Tactics from proofs

Thesis

UNRESTRICTED

TACTICS FROM PROOFS

By

Tawanda Gurukumba

BSc (Computer Science and Physics, University of Zimbabwe)

MSc (Computation, University of Oxford)

SUBMITTED FOR THE DEGREE OF

MASTER OF PHILOSOPHY

AT

THE COMPUTING DEPARTMENT

FACULTY OF MATHEMATICS, COMPUTING AND TECHNOLOGY

THE OPEN UNIVERSITY

WALTON HALL, MILTON KEYNES

OCTOBER 2007

DATE OF SUBMISSION 30 MARCH 2007
DATE OF AWARD 24 OCTOBER 2007

Declaration

I declare that the work in this thesis is my own independent contribution. No portion of the material offered in this thesis has previously been submitted by me in support of an application for another degree or qualification to this or any other University, or other institution of learning. The theory of Architectural Retrenchment in Chapter 5 has been previously published by the Open University Computing Department [HG01].

Table of Contents

List of Tables

List of Figures

Abstract

Proof guarantees the correctness of a formal specification with respect to formal requirements, and of an implementation with respect to a specification, and so provides valuable verification methods in high integrity system development. However, proof development by hand tends to be an erudite, error-prone and seemingly interminable task.

Tactics are programs that drive theorem-provers, thus automating proof development and alleviating some of the problems mentioned above. The development of tactics for a particular application domain also extends the domain of application of the theorem-prover. A LCF-tactic is *safe* in that if it fails to be applicable to a particular conjecture, then it will not produce an incorrect proof.

The current construction of tactics from proofs does not yield sufficiently robust tactics. Proofs tend to be specific to the details of a specification and so are not reusable in general, e.g. the same proof may not work when the definition of a conjecture is changed. The major challenges in proof development are deciding which proof rule and instantiations to apply in order to prove a conjecture.

Discerning patterns in formal interactive proof development facilitates the construction of robust tactics that can withstand definitional changes in conjectures. Having developed an interactive proof for a conjecture, we develop the necessary abstractions of the proof steps used, to construct a tactic that can be applicable to other conjectures in that domain. By so doing we encode human expertise used in the proof development, and make proofs robust and thus generally reusable.

We apply our theory on the proofs of conjectures involving some set theory operators, and on the proof obligations that arise in the formal development of numerical specifications using the retrenchment method under the IEEE-854 floating-point standard in the PVS theorem-prover/proof-checker.

Acknowledgements

I am grateful to my supervisory team, Dr Jon Hall and Prof Darrell Ince, and to Rob Griffiths and Dr Marian Petre for their many suggestions and support during this research. Thanks also to The Open University Research School team, the Computing Department and the MCS TBT team for financial and technical software/hardware support respectively required for this research.

My interest in formal methods and critical systems engineering is largely due to my undergraduate and postgraduate studies; thanks in particular to Dr Fortune Mhlanga, Dr Gifford Hapanyengwi, Dr Rosemary Shumba, Dr Andrew Simpson, Prof Jim Woodcock. I am also thankful to Dr Mike Poppleton and Dr Richard Banach for support on the retrenchment method; to Prof Brian Wichmann and Dr Paul Miner for support on floating-point computation; to Dr Ceaser Munoz and Dr David Stringer-Calvert and the SRI team, for useful discussions on the PVS theorem-prover used in this work; Marina Davidson, and to all those others referenced in this work.

Last but not least, thanks to my family, workmates, friends, and others too numerous to mention by name, for their salutary companionship throughout the course of this work.

This thesis was written using Latex2*e* in WinEdit.

Chapter 1

Introduction

Computer systems are expected to deliver a proper service that adheres to their specified requirements. One promising way to specify the requirements of a computer system in an unambiguous manner is to use formal specification languages which are based on Mathematical logic. The requirements specification can then be checked to ensure that it is correct with respect to the formal logic system by the use of formal mathematical proof. This thesis is on the development of computer programs called *tactics* that can be used in order to partially automate the task of developing a formal proof in an Interactive Theorem-Prover/Proof-Checker (ITP/PC).

A proof obligation that can arise from a formal specification of a computer program can be expressed in the form of a conjecture in Mathematical Logic. For such a conjecture, a formal proof can be developed by hand (and/or with the assistance of a proof-checker) in the proof system of the formal logic used—such proofs are called *hand-generated proofs*. Having derived a formal proof of a conjecture, a tactic can then be encoded from the proof by collating the proof-steps used in developing that formal proof. The main goal of our research is to derive the so-called *robust tactics*, i.e. tactics that can still be applicable in developing formal proofs of other expressions similar to the conjectures from which those tactics themselves were derived.

Since the robust tactics are to be used in the formal development of mathematical proofs, the development of such robust tactics themselves should be under a rigorous mathematical basis. In particular, a robust tactic should yield a normal form of a formal proof, which is equivalent to the original formal proof from which the robust tactic was derived. Thus a second goal of this work is to yield a mathematically correct procedure which can be used to derive such robust tactics from proofs.

The third main goal of the research on tactics is to minimise the amount of interaction between the Interactive Theorem-Prover/Proof-Checker (ITP/PC) and its human user in the task of developing formal proofs. The amount of interaction between a human user and the Interactive Theorem-Prover or Proof-Checker can then be taken as a metric for the usefulness of a tactic in this endeavour.

