic to direct lemma application. B may be used either interactively or automatically. In its automated mode B uses a pattern matching mechanism to select and apply previously proven lemmas.

At the other end of the spectrum, much attention has been payed to methodologies which attempt to increase the productivity of programmers while reducing the amount of errors in code. Proponents of these methodologies rarely include theorem proving based verification as one of their techniques, because it is perceived as not being practical for significant size programs.

Mills' Cleanroom [LM88] is a methodology for software development and management motivated by empirical statistics of defects in programs. Cleanroom has been used on a number of large software projects, including COBOL/SF^2, a 50,000 line upgrade to a program which automatically transforms unstructured COBOL code into structured code. This case study demonstrated the need to use uniform methodologies to organize and control the software development cycle. Dyer [DK90] has analyzed the use of the Cleanroom methodology in comparison to the conventional structural software testing process. He has concluded that the use of approaches which are directed at *designing software which meet specifications*, rather than *implementing programs which produce correct test results*, are more efficient and will result in higher quality software.

1.2 Overview

Section 2 contains a description of how we embedded SML in Nuprl, a brief outline of the weak division algorithm and an overview of our implementation and verification of *Pbs*. Section 3 is a detailed description of our methodology and Section 4 illustrates our methodology with the implementation and verification of three functions in *Pbs*. Each of these examples illustrates a specific technique in the methodology and builds upon the previous examples. Section 5 analyzes the results of the verification of *Pbs*. More detailed descriptions of the algorithms used in *Pbs* can be found elsewhere [AL91, Aag92].

2 Background

Pbs implements the weak division algorithm, which is a global approach to Boolean simplification. The weak division algorithm was first described in a paper by Brayton and McMullen [BM82]. It is currently used in several CAD tools, including *Mis*, which is part of the Berkeley Synthesis System [BR⁺87]. We based our implementation and verification of *Pbs* on the informal definitions, algorithms, and proof outlines presented in these articles.

Our goals were for *Pbs* to be a proven and usable implementation of the weak division algorithm. In order to meet these two goals, we decided to implement *Pbs* in a functional subset of the Standard ML (SML) programming language and to embed this subset of SML in a mechanical proof system. We chose SML because it is a very high level language, is primarily functional, and has well defined semantics. Embedding SML in a proof system allowed the code for *Pbs* to be reasoned about in the proof system and compiled and run using an SML compiler. Thus there is a very high degree of confidence that the SML implementation has the same behavior as the implementation in the proof system.

For the proof system we chose to use Nuprl, which is based on Martin-Löf's constructive type theory and is a member of the LCF [GMW79] family of proof systems. With the LCF family of systems, the user begins by entering a theorem to be proved. The theorem represents the *goal* of a proof. The user applies *tactics* which manipulate the goal, usually by breaking it down into a set of sub-goals. This creates a structure known as a proof tree. In order to successfully complete a proof, the sub-goals should become increasingly simple. Eventually an individual sub-goal will be simple enough that the system can automatically prove it with built in rules. When all of the leaves of the proof tree have been shown to be true, the proof is completed and the theorem is proved.

In the rest of this section we present our embedding of SML in Nuprl and describe the weak division algorithm.

2.1 Embedding SML in Nuprl

Standard ML is a very high level programming language and is based upon a formal definition which prescribes the precise semantics of the language [RH90]. SML is primarily a higher order functional language, but it does support some non-functional features, such as sequential operations, references, and exception handling. SML is strongly typed and polymorphic, thus it closely resembles much of the Nuprl type system.

Nuprl contains a set of primitive operations which are the basis for its computation system. Many SML instructions are very similar to these primitive operations. By limiting ourselves to only these operations, it was quite easy to embed a subset of SML in Nuprl. The primitive operations upon which we based our subset include integer arithmetic, list recursion, integer equality, string equality, and pairing. For these primitive operations we developed a set of constructs which have the same behavior in SML and in Nuprl, but which have different implementations in each system. These constructs are either part of the SML Definition, or are so simple that the correspondence between the theorems in Nuprl and the code in SML is obvious. The principal features which we did not include in our subset (because of the difficulty of implementing them in terms of the Nuprl primitives) are: references, exceptions, sequentiality, primitive recursion, pattern matching, real numbers, modules, streams, and records.

In this section we demonstrate our method for embedding SML in Nuprl by showing the definition of our list recursion function in Nuprl and SML. Nuprl has a list recursion operation (*list_ind*), while SML does primitive recursion (the name of the function appears within the body of the function). In accordance with our plan to embed SML in Nuprl, we began by creating a function (*recurse*) in Nuprl based upon the *list_ind* operator. We then proved several theorems describing the behavior of *recurse*. Based upon these theorems we wrote *recurse* in SML. Because SML allows primitive recursion, we were able to write the implementation of *recurse* in SML to match the Nuprl theorems that describe its behavior. All other functions which do list recursion are built on top of *recurse*.

The function *recurse* recurses over a list, applying a function at each step. Equations 1 and 2 show the semantics for *list_ind* in Nuprl by describing its behavior on an empty list and on a non-empty list. In these equations *nil* represents an empty list, *hd::tl* is a non-empty list consisting of a first element (*hd*) and a tail list (*tl*), *nil_val* is the value to return when the list is empty and *f* is the function to apply when recursing down the list. This definition allows very general list recursion terms to be written, while still guaranteeing that the function will terminate.

$$\mathit{list\_ind}(\mathit{nil};\ \mathit{nil\_val};\ f) = \mathit{nil\_val} \tag{1}$$

$$\mathit{list\_ind}(\mathit{hd}::\mathit{tl};\ \mathit{nil\_val};\ f) = f\,(\mathit{hd})\ (\mathit{tl})\ (\mathit{list\_ind}(\mathit{tl};\ \mathit{nil\_val};\ f)) \tag{2}$$

Equation 3 shows the definition of *recurse* in Nuprl.

$$\mathit{recurse}\ f\ \mathit{nil\_val}\ \mathit{a\_list} = \mathit{list\_ind}(\mathit{a\_list};\ \mathit{nil\_val};\ f) \tag{3}$$

Based upon this definition we proved Theorems 1 and 2. These theorems were trivial to prove, because they are essentially identical to the semantics which are defined for the *list_ind* operator.

Theorem 1 *Recurse – base case*

$$\vdash \forall f,\ nil\_val.\quad recurse\ f\ nil\_val\ nil = nil\_val$$

Theorem 2 *Recurse – inductive case*

$$\vdash \forall f,\ hd,\ tl,\ nil\_val.\quad recurse\ f\ nil\_val\ (hd::tl) = f(hd)(tl)(recurse\ f\ nil\_val\ tl)$$

Based upon these theorems, we wrote *recurse* in SML as shown in Equation 4. Having defined the function *recurse*, we then used it in our implementation of other functions and used Theorems 1 and 2 to prove theorems describing the behavior of functions built upon *recurse*. Using this methodology, SML functions built upon this base can be verified in Nuprl and can be used in real programs that are compiled and run using an SML compiler.

$$\begin{array}{llllll} fun & recurse & f & nil\_val & nil & = nil\_val \\ \mid & recurse & f & nil\_val & (hd::tl) & = f\ (hd)\ (tl)\ (recurse\ f\ nil\_val\ tl) \end{array} \tag{4}$$

The only informal link in the connection between Nuprl and SML arises because Nuprl uses lazy evaluation and SML uses eager evaluation. In reality, this does not pose a problem for us, because the subset of SML that we are using is purely functional and all of the functions are guaranteed to terminate. Thus, for the subset that we are using, programs will have identical behavior in Nuprl and in an SML compiler[1].

2.2 Weak Division Algorithm

The weak division algorithm seeks to decrease circuit area by removing redundant combinational logic. A sub-circuit contains redundant logic if it implements precisely the same function as another. Weak division removes redundant logic by finding common subexpressions among the *diviscrs* of different functions. The common subexpressions are replaced by new intermediate variables. This results in the duplicated logic being implemented only once, thereby reducing the area of the circuit.

For example, Equation 5 contains two functions, one defining the variable p and one defining the variable q.

$$\begin{array}{l} p = (a \wedge b \wedge c) \vee (a \wedge b \wedge d) \vee (a \wedge b \wedge e) \\ q = (g \wedge c) \vee (g \wedge d) \vee h \end{array} \tag{5}$$

[1]Ongoing research at Cornell includes work aimed at creating a type theoretic semantics for SML within Nuprl. Once this has been done, programs will be able to be verified without relying on informal arguments to show the correspondence between the Nuprl and SML semantics.

There is one common subexpression, $(c \lor d)$, among the divisors of p and q. We can substitute a new variable z into the equations in place of $(c \lor d)$. Next, we can substitute a new variable (x) for the term $(a \land b)$, which appears multiple times in the expression for p. This results in the set of equations shown below.

$$
\begin{aligned}
p &= x \land z \lor x \land e \\
q &= (g \land z) \lor h \\
z &= c \lor d \\
x &= a \land b
\end{aligned}
\tag{6}
$$

These substitutions have reduced the size of the circuit from twelve two input gate equivalents to seven, because the factors $(c \lor d)$ and $(a \land b)$ are now only implemented once. The substitutions increased the delay through the circuit from two gate delays to three, because the signals z and x added an additional layer of logic to the circuit. We have found that as the size of circuits increase the reduction in area increases significantly, but the additional delay converges rapidly: we achieved reductions in area of 88% for circuits with more than three thousand gates, but yet added no more than nine additional layers of logic.

In comparing these results to those produced by *Mis* [BR+87], a conventional implementation of the weak division algorithm, we found that *Pbs* produces results of comparable quality to those of *Mis*, but that *Pbs* takes longer to run [AL92]. The longer run times are primarily due to the fact that the purpose of *Pbs* is to demonstrate that formally verified synthesis tools are a viable alternative to conventional tools. Because we knew from the beginning that we were going to prove *Pbs*, we aimed our implementation more towards code that was easy to verify than code which was fully optimized. In comparison, when developing a conventional tool, such as *Mis*, a great deal of effort is put into writing code which will run as fast as possible. The price that is paid for the extra speed in a program such as *Mis* is that the code would be extremely difficult to verify. If we wished to develop a more efficient implementation, we could view our current code as a low level specification, and re-implement more efficient versions of the functions. This is similar to the technique used in hardware verification where each level of the design hierarchy acts as an implementation of the level above and a specification for the level below.

The goal of the weak division algorithm is to eliminate all redundant combinational logic in circuits. This is done by substituting new variables in for common subexpressions of divisors of expressions. There are two phases in the algorithm; each phase removes common subexpressions of a different type of divisor. The final result of this substitution of new variables for common divisors is a system of Boolean equations without any duplicated combinational logic. A more formal way to state this is that the largest divisors shared by any pair of equations in the system consist solely of single literals [BM82].

The reason for a two phase approach is to increase the efficiency of the algorithm. To remove all duplicated combinational logic it is necessary to eliminate all common subexpressions of all divisors. Calculating the complete set of divisors is an extremely time intensive process. The principal advantage of the weak division algorithm is that it has the same power as removing all subexpressions of all common divisors, but it does not enumerate all the divisors. Instead, a subset of the divisors, known as the *kernels*, are calculated in the first phase of the algorithm, which is called *distill.* (The calculation of kernels is discussed in detail in Section 4.3.) In the second phase of the algorithm, known as *condense*, the *common cubes* are calculated. Verifying a program which implements the weak division algorithm requires first showing that the algorithm is correct: that is, prove that removing common subexpressions of kernels followed by common cubes is equivalent to removing all common divisors. Second, the program must be shown to be a correct implementation of the algorithm. This requires showing that the program removes all common subexpressions of kernels and then removes all common cubes.

3 The Methodology

In this section we outline the methodology we discovered while implementing and verifying *Pbs*. Section 4 demonstrates the methdology by describing the implementation and verification of three different functions in *Pbs*. The methodology can be summarized as:

- Make extensive use of modular and higher order functions
- Write short, simple functions
- Describe the behavior of functions with *characterization lemmas*
- Use characterization lemmas in proofs of abstract properties of functions
- Use *induction lemmas* for complicated functions

The first two part of the methodology are not so much methods, but rather guidelines. In order for it to be viable to prove properties about a program, the program itself must be well written. The good habits which programmers must follow in order to write verifiable code include many obvious points, such as the use of very modular code and higher order functions. This style of programming allows each function and its corresponding lemmas to be used many times. When just writing code, it may seem easier to simply duplicate a piece of code if it is extremely short and is only used a few times. But, when proving code correct, not only must the code be duplicated, but the proofs describing the code must also be duplicated.