Therefore the topic 'Tactics from Proofs' is associated with two main research questions: (1) Can *robust tactics* be derived from hand-generated proofs?; and (2) Can such tactics be incorporated in a state of the art theorem prover?

1.1 Why formal proof in software development?

Most of the errors incurred in software development are due to logical errors on the part of the programmer [YBH97]. Software simulation and testing are the conventional means of detecting errors in software development. However, it is impossible to create an environment that is a perfect replica of reality when relying on simulation alone to ensure that a system meets its requirements [Ste98]. It is also impossible to exhaustively test software for defects using conventional software testing techniques because there are too many paths, possible inputs, and hardware failure modes in a computer system [Ham89]. In addition, software testing also accounts for over half the cost of a software development project [YBH97].

To aid the process of developing software that meets its specified requirements, some industry standards have mandated and advocated for the use of formal methods in software development, e.g. the railways, defense, and aviation industries [BS92]. However software testing and simulation will always be required to validate those parts of the software that are not amenable to mathematical representation and consequently cannot be formally verified.

1.1.1 Formal methods in software engineering

Formal methods is a collection of software development techniques based on mathematics. The application of mathematical techniques in software development establishes software engineering as a true engineering discipline since other branches of engineering, e.g. electrical, mechanical, and civil engineering disciplines, are based on mathematics. The main issue addressed by formal methods is that of correctness, i.e. "the delivery of a proper service that adheres to specified requirements" [BS92].

The major criticisms of formal methods are that formal methods are: (1) unscalable, i.e. formal methods have been said to apply only to toy academic projects and are not usable on industrial-scale projects; (2) intractable, i.e. formal methods techniques are difficult to reuse; and (3) limited to the mathematical domain, i.e. not everything that is informal can be formalized and conversely [Sim96]. Furthermore, formal methods have been said to take too long to be successfully applied, especially when applied by hand. The judicious application of formal methods, i.e. applying formal methods to only those critical parts of a system, as well as the provision of software tools such as theorem-provers and model-checkers, can go a long way to alleviate these problems [Gri81].

1.2 Software tool support for formal methods

Theorem-provers and model-checkers are two main software tools that are used to automate formal methods. However model-checking inherently suffers from the state explosion problem and thus is restricted to finite domains. Theorem-provers are free from the state-explosion problem since proof techniques such as mathematical induction can be used to argue about very large and/or infinite domains. However full automatic theorem-proving cannot be achieved due to the undecidability problem in First-order and Higher-order logics, which are the logics of choice for specifying properties of computer systems. As such, Automatic Theorem-Provers (APTs) often fail to prove putative theorems of computer specifications.

In Interactive Theorem-Proving/Proof-Checking (ITP/PC), the theorem-prover acts as a 'slave' in performing automated tasks for a 'master' expert human-prover/user, who guides the proof development process. Since humans are better at making strategic decisions and computers are better at mechanical processing, ITPs/PCs tend to be more effective than ATPs, and often succeed where ATPs fail. However, the user has to be an expert in mathematical proof development techniques, as well as in the language and proof system of the ITP/PC being used. In addition, the proof system of most ITP/PCs consist of the most basic proof rules of the proof system, e.g. the proof system of a ITP/PC based on classical logic may only consist of the classical logic rules of inference. The formal specification and verification of a system can thus demand extensive man-hours for even the most simple systems.

The partial automation of the proof development using tactics can go a long way to reduce the level of expertise and man-hours required to specify and verify a computer system in ITP/PCs. The use of such software tools partially increases the efficiency of formal proof development since tedious tasks may be automated, thus enabling faster processing by computer. The incidence of random or human error is also significantly reduced since computers can faithfully and securely perform mechanical tasks. However care still needs to be taken that the software tools themselves are correct; otherwise systematic errors would occur when the software tools are used.

1.3 Tactics from proofs

Proof development by hand tends to be an erudite, error-prone and seemingly interminable task. Tactics are programs that drive theorem-provers by automating the basic tasks in proof development, and thus tactics provide a means of alleviating these problems. The development of tactics for a particular application domain also extends the domain of use of the theorem-prover. A tactic is *safe* in that if a tactic fails to be applicable to a particular conjecture, it will not produce an incorrect proof.

The current construction of tactics from proofs does not yield sufficiently robust tactics. In the development of a formal proof, the major challenges are deciding which proof rule and variable instantiations to apply in order to prove a conjecture. Therefore proofs tend to be specific to the details of a specification and so are not reusable in general, e.g. the same proof may not work when the definition of the conjecture is changed.

Our solution to developing robust tactics is to discover patterns in a proof development in some domain, and then to encode these patterns as a tactic. In particular, the proofsteps used in the proof development are classified and rearranged to yield a proof in a normal form. This is done incrementally, i.e. when a tactic from one proof development fails on a different conjecture, a hand proof is developed for that conjecture; a new tactic is developed from that proof; and the new tactic is composed with the existent tactics, thus yielding a more powerful or robust tactic, which is reusable in different proof development exercises. When the proofs are developed using an ITP/PC, the human expertise used in the proof development can also be encoded in the tactic, e.g. as formal parameters of the tactic.

The main contribution of this thesis is the development of an algorithm that can be used to construct such robust tactics from proofs. Another of our contributions is to use this algorithm to develop tactics that can be used to verify retrenchment proof obligations and thus paving the way towards its mechanization. The Retrenchment method for software development has been developed to argue about the correctness of computer hybrid systems.

1.4 Hybrid systems

Hybrid systems are "critical systems composed of continuous components, which are controlled and supervised by digital components" [NASa], e.g. control systems. This has led to an integration of the continuous and the discrete computation models, i.e. real computations and floating-point computations respectively.

The liberalization of the transformational form of the stepwise refinement method yielded the Retrenchment method, which is a formal method to specify and reason about hybrid systems [Ban98, BP99b, BP99a, PB02, Pop01]. The retrenchment method has been formalized in the B-Method for formal software development [Abr96, Abr98a, BCo02, Wor96]. However, the B-Method does not adequately formalize continuous mathematics which is based on the real number datatype. Therefore a theorem-prover/proof-checker, such as PVS, which supports the formalization of the reals, and the formalization of digital system computation as floats, as well as the encoding of tactics, is more appropriate as a software tool for the retrenchment method.

1.5 Organization of the thesis

This thesis makes use of standard mathematical and computer science notation, e.g. the symbol □ is used to signal the end of a definition, or a theorem. Teletype text denotes machine readable code, e.g. PVS specification text, `U0:TYPE = [# a0:real, b0:real #]`, or user input `(grind)`.

The state of the art in Critical Systems, Formal Methods, Tool Support, Mathematical Logic, and the current methods of constructing tactics from proofs are reviewed in Chapter 2. The nature of the robust tactics to be developed, and the research questions and goals pertaining to this task are outlined.

Based on the original LCF idea [GMW79, Mil84], Chapter 3 presents our theory for deriving robust LCF-like tactics from proofs of similar conjectures. The theory is based on the classification and permutation of proof-steps in a hand-generated proof of a conjecture, as well as the ability to extend one robust tactic with another. For the tactics produced by this theory, arguments are given for their mathematical integrity, robustness or reuse, and ability to reduce user-computer interaction in Interactive Theorem-Proving and Proof-Checking. The procedure is applied on some example simple similar conjectures.

Chapter 4 describes how the theory derived in Chapter 3 can be carried out in the chosen state of the art Interactive Theorem-Prover and Proof-Checker, PVS. PVS proof commands (defined-rules) are analogous to LCF-tactics and PVS strategies are analogous to LCF-tacticals [SORSC98c]. However the most powerful tactic `(grind)` in PVS is found unsafe in that it can sometimes invoke an incorrect instantiation which results in an incorrect proofstate. Therefore `(grind)` is made more robust according to the theory of Chapter 3. The encoding of such robust tactics in PVS is described for the simple similar example conjectures in Chapter 3.

Chapter 5 introduces the Architectural Retrenchment method [HG01] for the specification and proof of the vanilla Banach-Poppleton retrenchment method [Pop01, BP99b, PB02]. This decomposes a retrenchment specification into three separate architectural retrenchments; and the operational retrenchment proof obligation into a subrefinement or a concession. The change in variables from state to input and output variables, and the Operational Retrenchment proof obligation are made more transparent. PVS specification templates are formulated for Architectural retrenchment, and the robust `(grind)` tactic from Chapter 4 is extended in order to prove all the retrenchment proof obligations automatically for a given example retrenchment taken from [Pop01].

Chapter 6 tackles the task of formulating tactics for reliable numerical computation using a form of Architectural Evolving Retrenchment incorporating a specification of the IEEE_854 Floating Point Standard in PVS. This involves a change in datatype from reals to floats and therefore a change in complexity in the Architectural Retrenchment specifications as given in Chapter 5. The specification templates and robust tactics from Chapter 5 are extended to yield a robust tactic (`FOpRetTac` *fnum terms*) which proves all the retrenchment proof obligations for the exact and inexact representation of reals by floats, and error propagation in floating-point operations according to floating-point theory [Gol91].

Chapter 7 discusses the main contributions of this thesis, which include (1) a mathematically correct procedure for constructing robust tactics from proofs, (2) the Architectural Retrenchment method, and (3) specification and proof using the Architectural Retrenchment method in PVS. An inference of this work is that our specification templates and robust tactics constitute a specification and proof system for Architectural retrenchment. Chapter 8 concludes with how our work achieves the goals and subgoals of the research, and outlines some possible future work on (1) our tactic construction method; and (2) the automation of the retrenchment method.

The references used in this work are listed in the References section. Appendix A lists the proof rules used in this work and their permutation analysis. Appendix B contains a sample proof using a robust tactic, the Banach-Poppleton retrenchment and their corresponding PVS specification templates for Architectural Retrenchment.

Chapter 2

Tactics in formal methods

"The search for a proof of a conjecture expressed in some formal language is strikingly similar to many goal-seeking activities ... In general then we may express tactics as partial functions from goals to lists of goals." [Mil84].

This chapter reviews some background information and the state of the art on tactics in formal methods. The purpose is to identify and evaluate the existing problem(s) in the construction of robust tactics from proofs. From this discussion, the major research questions are formulated.

2.1 Introduction

Computer systems are expected to be correct, i.e. to deliver a proper service that satisfies specified requirements. Correctness is particularly important for critical computer systems, which are becoming increasingly used in situations where a malfunction could lead to catastrophe. It is infeasible to demonstrate correctness through traditional testing and/or simulation alone [LS93](Section 2.2).