In addition, each function should be as short and simple as possible. This will allow the behavior of the function to be described in a few simple lemmas. It is almost always a good idea to break up a single complex function into several smaller functions, even if the code is not used anywhere else. Linger, in describing the use of the Cleanroom methodology, says that to make the verification more tractable "simpler design approaches were actively sought in reviews and redesigning for simplicity was made an explicit objective." [LM88]

In our methodology, we make use of three categories of theorems. The first, called *characterization lemmas*, includes theorems which describe the behavior of functions at a level which is very close to the implementation. The second category is comprised of theorems describing abstract behavioral properties of functions. For complex functions, or functions for which there are a large number of abstract properties that must be verified, *induction lemmas* are used to structure proofs and capture reasoning which would otherwise be common to multiple proofs.

The purpose of characterization lemmas is to provide an implementation independent description of the behavior of a function. Characterization lemmas describe the behavior of a function for a specific set of inputs. For example, for functions which recurse over lists we use one characterization lemma for the case when the list is empty and one lemma for when the list is not empty. For functions based on *if-then-else*, we use one lemma for the true case and one for the false case. The proofs of abstract properties rely upon characterization lemmas and do not make direct use of the implementation of the function. This allows us to abstract away from implementations so that minor changes in implementations only affect proofs of the characterization lemmas.

For functions which require complicated implementations, or for functions which have a large number of abstract properties that need to be verified, induction lemmas are used. Just as an induction lemma for lists breaks down a proof about lists into a base case and an inductive case, an induction lemma for a function breaks down proofs about the behavior of the function into a number of simpler cases. Similarly, just as the inductions lemma for lists is parameterized by the proposition to be proved about lists, an induction lemma for a function is parameterized by the proposition that is to proved about the function. Redundant reasoning is avoided because, without the use of induction lemmas, all of the proofs about a function would most likely contain the same steps for simplifying the proof goal.

The problem with relying solely on characterization lemmas for complex functions is that it may take many steps to iteratively perform induction and apply characterization lemmas to simplify the goal for each of the parameters to a function. For example, in the function f (Equation 7), there is a nested *if-then-else*. This function requires three characterization lemmas, the rewrite rules of which are shown in Equations 8 – 10.

$$\begin{aligned} &\mathit{fun}\ f\ a\ b = \mathit{if}\ a\ \mathit{then}\ (\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) \\ &\qquad\qquad\quad \mathit{else}\ g(b) \end{aligned} \tag{7}$$

$$f(\mathit{true},\ \mathit{true}) \quad = x \tag{8}$$
$$f(\mathit{true},\ \mathit{false}) \quad = y \tag{9}$$
$$f(\mathit{false},\ b) \quad = g(b) \tag{10}$$

Theorems describing a behavioral property of f can be written as shown in Theorem 3, where $\mathcal{P}$ is the property to be verified.

Theorem 3 *Generic theorem for f*

$\vdash \forall a,\ b\ .\ \mathcal{P}(a,\ b,\ f(a,b))$

To verify a theorem in the form of Theorem 3, we would begin by performing induction on a. This would give us two subgoals, one for the case where a is true and one for the case where a is false. We would then have to perform induction on b in the first subgoal, which would give us two additional subgoals. This would leave us with a total of three subgoals; and we would have to apply one of the characterization lemmas to each of these subgoals to simplify the instantiation of f. In all this would take us about five steps. These five steps would most likely be repeated in every single proof about f. Using an induction lemma for f would allow us to perform these steps just once.

We construct induction lemmas by examining the characterization lemmas, and writing one clause for each characterization lemma. An induction lemma for f would say that for any proposition ($\mathcal{P}$) to be true about f, the proposition must be true for the cases where a and b are both true, where a is true and b is false, and where a is false.

Lemma 1 is an induction lemma for f. We have three characterization lemmas for f, so we have three clauses on the left hand side of the implication. The first clause says that the proposition $\mathcal{P}$ is true when a and b are both true, the second clause says that $\mathcal{P}$ is true when a is true and b is false, and the third clause says that $\mathcal{P}$ is true when a is false.

Lemma 1 *Induction lemma for f*

$$\begin{array}{l} \vdash \forall \mathcal{P} \,. \\ \quad \mathcal{P}(\mathit{true},\ \mathit{true},\ x)\ \& \\ \quad \mathcal{P}(\mathit{true},\ \mathit{false},\ y)\ \& \\ \quad \forall b \,.\ \mathcal{P}(\mathit{false},\ b,\ g(b)) \Rightarrow \\ \qquad \forall a, b \,.\ \mathcal{P}(a,\ b,\ f(a,\ b)) \end{array}$$

Now, given a theorem written in the form of Theorem 3, we can apply Lemma 1 and simplify the goal as shown in Proof Step 1.

$\vdash \forall a, b \,.\ \mathcal{P}(a,\ b,\ f(a,\ b))$

BY *Lemma 'f_induction'*

$\vdash \mathcal{P}(\mathit{true},\ \mathit{true},\ x)$

$\vdash \mathcal{P}(\mathit{true},\ \mathit{false},\ y)$

$\vdash \forall b \,.\ \mathcal{P}(\mathit{false},\ b,\ g(b)))$

Proof Step 1: Application of induction lemma

In many cases, such as the simple example here, after applying the induction lemma in a proof, the function will no longer appear in the proof. This occurs whenever the function being described is not recursive. It may also occur if there is extra information about the properties which will be verified, as is true with the function described in Section 4.3. When the function being described does not appear in the proof after the application of the induction lemma, then the induction lemma has captured all of the reasoning which is specific to the function.

Based upon our experiences, we believe that just as programming style becomes increasingly important as the size of programs increases, proof style becomes increasingly important as the size of a verification effort increases. The methodology described here is a uniform and systematic approach that may be used to implement and verify a wide variety of software programs. The methodology requires a proof development system which uses a higher order logic. This is because induction lemmas are parameterized by propositions. Also, we have found the ability to write higher order functions to be very useful in developing software that is geared towards verification.

4 Examples from the Implementation and Verification of PBS

In this section we illustrate our methodology by describing the implementation and verification of three functions. The first example (Section 4.1) is the membership function, which tests if an element is a member of a list. It is a simple function and is used to introduce our style of proof and describe how characterization lemmas are proved. In the next two sections we describe the verification of two other functions. These functions are much more complex than the membership function, and so entire proofs are not shown. Instead we concentrate on the interesting aspects of the verification of each function. Section 4.2 describes a function for dividing an expression by a cube and is an example of using characterization lemmas to verify more abstract properties of a function. The most complex function in *Pbs* is described in Section 4.3 and shows the use of induction lemmas. Throughout this section we use SML notation for pieces of code; our proof notation is described in Appendix A.3.

4.1 The Membership Function – A Simple Example

The first functions which we wrote in *Pbs* were simple operations on lists, such as testing if an element is a member of a list and deleting an element from a list. These two functions, testing for membership and deleting, are the basis for almost every other function in *Pbs*. This is because in *Pbs* all but one of the data types are built from lists. For example, a *cube*, which is a conjunction of *literals*, is represented as a list of literals; and an *expression*, which is a disjunction of cubes, is represented as a list of cubes. Dividing an expression (a list of cubes) by a cube (a list of literals) consists primarily of testing if literals are members of cubes and deleting literals from cubes. This division function is described in detail in Section 4.2.

Because membership in a list is such a fundamental operation in *Pbs*, and because it is a simple function, we use it as the first example for describing the techniques we used. Our membership function is parameterized by an equivalence function *eq_fn*. In *Pbs*, we have separate equivalence functions for literals, cubes and expressions. This allows us to use the same membership function in many different situations.

For functions which operate on lists, it is common to define their behavior in an inductive style. To do this for the membership function, we wrote characterization lemmas (Theorems 4 and 5) which describe the behavior of the function for the base case (*nil*) and for the inductive case (*hd::tl*)

Theorem 4 *Membership in the empty list*

$\vdash \forall eq\_fn,\ x\ .\ \neg(mem(eq\_fn)(x)(nil))$

Theorem 5 *Membership in a non-empty list*

$\vdash \forall eq\_fn,\ x,\ hd,\ tl\ .$
$\quad mem(eq\_fn)(x)(hd::tl) \iff eq\_fn(x)(hd) \ \vee\ mem(eq\_fn)(x)(tl)$

Now that we have defined the desired behavior of our membership function, we can write the code for it. Our function takes three parameters: an equivalence function (*eq_fn*), an element (x) and a list (*a_list*). The function tests each element in *a_list* to determine if it is equal to x under the equivalence function *eq_fn*. If an element is found which is equivalent to x then *mem* returns *true*, otherwise it returns *false*. To test each element of *a_list*, we recurse over the list using a higher order function named *reduce*. The function *reduce* is implemented in terms of *recurse*, which was described in Section 2.1. The function takes three parameters: the function to apply to each element of the list (f), the value to return when the list is empty (*nil_val*) and the list to recurse over. Equations 11 and 12 define the behavior of *reduce* for the base and inductive cases[2].

$$reduce\ f\ nil\_val\ nil = nil\_val \tag{11}$$

$$reduce\ f\ nil\_val\ (hd::tl) = f\,(hd)\ (reduce\ f\ nil\_val\ tl) \tag{12}$$

Figure 1 shows our implementation of the function *mem*. Inside the body of the function a local function named f is declared. The function f takes two parameters, the current head of the list (*hd*) and the result of testing for membership in the tail of the list. The function returns *true* if x is equivalent to *hd* or if the result is *true*.

[2]As an alternative to the approach taken here, we could have defined the membership function in such a way that Nuprl could have verified it automatically. This approach would have been similar to that of proof systems which are capable of automatically verifying many inductively defined functions [BM88]. We could have done this by writing the function directly in terms of Nuprl's primitive list induction operator. For us, there were several disadvantages to choosing this alternative. Most importantly, it would prevent our implementation of *Pbs* in Nuprl from being the same as our implementation in SML. Secondly, verifying more complicated functions in this alternative style would be more difficult than in the style which we used. By using the function *recurse* as the only primitive function for recursion and implementing other functions such as *reduce* in terms of *recurse*, we were able to hide the implementation details of recursion and thereby prevent our proofs from becoming cluttered with low level details.

```
fun mem eq_fn x a_list =
    let
        fun f hd result = eq_fn(x)(hd) orelse result
    in
        reduce f false a_list
    end
```

Figure 1: Implementation of membership function

Now that we have implemented the membership function, we need to prove that it satisfies the theorems that we used to described its desired behavior. The first theorem, Theorem 4, is easily proved by direct computation and Theorem 5 can be proved using lemmas to describe the behavior of the recursion function *reduce* and the Boolean disjunction function *orelse*. The complete proof, which requires four steps, appears in Figure 2 and is described in the following paragraphs. In the proof, only the conclusion and the rule for each step are shown. The hypotheses contain variable declarations and are not modified in the proof. The rules, which appear after "**BY**", are the only text other than the initial goal that the user types in.

Theorem 5 says that an element (x) is a member of a list consisting of a head element (hd) and a tail list (tl) if and only if x is equal to hd or x is a member of tl. The proof begins by unfolding the definition of *mem* to expose that it is implemented in terms of *reduce*. In the second step a rewrite rule based on Equation 12 is used to simplify the inductive call to reduce. At this point, the right hand side of the *orelse* is equivalent to testing if x is a member of tl, so the definition of *mem* is folded back up. The proof is completed by replacing the SML function *orelse* with a logical disjunction ($\vee$).

One of the first steps of each proof is to unfold the definition of the function being described. In Nuprl "unfolding" means to replace an instantiation of a function with the code used to implement the function. It is analogous to the compiler optimization of in-line expansion. The purpose of unfolding definitions is to reveal the implementation of functions. When this has been done, rewrite rules or lemmas describing lower level functions can be used in the proof.

As illustrated here, unfolding is really just one type of rewriting that can be performed. Nuprl has a very powerful rewriting package, which is used to replace one term with another term, where the two terms are related by an equivalence relation. In the proof of Theorem 5, four different rewrite rules are used. In the first and third steps, direct computation rules are used to unfold and fold the instantiation of *mem*. The rewrite rules used in steps two and four are derived from lemmas that were proved about the functions *reduce* and *orelse*.