Most errors incurred in software development, which may lead to the software malfunctioning in operation, are logical errors [YBH97]. Formal Methods (Section 2.3) address correctness by employing Formal Mathematical logic (Section 2.4) to specify and verify properties of computer systems. However, Formal Methods tend to be an erudite, error-prone, seemingly interminable, unscalable and intractable to apply successfully. The study of formal mathematical systems, i.e. Proof Theory (Section 2.5) can help develop better formal methods. The use of Software Tool Support (Section 2.3.2) as an aid in the study and application of Formal methods can also greatly improve the tractability and scalability of Formal Methods, as well as reduce the incidence of human random error. However the user is expected to be an expert in both the Software Technology used and Mathematical logic.

LCF-tactics (Sections 2.6, 2.7) are software programs based on a rigorous mathematical framework, that partially automate the task of formal proof development in interactive theorem-proving and proof-checking. Since the tactics are intended for use in a formal method, the tactics need to be based on formal logic rather than heuristic or plausible techniques (see Section 2.8). A major desirable is to derive such robust tactics based on a rigorous formal mathematical framework, which can be reused to develop proofs with minimal human assistance despite modest system changes (Section 2.9.1). Such robust tactics when incorporated in an Interactive Theorem-Prover (ITP/PC), can also minimize the amount of user-computer interaction in a proof development exercise, as well as the level of domain-knowledge required of the user ITP/PC.

The major research questions to be addressed in this work are: (1) Can robust tactics be derived from hand generated proofs?; and (2) Can such tactics be incorporated in a state of the art theorem-prover?. The main goals of our research are developing a method for: (1) deriving robust or reusable tactics with a rigorous mathematical integrity; (2) encoding such robust tactics in a state of the art theorem prover; and (3) demonstrating development of such tactics on a formal software development method, and the usefulness of such tactics in reducing the user-computer interaction (via automation) and the user-knowledge in formal specification (via specification templates) and proof (via tactic reuse).

2.2 The state of the art in systems development

The specified requirements of a computer system can be classified into two types [DB82]: (1) *functional requirements* which specify the internal workings (i.e. specific behaviour) of the system, e.g. the behaviour of state and input/output variables (see Chapters 5, 6); and (2) *non-functional* requirements which specify overall characteristics of the system, e.g. critical system properties such as reliability, safety, security, and real-timeliness.

A computer system is taken to consist of the following components [Abr98b]: (1) the software program, i.e. the specified functional requirements expressed in a programming language; (2) the operational environment, i.e. the entities which interact with the software program, e.g. the compiler which automatically generates machine code (for the hardware) from the software program; and (3) hardware, i.e. the physical entities which execute the machine code (from the compiler), e.g. the processor type of the computer which runs the software program. From such a system, the functional and non-functional requirements can then be investigated.

Critical systems must satisfy their functional requirements and must be seen to satisfy their non-functional requirements, e.g. safety-critical systems must be seen to satisfy the ultra-critical range of 10^{-7}–10^{-12} failures per hour, i.e. at most one failure in 1141 to 114 155 251 years! [Rus94]. It is infeasible to show through testing and/or simulation that safety-critical systems—whose failure could result in catastrophe—meet such ultra-high reliability requirements [BF91]. However, if the formal specification of a software system is provably correct, then this gives confidence that the software system satisfies its requirements. Hence the analysis of safety-critical systems has been classified to fall in the deductive reasoning domain, which preserves truth [Lev86].

2.3 Formal methods

Formal methods are mostly relevant in the Critical (or High Integrity) Systems domain where a very high level of assurance is required that the critical system meets is specified requirements. Traditional Formal methods refer to the use of techniques from formal logic and discrete mathematics to software engineering problems, i.e. in the specification, design, and construction of computer systems and software [NASb, Lev86]. However with the advent of hybrid systems which include both discrete and continuous components, continuous mathematics are also becoming involved [Har96, Pop01].

Definition 2.3.1. `(Formal methods)` [HB95]: "A formal method is a set of tools and notations (with a formal semantics) that: (1) are used to specify unambiguously the requirements of a computer system; (2) support the proof of properties of that specification, as well as proofs of correctness of an eventual implementation with respect to that specification." □

Examples of commonly-used Formal methods include the B-Method [Abr96] (see Chapter 5), the Z method [WD96], and the VDM method [Jon86].

The criteria for using a formal methodology include that the formal method must be:

documented, repeatable, teachable, based on proven techniques, validated, and appropriate to the problem being solved [Kne97]. The benefits of using formal methods include among others: precise and rigorous specification, better communication, higher quality, higher productivity, unification of philosophies, separation of concerns, higher confidence in the correctness of software, and making testing easier [Vie93]. The incorporation of lifecycle processes in the application of formal methods also helps to minimize fault introduction in systems design, and to maximize the likelihood and the timeliness of the detection and removal of those faults that may creep in [Vie93].

2.3.1 The state of the art in Formal methods

The two main viewpoints of formal methods [Kne97] are (1) the theoretical study of mathematical logic systems (i.e. Proof Theory or Metamathematics [Kle64]) in order to derive new robust formal methods; and (2) the development of software tool support (i.e. Software Technology) for the pragmatic application of formal methods.

2.3.1.1 Formal specification and verification lifecycle

The incorporation of lifecycle approaches in the application of Formal methods greatly aids the study and development of better formal methods (see Procedure 2.3.1 below). The derivation of tactics from proofs is mainly concerned with the Generalization and Maintenance phase (Step 4.d). However all the phases are relevant when the formal specifications and proofs are developed in-house instead of outsourced—developing the proofs in-house enhances the level of control over the task of deriving robust tactics from proofs in that the human expertise required will be self-evident.