By adopting the proof style described here, we were able to write concise characterization lemmas describing complex functions. The purpose of this example was to introduce our notation and proof style and to provide a detailed illustration of a simple proof in Nuprl.

$\vdash$ (*mem eq_fn x* (*hd::tl*))
$\Longleftrightarrow$
(*eq_fn x hd*) $\vee$ (*mem eq_fn x tl*)

BY *RewriteConcl* (*UnfoldFirst* '*mem*')

$\vdash$ (*reduce* (*fn hd* $\Rightarrow$ *fn result* $\Rightarrow$ (*eq_fn x hd*) *orelse result*) *false* (*hd::tl*))
$\Longleftrightarrow$
(*eq_fn x hd*) $\vee$ (*mem eq_fn x tl*)

BY *RewriteConcl reduce_ht_convn*

$\vdash$ (*eq_fn x hd*) *orelse*
reduce (*fn hd* $\Rightarrow$ *fn result* $\Rightarrow$ (*eq_fn*(x))(*hd*) *orelse result*) *false tl*)
$\Longleftrightarrow$
(*eq_fn x hd*) $\vee$ (*mem eq_fn x tl*)

BY *RewriteConcl* (*Fold* '*mem*')

$\vdash$ (*eq_fn x hd*) *orelse* (*mem eq_fn x tl*)
$\Longleftrightarrow$
(*eq_fn x hd*) $\vee$ (*mem eq_fn x tl*)

BY *RewriteConcl orelse_x_x_convn*

Figure 2: Proof of Theorem 5

4.2 The Quotient Function – A Typical Example

This section illustrates the verification of a typical function in *Pbs*. Section 4.1 showed how we proved characterization lemmas for functions; this section demonstrates how we used characterization lemmas to verify abstract properties about functions. The function chosen here calculates the quotient produced by dividing an expression by a cube. Recall that a *cube* is a conjunction of literals and an *expression* is a disjunction of cubes. (Definitions which are specific to the weak division algorithm are given in Appendix A.1.) Dividing an expression by a cube is the first function in *Pbs* which is specific to the weak division algorithm. Almost all of the lower level functions are general purpose Boolean operations, such as calculating the conjunction of two cubes or the disjunction of two expressions. This quotient function is crucial in *Pbs*, because most of the operations in the weak division algorithm are based upon dividing an expression by a cube. Dividing one expression by another expression, which is the function from which the weak division algorithm gets its name, begins by dividing the dividend expression by each of the cubes in the divisor expression. Substituting an intermediate variable into an expression in place of a common cube requires dividing the expression by the common cube. We begin by describing the algorithm for dividing an expression by a cube, and then present the proof of correctness for our implementation.

A cube (c_2)[3] is divided into an expression (e_1) by finding all of the cubes in e_1 which are supersets of c_2. The quotient is produced by deleting the literals in c_2 from each of these cubes. Equation 13 is a pseudo code description of this division operation.

$$\begin{array}{l} QUOT\_ec\ e_1\ c_2 = \\ \quad quot = \emptyset \\ \quad \textit{for each cube}\ c_1 \in e_1 \\ \quad\quad \textit{if}\ c_2 \subseteq c_1 \\ \quad\quad \textit{then}\ quot = quot \cup\ (c_1 - c_2) \\ \quad \textit{return}\ quot \end{array} \tag{13}$$

An example of dividing a cube into an expression is shown in Figure 3. To divide the cube $(a \wedge b)$ into the expression, we test if $(a \wedge b)$ is a subset of each of the cubes in the expression. For the first two cubes, $(a \wedge b \wedge c)$ and $(a \wedge b \wedge d)$, $(a \wedge b)$ is a subset, so we delete the literals x and b from the cubes. This results in c and d respectively. The cube $(a \wedge b)$ is not a subset of $(a \wedge g \wedge e)$, so the result is the empty set. In the last step the results are combined together to form the final quotient.

$$QUOT\_ec \quad ((a \wedge b \wedge c) \vee (a \wedge b \wedge d) \vee (a \wedge g \wedge e)) \quad (a \wedge b) \quad =$$

Initially: $quot = \emptyset$

For each cube $(a \wedge b \wedge c)$, $(a \wedge b \wedge d)$, $(a \wedge g \wedge e)$

$$\begin{array}{lcll} (a \wedge b) \subseteq (a \wedge b \wedge c) & \mapsto & quot = \emptyset \cup \{c\} & = \{c\} \\ (a \wedge b) \subseteq (a \wedge b \wedge d) & \mapsto & quot = \{c\} \cup \{d\} & = \{c,d\} \\ (a \wedge b) \not\subseteq (a \wedge g \wedge e) & \mapsto & quot = \emptyset \cup \{c,d\} & = \{c,d\} \end{array}$$

Result: $quot = (c \vee d)$

Figure 3: Example of dividing a cube into an expression

Our implementation of $QUOT\_ec$ is shown in Figure 4. For each data type there is a function which converts the data type into a list of the component type. The component type is the type from which the data type is built: cubes are the component type of expressions. A corresponding function

[3]Throughout the rest of this section and Section 4.3 we will use the following notation: variables beginning with l are literals, c are cubes, and e are expressions. Also, a cube which has the same subscript as an expression is a member of the expression. These are simply notational conventions used to make it easier to understand theorems and functions; there is no semantic meaning assigned to the names.

```
fun QUOT_ec(e1)(c2) =
    let
        fun f c1 result =
                if IN_cc(c2)(c1)
                then DEL_cc(c1)(c2)::result
                else result
    in
        Cs2E(reduce f NIL_e (E2Cs(e1)))
    end
```

Figure 4: Function for quotient of an expression and a cube

Function	Purpose
NIL_e	Base case (empty expression).
Cs2E	Convert list of cubes into an expression.
E2Cs	Convert expression into list of cubes.
CONS_ce	Concatenate a cube onto an expression.

Figure 5: Type conversion functions for expression data type.

is used to convert a list of the component type into the data type. Also, there is a special constant which represents the base case (empty list) of the data type. Based upon these primitive functions we wrote other functions to perform such tasks as concatenating a member of the component type onto the data type. Figure 5 lists these functions for the data type of expressions.

In *QUOT_ec*, *E2Cs* is used to convert the expression e_1 to a list of cubes, so that *reduce* can recurse over e_1. For each of the cubes (c_1) in e_1, we test if c_2 is a subset of c_1 using the function *IN_cc*. If c_2 (the divisor cube) is a subset of c_1 (the cube from e_1), then the literals in c_1 are deleted from the literals in c_2 using the function *DEL_cc*; the resultant cube is then concatenated onto the previous result. If c_2 is not a subset of c_1, then the previous result is returned. The type of the result of *QUOT_ec* should be an expression, so *Cs2E* is used to convert the list of cubes returned by *reduce* to an expression.

Three characterization lemmas are needed for this function: one for an empty expression and two for non-empty expressions. One of the characterization lemmas for non-empty expressions is for the case in which c_2 is a subset of c_1 and one is for when c_2 is not a subset of c_1. Lemma 2 states that the quotient of an empty expression and any cube is an empty expression. Lemma 3 states that if c_2 is a subset of c_1 then the quotient of *CONS_ce*$(c_1)(e_1)$ and c_2 is equal to *DEL_cc*$(c_1)(c_2)$ concatenated with the quotient of e_1 and c_2. For the purposes here, the behavior of *DEL_cc* is not important. This is because the lemma is describing the behavior of *QUOT_ec* in terms of lower level functions, such as *DEL_cc*. Lemma 4 states that if c_2 is not a subset of c_1 then the quotient of $CONS\_ce(c_1)(e_1)$ and c_2 is the empty expression (NIL_e).

Lemma 2 *Base case of quotient*

$$\begin{array}{l} \vdash \forall c_2{:}Cube\_t. \\ \quad\quad QUOT\_ec(NIL\_e)(c_2) = NIL\_e \end{array}$$

Lemma 3 *Positive inductive case of quotient*

$$\begin{array}{l} \vdash \forall c_1, c_2{:}Cube\_t. \\ \quad\quad \forall e_1{:}Expr\_t. \\ \quad\quad\quad IN\_cc(c_2)(c_1) \Rightarrow \\ \quad\quad\quad\quad QUOT\_ec(CONS\_ce(c_1)(e_1))(c_2) = \\ \quad\quad\quad\quad CONS\_ce(DEL\_cc(c_1)(c_2))(QUOT\_ec(e_1)(c_2)) \end{array}$$

Lemma 4 *Negative inductive case of quotient*

$$\begin{array}{l} \vdash \forall c_1, c_2{:}Cube\_t. \\ \quad\quad \forall e_1{:}Expr\_t. \\ \quad\quad\quad \neg IN\_cc(c_2)(c_1) \Rightarrow \\ \quad\quad\quad\quad QUOT\_ec(CONS\_ce(c_1)(e_1))(c_2) = NIL\_e \end{array}$$

After completing the proofs of Lemmas 2, 3 and 4, rewrite rules based on these lemmas (Equations 14, 15 and 16) can be derived.

$$QUOT\_ec(NIL\_e)(c_2) = NIL\_e \tag{14}$$

$$\begin{array}{l} IN\_cc(c_2)(c_1) \Rightarrow \\ \quad QUOT\_ec(CONS\_ce(c_1)(e_1))(c_2) = \\ \quad\quad CONS\_ce(DEL\_cc(c_2)(c_1))(QUOT\_ec(e_1)(c_2)) \end{array} \tag{15}$$

$$\begin{array}{l} \neg IN\_cc(c_2)(c_1) \Rightarrow \\ \quad QUOT\_ec(CONS\_ce(c_1)(e_1))(c_2) = QUOT\_ec(e_1)(c_2) \end{array} \tag{16}$$

Having created rewrite rules describing the behavior of *QUOT_ec*, we can use the rewrite rules to verify a more abstract theorem about the function. In *Pbs*, Theorem 6 is the primary theorem used to describe the behavior of *QUOT_ec*[4]. The theorem describes the conditions under which a cube is a member of the quotient produced by dividing an expression by a cube. This theorem states that a cube (co) is a member of the quotient of an expression (e_1) and another cube (c_2) if and only if there is a cube (c_1) in e_1 such that co is equal to c_2 deleted from c_1. An informal argument for the correctness of the theorem is:

[4]In Section 4.1, quantified variables were not shown with their types for simplicity. In the rest of this paper, we will include the types for quantified variables as they appear in Nuprl, so as to help clarify the meanings of the theorems. Appendix A.2 lists the types used in the implementation of *Pbs*.

Theorem 6 *Membership in a quotient*

$$
\begin{array}{l}
\vdash \forall e_1{:}Expr\_t. \\
\quad \forall c_2, co{:}Cube\_t. \\
\quad\quad MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \iff \\
\quad\quad \exists\ c_1{:}Cube\_t. \\
\quad\quad\quad MEM\_ce(c_1)(e_1)\ \& \\
\quad\quad\quad EQc\ co\ (DEL\_cc(c_1)(c_2))
\end{array}
$$

The quotient of an expression (e_1) and a cube (c_2), is produced by deleting c_1 from each of the cubes in e_1 which c_1 is a subset of.

We can restate this as: each cube in the quotient of e_1 and c_2 is produced by deleting c_2 from one of the cubes in e_1. Therefore, if a cube (co) is a member of the quotient of e_1 and c_2, then there must exist a cube (c_1), which is a member of e_1, such that co is equal to c_2 deleted from c_1.

We want our proof to use the characterization lemmas and not be directly dependent on the implementation of the quotient function. Because of this we use the rewrite rules (Equations 14, 15 and 16), which were derived from characterization lemmas. The rule in Equation 14 says the quotient of an empty expression and any cube is an empty expression. The rules in Equations 15 and 16 describe how the quotient of a non-empty expression and a cube is formed. There are two rules for the case of a non-empty expression because *QUOT_ec* acts differently depending upon if the cube (c_2) being divided into the expression is a subset of the cube (c_1) being concatenated onto the expression.

The structure of our proof of Theorem 6 will be to break the theorem down into a number of subgoals, so that we can make use of the three rewrite rules. The rules are stated as a base case and two inductive cases, so our proof of Theorem 6 will begin by performing induction on the expression which we wish to take the quotient of. The proof of the base case will use the rewrite rule from Equation 14. For the inductive case, we will do a case split on whether the the cube being divided into the expression is a subset of the cube being concatenated onto the expression. At this point, we will be able to apply the rewrite rules from Equations 15 and 16. After applying the rewrite rules we will use a variety of lemmas and general mathematical reasoning to finish the proof.

Figure 6 shows the proof tree for Theorem 6. The steps in the proof which are described in the rest of this section are labeled with the number of the lemma or proof step that they correspond to. Steps which are not described are shown as dashed triangles.

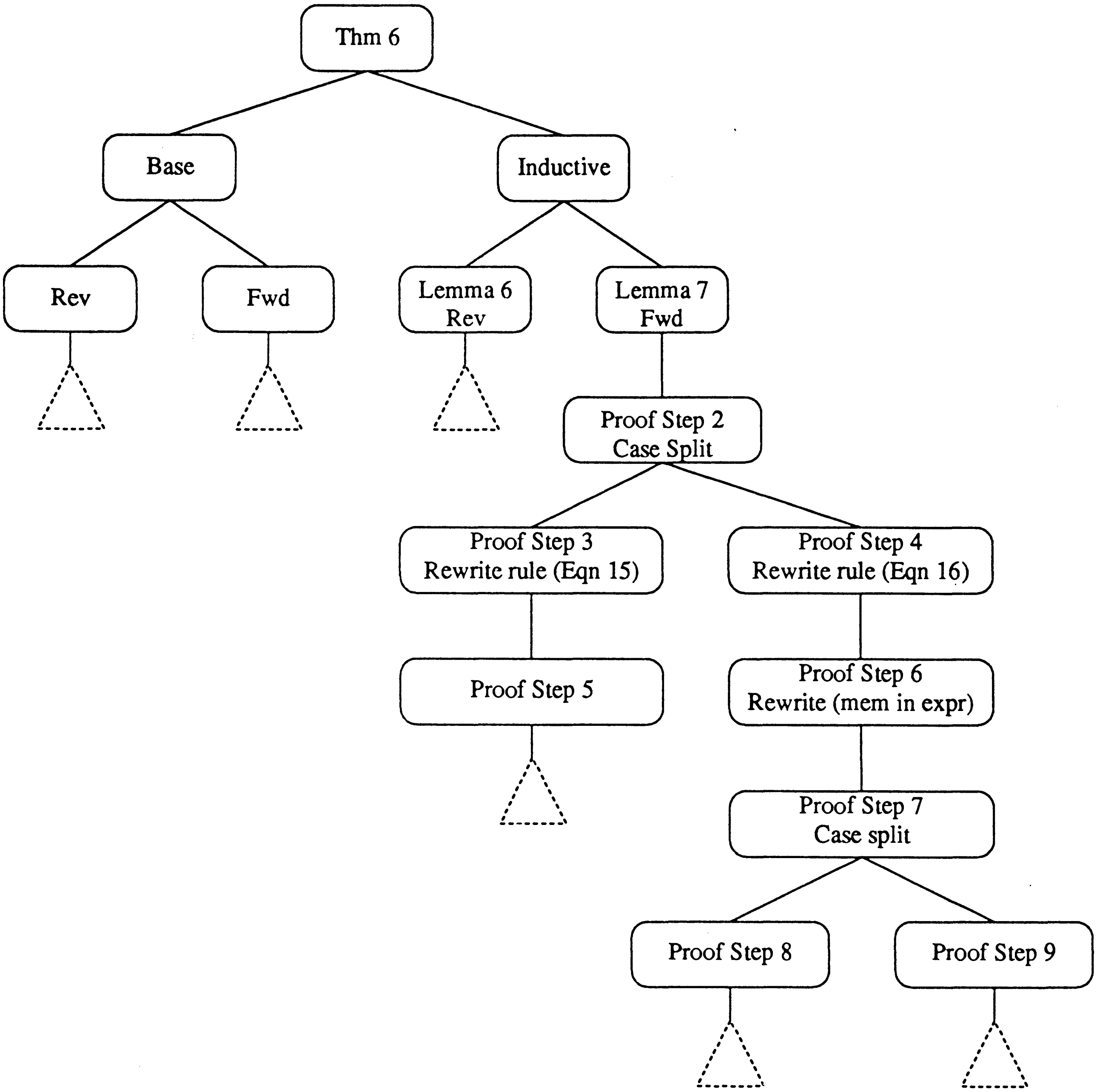

Figure 6: Proof tree of Theorem 6

We begin the proof of Theorem 6 by performing induction on the expression e_1. This results in a base case and an inductive case. The base case is quite simple and will not be dealt with here. For the inductive case we can split the *if-and-only-if* into two implications, resulting in two subgoals (Lemmas 5 and 6). For the rest of this section, the proof of the forward direction (Lemma 6) is discussed in detail. The proof of the reverse direction is similar.

Lemma 5 *Reverse direction of inductive case*

$$
\begin{array}{l}
\vdash \forall e_1{:}\mathit{Expr\_t}.\\
\quad\quad \forall c_2, co, hd{:}\mathit{Cube\_t}.\\
\quad\quad\quad (\exists\ c_1{:}\mathit{Cube\_t}.\\
\quad\quad\quad\quad \mathit{MEM\_ce}(c_1)(e_1)\ \&\\
\quad\quad\quad\quad \mathit{EQc}(co)(\mathit{DEL\_cc}(c_1)(c_2))\ \Rightarrow\\
\quad\quad\quad \mathit{MEM\_ce}(co)(\mathit{QUOT\_ec}(e_1)(c_2)))\ \Rightarrow\\
\quad\quad\quad\quad\quad (\exists\ c_1{:}\mathit{Cube\_t}.\\
\quad\quad\quad\quad\quad\quad \mathit{MEM\_ce}(c_1)(\mathit{CONS\_ce}(hd)(e_1))\ \&\\
\quad\quad\quad\quad\quad\quad \mathit{EQc}(co)(\mathit{DEL\_cc}(c_1)(c_2))\ \Rightarrow\\
\quad\quad\quad\quad\quad \mathit{MEM\_ce}(co)(\mathit{QUOT\_ec}(\mathit{CONS\_ce}(hd)(e_1))(c_2)))
\end{array}
$$

Lemma 6 *Forward direction of inductive case*

$$
\begin{array}{l}
\vdash \forall e_1{:}\mathit{Expr\_t}.\\
\quad\quad \forall c_2, co, hd{:}\mathit{Cube\_t}.\\
\quad\quad\quad (\mathit{MEM\_ce}(co)(\mathit{QUOT\_ec}(e_1)(c_2))\ \Rightarrow\\
\quad\quad\quad\quad \exists\ c_1{:}\mathit{Cube\_t}.\\
\quad\quad\quad\quad\quad \mathit{MEM\_ce}(c_1)(e_1)\ \&\\
\quad\quad\quad\quad\quad \mathit{EQc}(co)(\mathit{DEL\_cc}(c_1)(c_2)))\ \Rightarrow\\
\quad\quad\quad (\mathit{MEM\_ce}(co)(\mathit{QUOT\_ec}(\mathit{CONS\_ce}(hd)(e_1))(c_2))\ \Rightarrow\\
\quad\quad\quad\quad \exists\ c_1{:}\mathit{Cube\_t}.\\
\quad\quad\quad\quad\quad \mathit{MEM\_ce}(c_1)(\mathit{CONS\_ce}(hd)(e_1))\ \&\\
\quad\quad\quad\quad\quad \mathit{EQc}(co)(\mathit{DEL\_cc}(c_1)(c_2)))
\end{array}
$$

The first step in almost every proof is to instantiate universally quantified variables and bring the left hand sides of implications into the hypothesis list. This step is done automatically; the result of performing this step to Lemma 6 is shown in Proof Step 2.

The first three hypotheses in Proof Step 2 are variable declarations. The fourth hypothesis is the inductive hypothesis. In order to make use of this hypothesis, we must show that the left hand side of the implication is true. The strategy for this proof will be to modify the fifth hypothesis so that it matches the left hand side of the implication in the inductive hypothesis. Currently, the left hand side of the inductive hypothesis says that the cube *co* is a member of the quotient of e_1 and c_2 and the fifth hypothesis says that *co* is a member of the quotient of $\mathit{CONS\_ce}(hd)(e_1)$ and c_2. In order to

1. e_1 : *Expr_t*
2. c_2 : *Cube_t*
3. *co* : *Cube_t*
4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
   $\exists c_1 : Cube\_t.$
   $MEM\_ce(c_1)(e_1)$ &
   $EQc(co)(DEL\_cc(c_1)(c_2))$
5. $MEM\_ce(co)(QUOT\_ec(CONS\_ce(hd)(e_1))(c_2))$

$\vdash \exists c_1 : Cube\_t.$
$MEM\_ce(c_1)(CONS\_ce(hd)(e_1))$ &
$EQc(co)(DEL\_cc(c_1)(c_2))$

Proof Step 2: Forward direction of inductive case after initial set up

transform hypothesis five to match the left hand side of hypothesis four, we must look at the rewrite rules based on the characterization lemmas for *QUOT_ec* (Equations 15 and 16) and understand what it means for a cube to be a member of the quotient of $CONS\_ce(hd)(e_1)$ and c_2.

4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
   $\exists c_1 : Cube\_t.$
   $MEM\_ce(c_1)(e_1)$ &
   $EQc(co)(DEL\_cc(c_1)(c_2))$
5. $MEM\_ce(co)(QUOT\_ec(CONS\_ce(hd)(e_1))(c_2))$
6. $\neg IN\_cc(c_2)(c_1)$

$\vdash \exists c_1 : Cube\_t.$
$MEM\_ce(c_1)(CONS\_ce(hd)(e_1))$ &
$EQc(co)(DEL\_cc(c_1)(c_2))$

Proof Step 3: Negative side of first case split

Due to the *if-then-else* in *QUOT_ec*, we have two rewrite rules: one for the case when c_2 is not a subset of c_1 and one for when c_2 is a subset of c_1. To use these rules, a case split is done on the code $IN\_cc(c_2)(c_1)$ (Proof Steps 3 and 4). A case split takes a Boolean value and creates two subgoals, one in which the value is true and one in which the value is not true. Proof Step 3 is the case where c_2 is not a subset of c_1 and Proof Step 4 is the case where c_2 is a subset of c_1. In the rest of this section we will not show the first

4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
$\quad\exists\ c_1{:}Cube\_t.$
$\quad\quad MEM\_ce(c_1)(e_1)\ \&$
$\quad\quad EQc(co)(DEL\_cc(c_1)(c_2))$
5. $MEM\_ce(co)(QUOT\_ec(CONS\_ce(hd)(e_1))(c_2))$
6. $IN\_cc(c_2)(c_1)$

$\vdash\ \exists c_1{:}Cube\_t.$
$\quad MEM\_ce(c_1)(CONS\_ce(hd)(e_1))\ \&$
$\quad EQc(co)(DEL\_cc(c_1)(c_2))$

Proof Step 4: Positive side of first case split

three hypothesis, because they are just variable declarations and are not changed in the rest of the proof.

At this point, the two rewrite rules (Equations 15 and 16) can be used, one for each side of the case split. Proof Steps 5 and 6 show the affect of applying the rewrite rules to hypothesis five for the negative case (Proof Step 3) and positive case (Proof Step 4) respectively.

4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
$\quad\exists\ c_1{:}Cube\_t.$
$\quad\quad MEM\_ce(c_1)(e_1)\ \&$
$\quad\quad EQc(co)(DEL\_cc(c_1)(c_2))$
5. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2))$
6. $\neg IN\_cc(c_2)(c_1)$

$\vdash\ \exists c_1{:}Cube\_t.$
$\quad MEM\_ce(c_1)(CONS\_ce(hd)(e_1))\ \&$
$\quad EQc(co)(DEL\_cc(c_1)(c_2))$

Proof Step 5: Negative side of first case split after application of rewrite rule from Equation 16

In Proof Step 5, hypothesis five matches the left hand side of hypothesis four, thus hypothesis four can now be used in the proof. There remains some minor work involving membership in lists to finish off this part of the proof. This work is very straightforward and will not be described here. For the positive side (Proof Step 6), there is still some work to be done before the inductive hypothesis may be used.