Procedure 2.3.1. The Formal Analysis Lifecycle: *The Current Formal Analysis lifecycle consists of the following phases [Rus99][a]:*

1. Specification: *specify/design in some generally used language/notation, e.g. the Classical Higher-Order logic or B-method. The product of this phase is a formal specification.*

2. Abstraction: *extract the part relevant to the property of interest, e.g. such properties are proof obligations which need to be proved for the specification to be regarded correct. The product of this phase are proof obligations that must be discharged or proved for the specification to be considered correct or valid.*

3. Reduction: *reduce the proof obligations to a form and notation where algorithmic techniques can be applied, e.g. if resolution is the preferred algorithmic technique, the proof-obligations are to be expressed in negated form. The product of this phase is a rewrite of the proof-obligations.*

4. Verification: *Perform the analysis. This phase has its own lifecycle:*

   (a) Exploration: *find the best way to approach the chosen problem. The product of this phase should be a body of theorems that are mostly true.*

   (b) Development: *generate an efficient overall verification. The product of this phase is a plan of the proof, e.g. as depicted by a viable hand proof.*

   (c) Presentation: *prepare the proof for the social review process. The product of the verification should be a genuine proof that some property is satisfied by the specification [b]. Thus the choice of the algorithmic technique to develop the proof is important. For example, resolution theorem-provers do not generate a conventional proof at all; heuristic methods, e.g. proof-plans, can generate proofs that follow unnatural paths; low-level ITP/PCs overwhelm the reader with trivial data.*

   (d) Generalization and Maintenance: *code the specification and proof for reuse. The products of this phase are: (a) a coded description, i.e. program that guides the theorem-prover to repeat the verification "without" human guidance—this corresponds to the definition of LCF-type tactics [MGW96, GMW79]. (b) specification templates to support future applications, to distil general principles, or to explore alternative assumptions and designs.*

5. *Iterate steps (1) to (4.d) until satisfied.*

Figure 2.1: The Formal methods application Lifecycle.

[a]Since this thesis is on Formal Methods, the ideas developed in this theses are presented in a manner which mirrors this Formal Methods Lifecycle.

[b]A genuine proof is defined as a chained argument as in Definition 2.4.6 that will convince a human reviewer [ORSvH95].

The main purpose of the verification lifecycle steps (4.a) to (4.d) sublifecycle is to redress the criticism of some verification methods which typically return 'yes/no' type verifications and/or possibly counterexamples at the level of the abstracted description. For example, theorem-provers present a formal proof as a prooftree in terms of the inference-rules used and the subgoals generated; and model-checkers give counterexamples in the form of a 'trace of events', which is a subset of the alphabet of the formal logic system used, both of which may appear meaningless to a novice user.

The Generalization and Maintenance phase facilitates reuse of specifications and proofs. The generalized proof description should be robust, i.e. the proof description should be a strategy derived from the generated proof, as opposed to an exact line by line collation of the proof commands used in the generated proof [ORSvH95, Wil97]. Deriving such robust tactics from generated proofs can enable the robust tactics to be reused even after small changes are introduced in the original proved specification, e.g. due to changes in requirements or in the interfaces and systems that interact with the one under study. Generalizations of specifications into specification templates is another class of modifications that may be made in order to support future applications, to distill general principles, or to explore alternative assumptions and designs.

Other criticisms of the Formal Analysis Lifecyle (Procedure 2.3.1 above) are mainly concerned with the software tool support available [Rus99], e.g. the reduced models (from step 3 above) are usually built by hand, and this downscaling is considered draconian in that it may produce false negatives and positives in the verification exercise. There is also usually no connection between different tools or analysis methods used in the lifecycle, e.g. whereas Theorem-Provers are mainly used for the Specification and Verification phases, Static Invariant-Generators and Model-Checkers may be used in the Abstraction and Reduction phases respectively. This has mainly led to providing software tool support and integrating different formal methods tools together.

2.3.2 The state of the art in Software Tool Support

Involving the computer in the proof process contributes to the level of formality required when using formal methods [Mar94]. The incidence of human error is reduced—parsers ensure that specifications are syntactically correct; type-checkers ensure that conjectures are type-correct; and the proof system ensures that inference-rules are applied faithfully and securely. The use of a computer can also shorten the time taken in hand specification and proof—computers perform mechanical tasks expeditiously and thus can make proof development efficient.

Since no one Formal Method can tackle all phases of the Formal Methods lifecycle, or a formal software development lifecycle [Bro87]), the state of the art in software tool support involves the integration of different formal methods approaches, e.g. integration of model-checking with automated abstraction, invariant generation, and theorem-proving in PVS [Rus99]. A common approach in formal software development is the embedding of different formalisms in general purpose theorem-provers, e.g. the deep embedding of the *B-Method* in PVS [Mun99].

2.3.2.1 Types of Theorem-provers

Human-oriented Theorem Provers (HOTPs) [Bun96, Bun91] model plausible (i.e. informal) reasoning. Machine-Oriented Automatic Theorem Provers (MOATPs) [Rob63, Rob65], and Interactive Theorem Provers and Proof Checkers (ITPs/PCs) [SORSC98a, SORSC98b, SORSC98c] model formal reasoning, which is advocated for in high-integrity systems construction.

MOATPs lack the techniques that human experts use in proving conjectures thus MOATPs sometimes take too long, or even fail completely, to prove putative theorems. MOATPs proofs also tend to be difficult to follow and understand [Ker98]. ITPs/PCs overcome this limitation by allowing the human user to interact with the ITP/PC in making the strategic decisions required in proof development which make a proof successful. In particular, domain proof experts are good at deciding which inference-rule to apply in the development of a proof, and which variable instantiations to make that can make the proof of a particular conjecture successful. ITPs/PCs however have been criticized for being too interactive [Sch00], thus prolonging the time taken to complete a proof. Partial automation of some of the tasks in interactive theorem-proving by tactics can reduce the amount of this interaction, as well as the knowledge of particular proof techniques required of the human prover.

Theorem-provers have also been integrated with Computer Algebra Systems (CAS), on the one hand to use CAS as calculation oracles to aid theorem-provers in program verification [HT93], and on the other hand to use theorem-provers to verify the results given by CAS [BJ01]. Different Theorem-provers can also be coupled to interact with each other in a proof development exercise. A top-down prover can generate subgoals which are then processed by a bottom-up prover; or conversely lemmas generated by the bottom-up prover are used in a top-down prover to significantly reduce the proof lengths such that proofs can be found with smaller resources [FF98] [1]. A HOTP can be integrated with a MOATP in order to emulate the flexible problem solving behaviour of humans in an

[1]A top-down theorem-prover implements goal-oriented (backwards) proof, whereas a bottom-up theorem-prover implements forwards proof (see Section 2.4.2.2)

agent-based reasoning approach [BJKS99]—the proof-planner splits a goal into subgoals, and the agents, i.e. ATPs, are then appropriately allocated to the subgoals to be solved. The agents allow a number of proof attempts to be executed in parallel, where each attempt is on a separate ATP, thus a number of different proof strategies can be used at the same time in a proof search [Section 2.4.2.2].

A criticism of the use of theorem-provers in verification, is the chicken-and-egg problem of determining the reliability or correctness of the verifier itself, i.e. who verifies the verifier?. The use of model-based specification methods in theorem-provers can help the interpretation of the results given by a theorem-prover.

2.3.3 Model-based formal specification methods

The specification language of a formal method is defined syntactically, i.e. in "strict" formalisation, there is total abstraction from the meaning of the statements being manipulated and the main concern is the arrangement of these statements, and specifically whether proofs or refutations can be constructed [Kle64, Gal86] [2].

Definition 2.3.2. (`Formal specification`) [NASa]: "A formal specification is a set of formulae in a formal language that characterizes a planned or existing system in a particular domain." □

Appendix B lists some formal specifications in the B-Method [Abr96] and PVS specification languages [SORSC98a] used in this work, where the specified requirements are the functional requirements of the computer system.

[2] "Abstraction is the process of simplifying and ignoring irrelevant details and focusing, distilling, and generalising what remains. In formal methods, abstraction is a tool for eliminating distracting detail, avoiding premature commitment to implementation choices, and focusing on the essence of the problem at hand" [NASb].

In a model-based specification the statements in a specification are arranged according to some model of representation and computation thus giving the specification a certain structure (or normal form) as well as semantics. A specification method determines the process by which a satisfying model might be arrived at, e.g. the B-Method [Abr96] uses the Abstract Machine Notation as its specification method. A B-machine specification is a formalisation of an imperative software program in terms of the state-machine model, which typically consists of an abstract representation of a system state, i.e the variables and their values in the system, and a set of operations that manipulate the system state to effect a transition from one state to the next [3].

2.3.3.1 Executable specifications

Traditionally a formal specification describes *what* a system does, but not *how* it does it, i.e. specifications are not executable [Mor94]. However there has been recent and on-going work on making specifications executable, e.g. the execution of Z specifications [Gri00], and the use of declarative languages such as Prolog (logical) [Rob63] and Gopher (functional) [BW88] as both "formal specification and implementation languages". In the logical paradigm, an automatic resolution theorem-prover provides the computational mechanism of proof search; and in the functional paradigm, recursion provides the computational mechanism of reasoning by mathematical induction. However, proof search and recursion are very expensive on computer resources such as memory and CPU-time, which makes declarative languages inefficient for systems implementation.

In order to make the execution of declarative languages more efficient, prover/compiler optimization techniques for these languages can use heuristic methods, which then make

[3]The imperative paradigm is based on Turing machines; the functional paradigm is based on Church's Lambda Calculus; and the object-oriented paradigm is based on the Russell's Simple Theory of Types.

declarative languages unsuitable as formal specification languages since heuristics are informal. The main aim of High Integrity Compilation [Ste98, Str98] is to specify and verify imperative programming languages as well as their respective compilers in order to yield formally verified programming languages and compilers which can be used in High-Integrity systems development.

2.4 Formal Mathematical Logic

Formalism was introduced by David Hilbert in order to tackle the crisis caused by paradoxes and the challenges to classical mathematics by the Intuitionists Brouwer and Weyl [Kle64]. Formalisation is the application of a notation that enables specification of a problem in that notation as well as reasoning about some properties of that specification. The purpose of formalising a theory is to get an explicit definition of that theory (i.e. specification), and what constitutes proof in the theory (i.e. verification), both of which are difficult to do in an informal language, such as natural languages.

Definition 2.4.1. (`Formal system`) [GS93]: A Proof Theory (or Formal (logical) or Natural Deduction System) is a set of rules defined in terms of the following components:

(1) *Alphabet*, i.e. a set of symbols. The symbols in the alphabet usually correspond to whole words (i.e. definitions) instead of to letters [Kle64].

(2) *Syntax*, i.e. rules for building sentences from the alphabet. Sequences of symbols correspond to sentences called formulas. The set of well-formed formulas is the language of the logic.