4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
$\exists\ c_1{:}Cube\_t.$
$MEM\_ce(c_1)(e_1)\ \&$
$EQc(co)(DEL\_cc(c_1)(c_2))$

5. $MEM\_ce(co)(CONS\_ce(DEL\_cc(hd)(c_2))QUOT\_ec(e_1)(c_2))$

$\vdash\ \exists c_1{:}Cube\_t.$
$MEM\_ce(c_1)(CONS\_ce(hd)(e_1))\ \&$
$EQc(co)(DEL\_cc(c_1)(c_2))$

Proof Step 6: Positive side of first case split after application of rewrite rule from Equation 15

Proof Step 7 shows the proof after applying the rewrite rule in Equation 17 to hypothesis five in Proof Step 6.

$$MEM\_ce\ c\ (CONS\_ce\ c\_hd\ e)\ \Longleftrightarrow\ (EQe\ c\ c\_hd)\ \vee\ (MEM\_ce\ c\ e) \tag{17}$$

This rewrite rule says that if a cube is a member of an expression consisting of a head cube and a tail expression, then the cube is either equal to the head cube or is a member of the tail expression. It is derived from Theorem 5, which describes membership in any non-empty list. After applying the rewrite rule, hypothesis five consists of a disjunction: either co is equal to $DEL\_cc(hd)(c_2)$ or co is a member of $QUOT\_ec(e_1)(c_2)$. Because of this disjunction, we split the proof into two subproofs: one for each side of the disjunction. In the left subproof, hypothesis five is $EQc(co)(DEL\_cc(hd)(c_2))$ and in the right subproof hypothesis five is $MEM\_ce(co)(QUOT\_ec(e_1)(c_2))$.

4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
$\exists\ c_1{:}Cube\_t.$
$MEM\_ce(c_1)(e_1)\ \&$
$EQc(co)(DEL\_cc(c_1)(c_2))$

5. $EQc(co)(DEL\_cc(hd)(c_2))\ \vee\ MEM\_ce(co)(QUOT\_ec(e_1)(c_2))$

$\vdash\ \exists c_1{:}Cube\_t.$
$MEM\_ce(c_1)(CONS\_ce(hd)(e_1))\ \&$
$EQc(co)(DEL\_cc(c_1)(c_2))$

Proof Step 7: Positive side after membership rewrite

The left subproof (where co is equal to $DEL\_cc(hd)(c_2)$) can be completed by instantiating c_1 in the conclusion with hd. This results in Proof Step 8, which contains two goals. The first goal says that hd is a member of $CONS\_ce(hd)(e_1)$, which is clearly true. The second goal says that co is equal to $DEL\_cc(hd)(c_2)$, which we know is true from hypothesis five. Thus, the left subproof has been completed.

---

4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
   $\exists\ c_1{:}Cube\_t.$
   $MEM\_ce(c_1)(e_1)\ \&$
   $EQc(co)(DEL\_cc(c_1)(c_2))$
5. $EQc(co)(DEL\_cc(hd)(c_2))$
6. $hd : Cube\_t$

$\vdash$ $MEM\_ce(hd)(CONS\_ce(hd)(e_1))$
$\vdash$ $EQc(co)(DEL\_cc(hd)(c_2))$

Proof Step 8: Left side of second split after instantiating conclusion with hd

---

In the right subproof (Proof Step 9), co is a member of $QUOT\_ec(e_1)(c_2)$. Here, hypothesis five matches the left hand side of the induction hypothesis (hypothesis four), so it can be completed in the same manner as Proof Step 5. This involves using hypothesis four; which is now possible because hypothesis five matches the left hand side of hypothesis four. Finally, some previously proven lemmas describing membership in lists are used to complete the proof.

---

4. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2)) \Rightarrow$
   $\exists\ c_1{:}Cube\_t.$
   $MEM\_ce(c_1)(e_1)\ \&$
   $EQc(co)(DEL\_cc(c_1)(c_2))$
5. $MEM\_ce(co)(QUOT\_ec(e_1)(c_2))$

$\vdash$ $\exists c_1{:}Cube\_t.$
$MEM\_ce(c_1)(CONS\_ce(hd)(e_1))\ \&$
$EQc(co)(DEL\_cc(c_1)(c_2))$

Proof Step 9: Right side of second split after membership rewrite

---

This completes our discussion of the proof of Lemma 6, which is used in the proof of Theorem 6.

□

We began this section by describing Brayton and McMullen's division algorithm and then presented our implementation. Based upon the implementation we proved three characterization lemmas

(Lemmas 2, 3 and 4), from which we derived three rewrite rules. We then showed the proof of an abstract property of the quotient function. This proof illustrated many of the proof techniques which were used throughout *Pbs*, and is a good example of the style of reasoning which we employed. The proof began by analyzing the characterization lemmas for the function and then picking a variable to perform induction on. This created a base case and an inductive case. The inductive case was then further refined into two lemmas (Lemma 5 and 6) that were proven individually. The proof of Lemma 6 was described in detail. This proof relied upon the use of characterization lemmas and general mathematical reasoning. Using characterization lemmas allowed us to hide the implementation details of lower level functions which are used in the quotient function. These lower level functions include the function for deleting a cube from an expression, the function for testing if a cube is a member of an expression, and the function for concatenating a cube onto an expression.

In looking back on the quotient function, it is clear that the structure of the function is complex enough that an induction lemma would have been useful. At the time that we were verifying the quotient function, we had not yet discovered induction lemmas, and so proceeded without them. The next section includes a description of the use of an induction lemma to verify a function.

4.3 The Kernels Function – A Complex Example

The most complicated function in *Pbs* is the function to calculate the kernels of an expression. In this section we describe Brayton and McMullen's algorithm for calculating the kernels of an expression, how we implemented this algorithm in *Pbs* and how we verified our implementation using an induction lemma for the kernels function.

4.3.1 Algorithm and Implementation

The goal of multi-level logic synthesis is to eliminate all divisors of two or more expressions in a system of Boolean equations. The most important contribution of Brayton and McMullen's work is an algorithm for efficiently eliminating all divisors. The key to their algorithm is that it is a two pass approach. In the first pass all common sub-expressions of *kernels* are eliminated, and in the second pass all common cubes are eliminated. The proof of correctness of the overall algorithm requires showing that eliminating all common sub-expressions of all kernels followed by all common cubes is equivalent to eliminating all common divisors. The kernels of an expression are the set of all cube free primary divisors of the expression. The primary divisors of an expression are all expressions which can be produced by dividing the expression by a cube. An expression is cube free if no cube divides the expression evenly.

The set of primary divisors of an expression can be calculated by dividing the expression by all possible cubes which will produce non-empty quotients. One of the conditions for ensuring that the quotient of an expression and a cube is non-empty is that all of the literals in the cube appear in the expression. Thus the primary divisors of an expression can be calculated by creating the set of all literals appearing in the expression and then dividing the expression by all possible cubes that can be created from these literals.

Dividing an expression by a cube is equivalent to dividing the expression by the first literal in the cube, dividing the resulting quotient by the second literal in the cube, then dividing that quotient by the third literal, and so on. To generate all of the primary divisors of an expression we can first divide the expression by each of the literals appearing in the expression. This will result in all primary divisors which can be produced by dividing the expression by cubes with one literal. Next, we divide each of these primary divisors by each of the other literals in the expression. This will result in all primary divisors which can be produced by dividing the expression by cubes with two literals. Proceeding in this manner, we can divide the expression by all possible cubes with three literals, then four literals, and so on. Eventually we will have divided the expression by all possible cubes, thus we will have produced the complete set of primary divisors for the expression.

Given the primary divisors of an expression, we can find the kernels by selecting those primary divisors which are cube free. The problem with this approach is that it is not very efficient, because the cube free primary divisors are usually a very small subset of the primary divisors. The algorithm proposed by Brayton and McMullen, and which was used in *Pbs*, generates only the cube free primary divisors. An expression is cube free if no cube can divide the expression evenly; an expression can be made cube free by dividing it by the largest cube which divides the expression evenly.

In creating the implementation of the kernels function we assume that the inputs to the function are a cube free expression and the set of all literals in the expression. The function works by recursing over the list of literals. The general structure of the recursion is shown in Equations 18 and 19, where @ appends two lists together. Equation 18 says that the kernels of an empty list of literals and a cube free expression include just the cube free expression. Equation 19 says that each inductive call to *Kernels* generates two recursive calls. The first call generates the kernels of *tl* and the expression divided by *hd* and then made cube free; the second call generates the kernels of *tl* and the expression.

$$
\begin{aligned}
&\mathit{Kernels}\ \mathit{nil}\ e && = [e] && (18)\\
&\mathit{Kernels}\ (\mathit{hd}::\mathit{tl})\ e && = &&\\
&\quad (\mathit{Kernels}\ \mathit{tl}\ (\mathit{mk\_cube\_free}\ (\mathit{QUOT\_ec}\ e\ (\mathit{CONS\_lc}\ \mathit{hd}\ \mathit{nil})))) \ @\ (\mathit{Kernels}\ \mathit{tl}\ e) && && (19)
\end{aligned}
$$

We can write a preliminary implementation of this function as shown in Figure 7, where the function *mk_cube_free* takes an expression and makes it cube free by dividing it by the largest cube which divides the expression evenly.

We could have written this function more directly if we had allowed primitive recursive functions. As discussed in Section 2.1 we chose not to do this because it would have complicated our embedding of the subset of SML in Nuprl. In looking at the implementation of *Kernels*, we may begin to doubt the wisdom of our choice. But, there are a number of reasons to justify our choice. First, *Kernels* is the only function which required this rather convoluted implementation, so the small amount of extra effort to deal with this one function was worth the benefit of having a simple embedding of

```
fun Kernels ls e =
    let
        fun f hd tl result =
            fn e =>
                result(mk_cube_free (QUOT_ec e (CONS_lc hd NIL_c))) @ result(e)
    in
        (recurse f (fn e => [e]) ls)(e)
    end
```

Figure 7: First implementation of Kernels

SML in Nuprl. Second, once we had implemented *Kernels*, we wrote characterization lemmas for the function. These lemmas were easy to prove and hid the implementation of the function. Thus we did not complicate the rest of our verification effort by choosing to implement *Kernels* in this fashion.

The function works by recursing over the list of literals (*ls*). When this list is empty, *recurse* returns it's second parameter. In this case, that parameter is *fn e* $\Rightarrow$ [*e*]. This is shown in Equation 20, which matches Equation 18. When the list of literals is non-empty, we wish the behavior of the function to match that described in Equation 19. Equation 21 shows the evaluation of the function from Figure 7 on a non-empty list of literals; it matches Equation 19.

$$
\begin{aligned}
\mathit{Kernels\ nil\ e} \quad &= (\mathit{recurse\ f}\ (\mathit{fn\ e}\ \Rightarrow [e])\ \mathit{nil})(e) \\
&= (\mathit{fn\ e}\ \Rightarrow [e])(e) \\
&= ([e])
\end{aligned}
\tag{20}
$$

$$
\begin{aligned}
\mathit{Kernels}\ (\mathit{hd{::}tl})\ e \quad &= (\mathit{recurse\ f}\ (\mathit{fn\ e}\ \Rightarrow [e])\ (\mathit{hd{::}tl}))(e) \\
&= (\mathit{f\ hd\ tl}\ (\mathit{recurse\ f}\ (\mathit{fn\ e}\ \Rightarrow [e])\ \mathit{tl}))(e) \\
&= (\mathit{fn\ e}\ \Rightarrow \\
&\qquad\quad ((\mathit{recurse\ f}\ (\mathit{fn\ e}\ \Rightarrow [e])\ \mathit{tl}) \\
&\qquad\qquad (\mathit{mk\_cube\_free}\ (\mathit{QUOT\_ec\ e}\ (\mathit{CONS\_lc\ hd\ NIL\_c})))) \ @ \\
&\qquad\quad (\mathit{recurse\ f}\ (\mathit{fn\ e}\ \Rightarrow [e])\ \mathit{tl})\ (e))(e) \\
&= ((\mathit{recurse\ f}\ (\mathit{fn\ e}\ \Rightarrow [e])\ \mathit{tl}) \\
&\qquad\quad (\mathit{mk\_cube\_free}\ (\mathit{QUOT\_ec\ e}\ (\mathit{CONS\_lc\ hd\ NIL\_c})))) \ @ \\
&\quad\ (\mathit{recurse\ f}\ (\mathit{fn\ e}\ \Rightarrow [e])\ \mathit{tl})\ (e) \\
&= \mathit{Kernels}\ (\mathit{mk\_cube\_free}\ (\mathit{QUOT\_ec\ e}\ (\mathit{CONS\_lc\ hd\ NIL\_c}))) \ @ \\
&\quad\ \mathit{Kernels\ tl\ e}
\end{aligned}
\tag{21}
$$

The implementation of *Kernels* shown in Figure 7 has the desired behavior, but is not very efficient. The first observation is that if the literal *hd* is not a member of any cube in *e*, then dividing *e* by *hd* will result in an empty expression. Furthermore, if *hd* is not a member of at least two cubes in

e, then making the quotient of e and hd cube free will result in an empty expression. So, if hd is not a member of at least two cubes in e, then there will not be any kernels in the quotient of e and hd. We can test for this, and return an empty list right away if hd is not a member of at least two cubes in e. This is shown in an improved implementation of *Kernels* in Figure 8.

```
fun Kernels ls e =
    let
        fun f hd tl result =
            fn e ⇒
                ((if 2 ≤ |filter (MEM_lc hd) (E2CS e)|
                  then result(mk_cube_free (QUOT_ec e (CONS_lc (hd::nil))))
                  else nil)
                @ result(e))
    in
        (recurse f (fn e ⇒ [e]) ls)(e)
    end
```

Figure 8: Second implementation of Kernels

The *if-then-else* statement in Figure 8 tests if the head of list of literals (hd) occurs in at least two cubes of the expression (e). If this is not the case, then f returns *nil*. This is reflected in the behavior of *Kernels* as shown in Equations 22 and 23.

$$\begin{aligned} &2 \leq |\mathit{filter}\ (\mathit{MEM\_lc}\ hd)\ (\mathit{E2Cs}\ e)| \Rightarrow \\ &\quad \mathit{Kernels}\ (hd{::}tl)\ e = \mathit{Kernels}\ (\mathit{mk\_cube\_free}\ (\mathit{QUOT\_ec}\ e\ (\mathit{CONS\_lc}\ hd\ \mathit{NIL\_c})))\ @ \\ &\qquad\qquad\qquad\qquad\quad \mathit{Kernels}\ tl\ e \end{aligned} \tag{22}$$

$$\begin{aligned} &2 \not\leq |\mathit{filter}\ (\mathit{MEM\_lc}\ hd)\ (\mathit{E2Cs}\ e)| \Rightarrow \\ &\quad \mathit{Kernels}\ (hd{::}tl)\ e = \mathit{Kernels}\ tl\ e \end{aligned} \tag{23}$$

The second observation that we can use to make the implementation of *Kernels* more efficient is that the implementation in Figure 8 generates redundant primary divisors. To see this, we can rewrite the code in Figure 8 as shown in Figure 9. The difference is that in Figure 9, the code for *mk_cube_free* has been expanded out.

The problem here is that the cube which e is divided by will not be unique in each case. This is because the cube consists of hd and the largest cube which divides $QUOT\_ec\ e\ (CONS\_lc\ hd\ NIL\_c)$ evenly. To take of this problem, we can create a variable *lgst_cube* (Equation 24) for the largest cube dividing e evenly and then test if the literals in *lgst_cube* are in the list of literals *tl*. If all of the literals are in that list, then none of them will have been used as a head element yet, therefore the cube will not have been used yet. This is because in order for the cube to have been used

```
fun Kernels ls e =
    let
        fun f hd tl result =
            fn e =>
                (if 2 ≤ |filter (MEM_lc hd) (E2Cs e)|
                then
                    let
                        val lgst_cube = Lgst_cube (QUOT_ec e (CONS_lc hd NIL_c))
                    in
                        result (QUOT_ec e CONS_lc(hd::lgst_cube))
                    end
                else nil)
                @ result(e)
    in
        (recurse f (fn e => [e]) ls)(e)
    end
```

Figure 9: Third implementation of Kernels

already, one of the literals in the cube would have to have been the head element of the cube. This optimization will guarantee that each cube divided into e is unique. We can capture this desired behavior by modifying Equation 22 to test if the literals in *lgst_cube* are in the list of literals *tl*. This is reflected in the behavior of *Kernels* as shown in Equations 26 and 25.

$$lgst\_cube = Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ NIL\_c)) \tag{24}$$

$$\begin{aligned}&2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)|\ \&\\ &(Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ NIL\_c))) \not\subseteq tl \Rightarrow\\ &\quad Kernels\ (hd{::}tl)\ e = Kernels\ tl\ e\end{aligned} \tag{25}$$

$$\begin{aligned}&2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)|\ \&\\ &(Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ NIL\_c))) \subseteq tl \Rightarrow\\ &\quad Kernels\ (hd{::}tl)\ e =\\ &\qquad Kernels\ (mk\_cube\_free\ (QUOT\_ec\ e\ (CONS\_lc\ hd\ NIL\_c)))\ @\\ &\qquad Kernels\ tl\ e\end{aligned} \tag{26}$$

Figure 10 shows the final implementation of *Kernels*, which uses this optimization to ensure that all cubes divided into the expression are unique.

Now that we have a final implementation of *Kernels*, we need to write characterization lemmas for the function. The function recurses over the list of literals *ls*, so we will need one characterization

```
fun Kernels ls e =
let
    fun f hd tl result =
        fn e =>
            let
                val cs = E2Cs(e)
            in
                if 2 ≤ |filter (MEM_lc hd) cs|
                then
                    let
                        val lgst_cube = Lgst_cube(QUOT_ec e (CONS_lc hd nil))
                    in
                        if List_in_list( eq_lit (C2Ls( lgst_cube )) tl )
                        then result(QUOT_ec e (CONS_lc hd lgst_cube))
                        else nil
                    end
                else nil
            end
            @result(e)
in
    (recurse f (fn e => [e]) ls)(e)
end
```

Figure 10: Implementation of kernels function

lemma for the base case (an empty list of literals) and several lemmas for the inductive case. Lemma 7, based on Equation 18 describes the behavior of *Kernels* when the list of literals is empty. For the inductive case we have described the behavior of the function using Equations 23, 25 and 26. In Equations 23 and 25, *Kernels* (*hd*::*tl*) *e* evaluates to *Kernels tl e*. This behavior is described in Lemma 8. The full induction step is described in Lemma 9, which contains two calls to *Kernels* and is based on Equation 26.

Lemma 7 *Kernels base case*

$\vdash \forall e{:}Expr\_t.\ Kernels(nil)(e) = e$

The implementation of *Kernels* as shown in Figure 10 is obviously confusing and difficult to understand. The benefit of using characterization lemmas to hide implementation details of functions becomes very apparent when comparing Lemmas 7, 8 and 9 with the code in Figure 10. These three lemmas clearly and accurately capture the desired behavior of the function. When writing

Lemma 8 *Kernels optimized inductive case*

$$
\begin{array}{l}
\vdash \forall hd{:}Lit\_t. \\
\quad \forall ls{:}Lit\_t\ list. \\
\quad\quad \forall e{:}Expr\_t. \\
\quad\quad\quad cube\_free(e) \Rightarrow \\
\quad\quad\quad\quad |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)| < 2\ \vee \\
\quad\quad\quad\quad\quad 2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)|\ \& \\
\quad\quad\quad\quad\quad C2Ls(\ Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ nil)))\ \not\subseteq ls\ \Rightarrow \\
\quad\quad\quad\quad\quad\quad Kernels(hd{::}ls)(e)\ = Kernels(ls)(e)
\end{array}
$$

Lemma 9 *Kernels full inductive case*

$$
\begin{array}{l}
\vdash \forall hd{:}Lit\_t. \\
\quad \forall ls{:}Lit\_t\ list. \\
\quad\quad \forall e{:}Expr\_t. \\
\quad\quad\quad cube\_free(e) \Rightarrow \\
\quad\quad\quad\quad 2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)|\ \& \\
\quad\quad\quad\quad C2Ls(Lgst\_cube(\ QUOT\_ec(e\ (CONS\_lc\ hd\ NIL\_c))))\ \subseteq ls\ \Rightarrow \\
\quad\quad\quad\quad\quad Kernels(hd{::}ls)(e) = \\
\quad\quad\quad\quad\quad\quad Kernels\ (mk\_cube\_free\ (QUOT\_ec\ e\ (CONS\_lc\ hd\ NIL\_c)))\ @ \\
\quad\quad\quad\quad\quad\quad Kernels\ tl\ e
\end{array}
$$

theorems describing properties of the function, we do not have to look at the implementation; we just have to use these three lemmas. In this sense, the lemmas can also act as documentation for the function, because they describe the behavior at a level which is more abstract than the actual implementation, but more concrete than the overall specification for the function.

4.3.2 Verification of Kernels Function

The implementation of *Kernels* has been optimized in a number of ways which make the code more efficient, but also more complicated. In addition *Kernels* is a doubly recursive function – each call to *Kernels* can generate two sub-calls. By using rewrite rules based on Lemmas 7, 8 and 9 to hide the implementation of *Kernels*, we have removed the added complication of the fact that *Kernels* is implemented in a convoluted fashion in terms of *recurse*, but we have not eliminated the complications caused by the optimizations to the function. If we had not optimized the implementation, we would have had only two lemmas, which would have resembled Equations 18 and 19. Instead, we now have three lemmas which describe four different conditions of the inputs. The added complexity in the verification process is acceptable, because the improvement in the efficiency of the implementation is very significant.

The generation of the kernels is the key to the weak division algorithm. At the highest level, the proofs of the weak division algorithm and of *Pbs* show that removing all common subexpressions of kernels from a system of equations followed by removing all common cubes is equivalent to removing all common divisors. In order to prove this theorem, we had to prove a number of theorems about the kernels function. These theorems describe abstract behavioral properties of the function; they are very far removed from the details of the implementation. An example of one of these theorems is shown in Theorem 7, which says that if k is a cube free expression, and there exists a cube c such that k is equal to some expression e divided by c and all of the literals in c are in the list ls then k is a member of the kernels of the list of literals ls and e.

Theorem 7 *Example Theorem About Kernels*

$$
\begin{array}{l}
\vdash \forall ls{:}Lit\_t\ list.\\
\quad \forall e{:}Expr\_t.\\
\quad\quad cube\_free(e) \Rightarrow\\
\quad\quad\quad \forall k{:}Expr\_t.\\
\quad\quad\quad\quad \exists c : Cube\_t\\
\quad\quad\quad\quad\quad cube\_free\ k\ \&\\
\quad\quad\quad\quad\quad EQe\ k\ (QUOT\_ec\ e\ c)\ \&\\
\quad\quad\quad\quad\quad (C2Ls\ c) \subseteq ls \Rightarrow\\
\quad\quad\quad\quad\quad\quad k \in Kernels\ ls\ e
\end{array}
$$

Almost all of the abstract theorems that we proved about *Kernels*, including Theorem 7, were written in the form shown in Theorem 8. While proving these theorems we realized that much of the reasoning required for these proofs was independent of the predicate $\mathcal{P}$. To reduce the duplication of work, we wrote an induction lemma to capture the reasoning which was common to all of these proofs.