(3) *Axioms*, i.e. a set of distinguished, (or self-evident) formulas.

(4) *Inference-rules*, i.e. a finite set of instructions for generating new formulas from existing

formulas. A formula is a theorem of the logic if it is an axiom, or if it is generated from the axioms and previously proved theorems by using the rules of inference. □

The *Alphabet* and *Syntax* components of the formal system define the formal specification language; the *Axioms* and *Inference-rules* of the formal system define the proof system. The three main properties considered of a formal system are completeness, soundness and decidability:

Definition 2.4.2. (`Completeness, Soundness, Decidability`) [WD96]: For an expression P specified in the language of the formal system, the formal logic system is:

(1) *complete* iff if P is a theorem of that formal logic, then P can be derived using only the inference-rules of that logic.

(2) *sound/consistent* iff there is no P such that P and $\neg P$ are theorems in that logic.

(3) *decidable* iff for any P there is an effective procedure for showing whether P is a theorem or not in that logic. □

An example of a state of the art Formal logical system or proof theory is the Gentzen Sequent Calculus for Classical Logic (LK) which is discussed in Section 2.5.

2.4.1 Formal logic languages

Propositional logic is regarded as the most basic formal logical system. Propositional classical logic is the language of statements of alleged facts which must be either true or false [WD96]. A proposition is: a *tautology* if it evaluates to *true* in every combination of truth-values of its constituent propositional statements; a *contradiction* if it evaluates to *false* in every combination of its propositional constituents; and a *contingency* if it is neither a tautology nor a contradiction [WD96]. Propositional classical logic is complete, sound, and decidable [Gal86].

First-Order logic allows to express assertions about elements of a given basic type—the assertions are predicates, i.e. statements with a slot for those elements. In First-order logic, completeness and soundness hold but decidability does not hold [Gal86]. Higher-order logics generally refer to n-order predicate logics, where n refers to second, third, etc, and in which quantification is over predicates, predicates of predicates, and so on respectively. Second-order logic appears to lose completeness too as well as decidability; only soundness holds [Gal86]. Higher-order logics are the most expressive of these logics, and are sufficient for the specification of software and hardware [Lei94]. A formal definition of First-order Logic highlights the important properties of propositional and higher-order logics.

Definition 2.4.3. (Alphabet for First-Order Logic) [Gal86]: The alphabet of First-Order logic consists of two parts:

(1) A fixed logical part consisting of:

(i) Truth values: $\top$ [4] (true) and $\perp$ (false) to denote the truth or falsity.

"(ii) Logical connectives: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), $\Leftrightarrow$ (iff/equivalence), and equality (=) to formulate compound (nonatomic) formulae from atomic formulae.

(iii) Quantifiers: $\forall$ (forall), $\exists$ (exists) for binding variables in predicates.

(iv) Variables: a countably infinite set $\mathbb{V} = \{x_0, x_1, ...\}$ to express unknowns.

(v) Auxiliary symbols: parenthesis "(" and ")" to scope the logical connectives.

(2) a non-logical part $\mathbb{L}$ consisting of:

(i) Function symbols: a (countable, possibly empty) set $\mathbb{F}$ of symbols $f_0, f_1, ...$ and a rank function r assigning a positive integer $r(f)$ (called the rank or arity) to every function

[4]$\top$ is not included in the definition in [Gal86]; rather if a formula is provable, then it is considered true. However, especially with tactics, if a formula is unprovable by one tactic, that does not mean the formula is false.

f.

(ii) Constant symbols: a (countable, possibly empty) set $\mathbb{C}$ of symbols $c_0, c_1, \ldots$ each of rank 0.

(iii) Predicate symbols: a (countable, possibly empty) set $\mathbb{P}$ of symbols $P_0, P_1, \ldots$ and a rank function r assigning a nonnegative integer $r(P)$ (called the rank or arity) to every predicate P." □

The sets $\mathbb{V}, \mathbb{F}, \mathbb{C}, \mathbb{P}$ are assumed disjoint; the predicate symbols of rank 0 are propositional symbols (i.e. atomic formula or atoms); the symbols in $\mathbb{V}$, $\mathbb{F}$, $\mathbb{C}$ are of type *term*, and the symbols in $\mathbb{P}$ are of type *formula*.

Definition 2.4.4. (Terms and Formulae) [Gal86]:

"(i) Every constant and every variable is a term.

(ii) If $t_1, \ldots, t_n$ are terms and f is a function symbol of rank $n > 0$, then $f(t_1, \ldots, t_n)$ is a term.

(iii) Every predicate symbol of rank 0 (i.e. a propositional symbol) is an atomic formula, and so is $\perp$.

(iv) If $t_1, \ldots, t_n$ are terms and P is a predicate symbol of rank $n > 0$, then $P(t_1, \ldots, t_n)$ is an atomic formula, and so is $t_1 = t_2$.

(v) For any two formula A, B, then $\neg A$, $A \wedge B$, $A \vee C$, $A \Rightarrow B, A \Leftrightarrow B$ are formulae. For any variable x_i and any formula A, then $\forall\, x_i : A(x)$ and $\exists\, x_i : A$ are formula." □

The semantics of the formulae in a first-order language $\mathbb{L}$ is obtained by interpreting the function, constant and predicate symbols in $\mathbb{L}$ and assigning values to the free variables.