Theorem 8 *Generic theorem about kernels*

$$
\begin{array}{l}
\vdash \forall ls{:}Lit\_t\ list.\\
\quad \forall e{:}Expr\_t.\\
\quad\quad cube\_free(e) \Rightarrow\\
\quad\quad\quad \forall k{:}Expr\_t.\\
\quad\quad\quad\quad \mathcal{P}(ls)(e)(k) \Rightarrow\\
\quad\quad\quad\quad\quad k \in Kernels\ ls\ e
\end{array}
$$

The purpose of induction lemmas is to break complex proof goals down into multiple subgoals, each of which is simpler than the original. The kernels induction lemma was derived directly from Lemmas 7, 8 and 9, which describe the behavior of the kernels function as a base case and two inductive cases. The induction lemma says that if some property holds for all three cases, then it will hold for all inputs to the function. We will describe the lemma in three steps, by writing one clause of the induction lemma for each of the three characterization lemmas. For the base case (Lemma 7) we can write the first clause of the induction lemma as shown in Figure 11.

$$
\begin{array}{l}
\forall e{:}\mathit{Expr\_t}. \\
\quad \mathit{cube\_free}(e) \;\Rightarrow \\
\qquad \forall k{:}\mathit{Expr\_t}. \\
\qquad\quad \mathcal{P}(\mathit{nil})(e)(k) \;\Rightarrow\; \mathit{EQe}\ k\ e
\end{array}
$$

Figure 11: First clause of induction lemma for kernels function

Lemma 8 says that if there are less than two cubes in e or if there are at least two cubes in the expression and the literals in the largest cube dividing the quotient of e and hd are not a subset of the remaining literals in ls, then the optimizations apply and $\mathit{Kernels}(hd{::}ls)(e)$ evaluates to $\mathit{Kernels}(ls)(e)$. This case is reflected in the second clause of the induction lemma, which is shown in Figure 12.

$$
\begin{array}{l}
\forall hd{:}\mathit{Lit\_t}. \\
\quad \forall ls{:}\mathit{Lit\_t\ list}. \\
\qquad \forall e{:}\mathit{Expr\_t}. \\
\qquad\quad \mathit{cube\_free}(e) \;\Rightarrow \\
\qquad\qquad |\mathit{filter}\ (\mathit{MEM\_lc}\ hd)\ (\mathit{E2Cs}\ e)| < 2 \;\vee \\
\qquad\qquad\quad 2 \le |\mathit{filter}\ (\mathit{MEM\_lc}\ hd)\ (\mathit{E2Cs}\ e)| \;\& \\
\qquad\qquad\quad \mathit{C2Ls}(\ \mathit{Lgst\_cube}(\mathit{QUOT\_ec}\ e\ (\mathit{CONS\_lc}\ hd\ \mathit{NIL\_c})))\ \not\subseteq ls \;\Rightarrow \\
\qquad\qquad\qquad \forall k{:}\mathit{Expr\_t}. \\
\qquad\qquad\qquad\quad \mathcal{P}(hd{::}ls)(e)(k) \;\Rightarrow \\
\qquad\qquad\qquad\qquad \mathcal{P}(ls)(e)(k)
\end{array}
$$

Figure 12: Second clause of induction lemma for kernels

When neither of the two above conditions is true, then the evaluation of $\mathit{Kernels}(hd{::}ls)(e)$ contains two calls to *Kernels*, as was shown in Lemma 9. This case makes up the third clause of the induction lemma and is shown in Figure 13.

$$
\begin{array}{l}
(\forall hd{:}Lit\_t. \\
\quad \forall ls{:}Lit\_t\ list. \\
\quad\quad \forall e{:}Expr\_t. \\
\quad\quad\quad cube\_free(e) \Rightarrow \\
\quad\quad\quad\quad 2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)|\ \& \\
\quad\quad\quad\quad C2Ls(\ Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ nil)))\ \subseteq ls \Rightarrow \\
\quad\quad\quad \forall k{:}Expr\_t. \\
\quad\quad\quad\quad \mathcal{P}(hd{::}ls)(e)(k) \Rightarrow \\
\quad\quad\quad\quad\quad \mathcal{P}(ls) \\
\quad\quad\quad\quad\quad\quad (QUOT\_ec \\
\quad\quad\quad\quad\quad\quad\quad e \\
\quad\quad\quad\quad\quad\quad\quad (CONS\_ce\ hd\ (Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ NIL\_c)))) \\
\quad\quad\quad\quad\quad\quad) \\
\quad\quad\quad\quad\quad\quad k \\
\quad\quad\quad\quad\quad \vee \\
\quad\quad\quad\quad\quad\quad \mathcal{P}(ls)(e)(k)
\end{array}
$$

Figure 13: Third clause of induction lemma for kernels

The induction lemma (Lemma 10) states that if some proposition $\mathcal{P}$ is true in all three cases, then it will be true for any inputs to the kernels function. The three induction clauses of Lemma 10 are shown in the order that they were presented here (Figures 11, 12 and 13); the fourth clause states that the predicate will hold for all possible inputs.

Lemma 10 *Kernels induction*

$$
\begin{array}{l}
\vdash \forall \mathcal{P}{:}Lit\_t\ list \to Expr\_t \to Expr\_t \to U_1 .\\
\quad (\forall e{:}Expr\_t.\\
\qquad cube\_free(e) \Rightarrow\\
\qquad\quad \forall k{:}Expr\_t.\\
\qquad\qquad \mathcal{P}(nil)(e)(k) \Rightarrow EQe\ k\ e\\
\quad) \Rightarrow\\
\qquad (\forall hd{:}Lit\_t.\\
\qquad\quad \forall ls{:}Lit\_t\ list.\\
\qquad\qquad \forall e{:}Expr\_t.\\
\qquad\qquad\quad cube\_free(e) \Rightarrow\\
\qquad\qquad\qquad |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)| < 2\ \vee\\
\qquad\qquad\qquad\quad 2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)|\ \&\\
\qquad\qquad\qquad\quad C2Ls(Lgst\_cube(QUOT\_ec(e)((CONS\_lc\ hd\ NIL\_c)))) \not\subseteq ls \Rightarrow\\
\qquad\qquad \forall k{:}Expr\_t.\\
\qquad\qquad\quad \mathcal{P}(hd{::}ls)(e)(k) \Rightarrow\\
\qquad\qquad\qquad \mathcal{P}(ls)(e)(k)\\
\qquad) \Rightarrow\\
(\forall hd{:}Lit\_t.\\
\quad \forall ls{:}Lit\_t\ list.\\
\qquad \forall e{:}Expr\_t.\\
\qquad\quad cube\_free(e) \Rightarrow\\
\qquad\qquad 2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)|\ \&\\
\qquad\qquad C2Ls(\ Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ nil))) \subseteq ls \Rightarrow\\
\qquad\quad \forall k{:}Expr\_t.\\
\qquad\qquad (\mathcal{P}(hd{::}ls)(e)(k) \Rightarrow\\
\qquad\qquad\quad (\mathcal{P}(ls)\\
\qquad\qquad\qquad (QUOT\_ec\ e\\
\qquad\qquad\qquad\quad (CONS\_lc\ hd\\
\qquad\qquad\qquad\qquad (Lgst\_cube(QUOT\_ec\ e\ (CONS\_lc\ hd\ nil)))\\
\qquad\qquad\qquad\quad))\\
\qquad\qquad\quad (k)\ \vee\\
\qquad\qquad \mathcal{P}(ls)(e)(k)\\
\qquad\qquad)\\
\qquad)\\
\quad) \Rightarrow\\
\qquad \forall ls{:}Lit\_t\ list.\\
\qquad\quad \forall e{:}Expr\_t.\\
\qquad\qquad cube\_free(e) \Rightarrow\\
\qquad\qquad\quad \forall k{:}Expr\_t.\\
\qquad\qquad\qquad \mathcal{P}(ls)(e)(k) \Rightarrow k \in Kernels\ ls\ e
\end{array}
$$

$\vdash\ \forall ls{:}Lit\_t\ list.$
$\quad\forall e{:}Expr\_t.$
$\quad\quad cube\_free(e)\ \Rightarrow$
$\quad\quad\quad\forall k{:}Expr\_t.$
$\quad\quad\quad\quad\mathcal{P}(ls)(e)(k)\ \Rightarrow\ k{\in}Kernels\ ls\ e$

Proof Step 10: Proof goal before application of kernels induction lemma

---

$.\ \mathcal{P}(nil)(e)(k)$

$\vdash\ EQe\ k\ e$

$.\ |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)| < 2\ \vee$
$\quad 2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)\ \&$
$\quad C2Ls(Lgst\_cube(QUOT\_ec(e\ (CONS\_lc\ hd\ NIL\_c))))\ \not\subseteq ls$

$.\ \mathcal{P}(hd{::}ls)(e)(k)$

$\vdash\ \mathcal{P}(ls)(e)(k)$

$.\ 2 \leq |filter\ (MEM\_lc\ hd)\ (E2Cs\ e)\ \&$
$\quad C2Ls(Lgst\_cube(QUOT\_ec(e\ (CONS\_lc\ hd\ NIL\_c))))\ \subseteq ls$

$.\ \mathcal{P}(hd{::}ls)(e)(k)$

$\vdash\ \mathcal{P}(ls)$
$\quad(QUOT\_ec\ e$
$\quad\quad(CONS\_lc\ hd\ (Lgst\_cube(QUOT\_ec(e\ (CONS\_lc\ hd\ NIL\_c))))\ )$
$\quad)$
$\quad k$
$\vee$
$\mathcal{P}(ls)(e)(k)$

Proof Step 11: Proof goals after application of kernels induction lemma

---

Applying Lemma 10 in a proof will have the affect of breaking a goal such as shown in Proof Step 10 (which is of the same form as Theorem 8) into the three subgoals shown in Proof Step 11. The function *Kernels* does not appear in any of the three subgoals: all of the reasoning in the theorem which was dependent upon *Kernels* has been captured in the induction lemma. After applying the induction lemma the majority of the proof effort is determined by the specific property $\mathcal{P}$ that is being verified. We used this induction lemma in most of the proofs about the kernels function, including Theorem 7.

5 Discussion

In this section we analyze our methodology and provide some general thoughts on the task of formal verification of functional programs. We conclude with some thoughts on the use of proof systems as integral tools in the process of developing software.

5.1 Benefits of Formal Verification

There are many benefits to verifying programs. The most apparent one is that there is increased confidence that the program works correctly. In addition, we found that there were several optimizations that we were able to perform to our implementation of *Pbs* without sacrificing verifiability. In fact, without formally verifying the code, we would have been reluctant to make some of these optimizations. The general observation to make here is that verifying *Pbs* led to an increased understanding of the algorithm and our implementation, which in turn led to better a implementation.

By examining the code and algorithm in great detail as the work was being formalized, several obscure errors in the implementation and formal description of *Pbs* were found. The nature of most of the errors was such that they would most probably manifest themselves only in rare occurrences in large systems of equations, exactly the times when they would be least likely to be detected. The three most important errors that were found are discussed below.

We originally thought that by using only a very small subset of the kernels (*the level zero kernels*), we could ensure that all intermediate variables would be substituted into at least two functions. This meant that we would not have to check to ensure that all intermediate variables were used at least twice. Avoiding this check produced a great speed up in *Pbs* and yielded results identical to a correct algorithm in most cases. It was not until we began to develop a formal proof of *Pbs* that we discovered that even if we used just the level zero kernels, we could not be sure all intermediate variables would be substituted into at least two functions. To correct this problem we modified our implementation so that we could guarantee that all intermediate variables were used at least twice.

In translating from our original implementation of *Pbs* to the final implementation which was used in the proof, we made a mistake in a function used in calculating the quotient of two expressions. This mistake would have been eventually caught in testing the new version of the code, because the error would affect the calculation of the kernels of every expression. We did not catch the mistake until we began proving a theorem about the function which calculates the kernels of an expression. We had already proved several theorems describing these two functions before we discovered the error. The error was caused by incorrectly understanding the original (and correctly working) function.

The final error that was found in *Pbs*, was not in the implementation, but in a proof that was originally done on paper. When doing the proof that the minimality property for kernels is equivalent to the minimality property for divisors we did not state the theorem correctly and included a false lemma as one step in the proof. When the proof of the theorem was done in Nuprl, this problem was quickly found and easily corrected.

No errors were found in Brayton and McMullen's work, but that does not mean that nothing was gained by developing a formally verified *implementation* of the weak division algorithm. Many of the definitions in Brayton and McMullen's work are informal and incomplete. Their proofs are very brief and describe the algorithm at an abstract level that is very far removed from an actual implementation. Because their presentation of the algorithm and proofs are so brief, there is a great disparity between the level of detail of the proofs and the implementation. It is in this process of going from an abstract description of an algorithm to the concrete implementation, that many errors are introduced. An informal proof that the algorithm is correct is a good first step toward a correct implementation, but unless formal verification is done at the level of implementation, there can not be a high level of assurance that the final implementation is correct.

In writing code and designing hardware, situations frequently arise in which an operation can be optimized if it can be guaranteed that certain properties will hold. For example, for each of the Boolean operations in the weak division algorithm there is a corresponding algebraic operation which is much faster, but returns the correct result only under certain conditions.

In our original implementation of *Pbs*, only Boolean operations were used. This sacrificed speed for increased assurance that the code was correct. In doing the proof in Nuprl, we discovered several instances where we could guarantee that the correctness conditions for the algebraic operations would hold. When these occurrences were found, we substituted the algebraic operation for the Boolean operation. This increased the speed of the code and also simplified the proof, because the lemmas describing the algebraic operations were simpler than those describing the Boolean ones.

In the implementation of the kernels function (Section 4.3), we performed a variety of optimizations which made the implementation more efficient but also more difficult to understand and verify. Through the use of characterization lemmas we were able to concisely describe the behavior of the function in a form which was much more readable than the actual implementation. This made it easier to understand the effect of the optimizations on the behavior of the function, so we were able to verify the optimized function with a reasonable amount of effort.

5.2 Summary and Analysis

Good programming habits are much more important when doing verification than when just programming. There is a complexity barrier that occurs when writing programs after which the user is incapable of maintaining or enhancing the program without adopting proper techniques. When doing verification this barrier is dropped so low that without good programming habits it becomes infeasible to verify anything more than the simplest program.

As the size of a verification effort increases, the style in which the verification is done becomes very important. Just as there is a complexity barrier beyond which it is infeasible to verify poorly written programs, there is a similar barrier beyond which it is infeasible to complete poorly developed proofs of programs.

After implementing each function, we proved a number of characterization lemmas. Each lemma described the behavior of the function on a specific set of inputs. Other proofs then used the characterization lemmas and were not directly dependent upon the the implementation of the function.

Induction lemmas act to structure the proof effort by breaking down a single complex goal into multiple simpler goals. Many times, by using induction lemmas, we were able to take very general theorems about functions and break them down into several simpler subgoals, none of which contained a reference to the original function. Thus, the induction lemma captured all of the reasoning that was specific to the original function. By writing induction lemma we had moved up a level of abstraction; we were reasoning about the *properties* that we wanted to verify about the function, not just the *behavior* of the function.

One technique which did not occur to us until we had completed the proof and were able to analyze our work as a whole, was to generalize the reasoning to general mathematical principles. The operations in *Pbs* can be described as an algebra. There is a large body of existing knowledge about algebras, which we could have used. Had we shown that the operators in *Pbs* were an algebra, we could have used general theorems about algebras to do the more complicated and abstract reasoning in *Pbs*.

5.3 Increasing Efficiency

In the process of implementing *Pbs* and doing the proof in Nuprl there was a large learning curve and several new tools were written to make the proof easier. We estimate that if we were to do it over again, it would take approximately one month to implement the code and an additional two months to complete a formal proof on paper. We believe that using the knowledge gained and tools written, it would take a total of four months to do the proof in Nuprl all over again. Thus, doing the proof in Nuprl would take approximately twice as long as doing the proof on paper.

In order to make proof systems more attractive, we must begin to reduce the disparity between the time required to do a proof on paper and to do the proof using a proof system. One step towards this goal would be to try to automate the generation and verification of characterization and induction lemmas. Because there is such a close correspondence between the characterization lemmas, the induction lemma and the implementation of a function; it should be possible to provide mechanical support that can synthesize and verify characterization lemmas and an induction lemma from the implementation of a function. The HOL proof system provides a number of utilities for automatically creating certain types of definitions. The most recent of these tools takes a set of rules and automatically generates an inductive relation and associated rewrite lemmas defined by the rules [CM92].

In *Pbs*, if the process of selecting and applying rewrite rules and lemmas was automated, then a great number of the characterization and induction lemmas could have been proven automatically and many of the others proofs could have been done significantly faster. Bundy has had some very promising results using the concepts of *proof plans* and *rippling* to guide the selection and application of rewrite rules [BHHS91].

5.4 An Overall Methodology

In a normal software development environment, the programmer begins with a notion of an algorithm for solving a problem. The programmer may sketch out an informal specification and a proof to show that the algorithm will satisfy the specification, but the process of implementing the algorithm usually begins very soon after the algorithm is chosen. If the implementation does not produce the desired results, it could be due to a bug in the implementation, an incorrect algorithm or an incorrect specification. There isn't any technique for quickly and easily determining the source of the problem.

By using a proof system the user can write a formal specification, prove high level theorems to verify the algorithm, and then from these theorems derive a more detailed specification for the implementation, all before writing the implementation. When the specification for the implementation is ready, the programmer may write the implementation and run it on a number of test cases to quickly gain some confidence that the implementation is close to being correct. Once the programmer has confidence that the implementation works correctly for the test cases, the implementation can be verified in the proof system. In the process of verification, errors will most likely be found. These can be fixed while doing the verification, and then the corrected program can be used with a great deal of confidence that it is correct. Having the ability to mix the formal verification and testing of the implementation allows the programmer to be as efficient as possible and use the most appropriate tool for the current task.

This method has the potential to be more efficient than the conventional method and will result in a much better understood algorithm and implementation. Furthermore, the specification and formal descriptions of algorithm and implementation act as documentation for the programmer's intentions. This type of documentation is guaranteed to be correct and unambiguous, something which can not be said for the informal documentation written for most programs.

6 Conclusion

Proof systems are commonly thought of as being used exclusively as tools for proving theorems. In fact they are much more powerful and much more useful than they are commonly perceived. Proof systems are tools for organizing and understanding concepts and objects. They have the ability to manipulate objects as well as reason about properties of objects. While a literate programming tool has the potential to reduce the number of errors in and increase the understanding of programs, these benefits are really just byproducts of its ability to gather, organize and present information to the programmer. It is up to the programmer to interpret the information that the tool provides. In comparison, when a programmer uses a proof system to prove that an implemenation satisfies a specification, the proof system is checking that the behavior of the program matches the behavior specified by the programmer.

In order to make theorem proving based verification more widely used, we need to enhance the productivity of users and make mechanical proof systems less intimidating. The goal should be to make the process of creating a formal proof as easy as creating an informal proof. Only then will formal proof techniques become widely accepted. One step in this process is to create a methodology which users can follow when undertaking large verification efforts. By following a methodology users are freed to concentrate on abstract reasoning, because the details of interacting with the proof system are uniform, consistent, and known ahead of time.

Acknowledgements

We would like to thank Robert Constable, John O'Leary and Wayne Luk for reading earlier drafts of this paper. Also, we are indebted to Paul Jackson for implementing the rewrite package and for his advice and assistance in using Nuprl. In addition we would like to thank Doug Howe and Chet Murthy who were always willing to help and gave us many useful suggestions. Mark Aagaard is supported by a fellowship from the Digital Equipment Corporation. Miriam Leeser is supported by an NSF National Young Investigator Award. This research was supported in part by the National Science Foundation under Award No. MIP-9100516.

References

[Aag92] M. Aagaard. A verified system for logic synthesis. Master's thesis, Department of Electrical Engineering, Cornell University, January 1992.

[AL91] M. Aagaard and M. Leeser. The implementation and proof of a boolean simplification system. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits, Oxford 1990.* Springer-Verlag, 1991.

[AL92] M. Aagaard and M. Leeser. PBS: Proven boolean simplification, 1992. Submitted to the *IEEE Transactions on CAD.*

[BHHS91] A. Bundy, F. Van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324, 1991.

[BM82] R. K. Brayton and C. McMullen. Decomposition and factorization of boolean expressions. In *International Symposium on Circuits and Systems*, 1982.

[BM88] R. S. Boyer and J. S. Moore. *A Computational Logic Handbook.* Academic Press, 1988. Volume 23 of Perspectives in Computing.

[BM889] Journal of automated reasoning, Dec 1989. Special Issue on Boyer-Moore Theorem Prover.

[BR+87] R. K. Brayton, R. Rudell, et al. MIS: A multiple-level logic optimization system. *IEEE Transactions on Computer-Aided Design*, CAD-6(6), 1987.

[C+86] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Chi90] Shiu-Kai Chin. Combining engineering vigor with mathematical rigor. In *Proceedings of the MSI Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects.* Springer Verlag, 1990. LNCS 408.

[CM92] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge, Computer Laboratory, Aug 1992.

[Coh88] A. Cohn. A proof of correctness of the Viper microprocessor: The first level. In *VLSI Specification, Verification, and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1988.

[DK90] M. Dyer and A. Kouchakdjian. Correctness verification: alternative to structural software testing. *Information and Software Technology*, 32(1):53–59, Jan-Feb 1990.

[FST92] S. Fischer, A. Scholz, and D. Tauber. Verification in process algebra of distributed control of track vehicles - a case study. In Gregor Bochmann and David Probst, editors, *Workshop on Computer-Aided Verification.* Springer-Verlag, June 1992.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78. Springer Verlag, 1979. Lecture Notes in Computer Science.

[Hun86] W. A. Hunt, Jr. *FM8501: A Verified Microprocessor.* PhD thesis, Institute for Computing Science, The University of Texasat Austin, 1986.

[LCA+93] Miriam Leeser, Richard Chapman, Mark Aagaard, Mark Linderman, and Stephan Meier. High level synthesis and generating FPGAs with the BEDROC system. *Journal of VLSI Signal Processing*, 1993. To appear.

[LLS90] C. Lafontaine, Y. Ledru, and P-Y Schobbens. Experiment in formal software development: Using the b theorem prover on a vdm case study. In *International Conference on Software Engineering*, pages 34–42, March 1990.

[LM88] R. Linger and H Mills. A case study in cleanroom software engineering: The IBM COBOL structuring facility. In *International Computer Software and Applications Conference*, pages 10–17, October 1988.

[Mar90] A. J. Martin. The design of a delay-insensitive microprocessor: An example of circuit synthesis by program transformation. In *Proceedings of the MSI Workshop on Hardware Specification, Verification, and Synthesis: Mathematical Aspects.* Springer Verlag, 1990. LNCS 408.

[McF91] M. C. McFarland. A practical application of verification to high-level synthesis. In *International Workshop on Formal Methods in VLSI Design.* ACM, 1991.

[RH90] M. Tofte R. Harper, R. Milner. *The Definition of Standard ML*. The MIT Press, 1990.

[TDW+88] D. E. Thomas, E. M. Dirkes, R. A. Walker, J. V. Rajan, J. A. Nestor, and R. L. Blackburn. The system architect's workbench. In 25^{th} *Design Automation Conference*, pages 337–343. ACM/IEEE, 1988.

[Win90] J. Wing. A specifier's introduction to formal methods. *Computer*, 23(9):19–25, Sept 1990.

A Definitions

A.1 Definitions and Symbols from Weak Division Algorithm

$sup(f)$	*Support*	The set of all variables appearing in expression f.
$f * g$	*Product*	The pairwise union of all cubes from two expressions f and g.
	Orthogonal	Two expressions are orthogonal if they do not have any common variables.
f/g	*Divide*	The expression f divided by the expression g is the largest set of cubes h such that: h is orthogonal to g the product of h and g is a subset of f
	Divide evenly	g divides f evenly if $(f/g) * g = f$.
	Cube free	An expression is cube free if no cube divides it evenly.
$\delta(f)$	*Divisors*	The divisors of an expression f are all cubes or expressions which can be produced by dividing an expression by a cube or expression.
$d \mid e$	*Divides*	The expression d divides e if the quotient of e divided by d is non-empty.
$\mathcal{D}(f)$	*Primary Divisors*	Those divisors of f which can be produced by dividing the expression by a cube.
$\mathcal{K}(f)$	*Kernels*	The set of all cube free primary divisors of expression f.
	Remainder	The remainder of dividing an expression f by g is the expression comprised of those cubes which are members of f, but not of f/g.

A.2 Type Definitions

Type	ML name	Definition	Examples
variable	*Var_t*	String of characters	a, wr, c_o
literal	*Lit_t*	Variable or its complement	a, $\neg b$
cube	*Cube_t*	Conjunction of literals	$a \wedge b \wedge c$, $wr \wedge \neg c\_o$
expression	*Expr_t*	Disjunction of cubes	$a \wedge b \wedge c \vee d \wedge \neg e$
function	*Func_t*	Variable associated with an expression	$a = b \wedge c \ \vee \ e \wedge d$
system	*Syst_t*	A set of functions	

A.3 Proof Notation

$\&$	logical and
$\vee$	logical or
$\in$	element of
$\cap$	intersection
$\cup$	union
$\Rightarrow$	implies
$\subseteq$	subset
$\Leftrightarrow$	logical equivalence
$\mapsto$	evaluates to
$\forall x : T.P$	for all x of type T, P is true
$\exists x : T.P$	there exists an x of type T, such that P is true