Definition 2.4.5. (Model) [Gal86]: "Given a first-order language $\mathbb{L}$, a $\mathbb{L}$-structure is a pair $\mathbb{M} = (M, I)$ where M (called the domain or carrier of the structure) is a nonempty set of values, and I (the interpretation function) assigns functions and predicates over M to the symbols in L as follows:

(i) For every constant c, $I(c)$ is an element of M.

(ii) for every function symbol f of rank $n > 0$, $I(f) : M^n \rightarrow M$ is an n-ary function.

(iii) For every predicate symbol P of rank $n \geq 0$, $I(P) : M^n \rightarrow BOOL$ is an n-ary predicate. Predicate symbols of rank 0 (i.e. propositions) are interpreted as having the truth values $\top$ (true) or $\bot$ (false)." $\Box$

The main concerns in Logic are validity (Model Theory) and provability (Proof Theory) [Gal86]. A formula A is *satisfiable* in a structure $\mathbb{M}$ iff there exists an assignment $s : \mathbb{V} \rightarrow M$ of variables in A to values in M, such that A evaluates to true (i.e. $\top$). A formula A is *valid* in $\mathbb{M}$ iff A is true for every assignment s, in which case $\mathbb{M}$ is a model of A (i.e. $\mathbb{M} \models A$). For a set of formulae, $\mathbb{M} \models \Gamma$ iff $\mathbb{M}$ is a model for every formula in Γ. A formula is *universally valid* iff the formula is true for any model $\mathbb{M}$. If a formula A is valid (universally or in a model), then that formula is provable (i.e. $\vdash A$).

2.4.2 The formal logic proof system

An efficient way to determine the provability of a logical formula is to construct a proof by using the inference-rules and axioms of the Formal logic system. The formal definition of a formal proof is as follows:

Definition 2.4.6. (Formal Proof) [RC90]: "A formal proof in a Formal System is a sequence of sentences $S_1, ..., S_n$ such that for all $i \leq n$, either: (2) S_i is an axiom; or (2) there are two members S_j, S_k of the sequence with $j, k < i$, which have S_i as a direct consequence by the (forwards) application of an inference rule. S_n is then a *theorem* of the formal deductive system. In such a proof, all of the S_i are theorems since the proof of

S_n can be truncated at S_i, giving a proof of S_i." □

A proof is said to be a proof of its last formula and this formula is said to be formally provable or to be a formal theorem [Kle64]. Such a formal proof can be graphically presented as a tree:

Definition 2.4.7. (Prooftree) [Gal86]: "Given a set Σ of labels, a Σ-tree (or prooftree) is a total function $t : D \to \Sigma$ where D (called the tree domain) is a non-empty subset of strings in the set $\mathbb{N}_+^*$ of all possible strings from the natural numbers satisfying the following conditions: (1) for each $u \in D$, every prefix of u is also in D; and (2) for each $u \in D$, for every $i \in \mathbb{N}_+$, if $ui \in D$, then $uj \in D$ for every j, $1 \leq j \leq i$.

The domain of a tree t is denoted $dom(t)$; and every u in $dom(t)$ is called a node or address." □

A propositional or predicate formula P is provable iff there exists a prooftree in which that formula is the conclusion; and a provable formula is denoted $\vdash P$ [Gal86]. A *proofscript* is an algebraic presentation of a prooftree as a traversal of the prooftree using depth-first search (which is more efficient than breadth-first search [Hop93]).

2.4.2.1 Intuitionistic, Constructive, and Classical proof systems

The Intuitionistic logic proof system consists of the inference-rules for the introduction and elimination of each of the logical connectives described in the fixed logical part of Definition 2.4.3, i.e. $\wedge$ (and), $\vee$ (or), $\Rightarrow$ (implies), $\neg$ (not), $\forall$ (for all), $\exists$ (exists). The Constructive logic proof system is based on the BHK-interpretation [5] that each definition or proof of existence of an object provides an algorithm for computing or constructing that

[5]BHK refers to the mathematicians Brouwer (Intuitionist), Hilbert (Formalist/Classicist) and Kronecker (Constructivist).

object. Intuitionistic and Constructive logic have been advocated for and used as computer programming logics because: (1) Intuitionistic logic has a one-to-one correspondence with typed functional programming languages, i.e. the Typed Lambda Calculus [Wad93]; and (2) an Automatic Theorem prover such as the NuPRL Theorem-Prover [CAB+86], which implements Constructive logic, can be used to automatically synthesize a program from a constructive proof of a logical formula which specifies the program (see Section 2.7.3).

The Classical logic proof system is the Intuitionistic logic proof system with one additional inference-rule, which can be one of the following [GS93]: (1) The rule of Indirect Proof, i.e. in proof by contradiction, to prove P, prove $\neg P$ and if $\neg P$ is true then by soundness P is not true; (2) The rule of Double Negation, i.e. $\neg\neg P = P$ by the truth-table for $\neg$; and (3) The Law of the Excluded Middle, i.e. to prove $P \vee Q$ it is sufficient to prove just Q since if Q is true then $P \vee Q$ is true by the truth table for $\vee$. The addition of these rules to Intuitionistic logic means that an explicit algorithm for the construction of an object may not be given or required in Classical logic, e.g. in the examples above it is not explicitly given how the object formula P is constructed from the Intuitionistic logical-connective inference-rules. Nevertheless some level of constructivism is available in the Classical logic proof system via the use of the logical-connective inference-rules. Ideally, the same proof technique should be used throughout a proof development in order to aid the social review process. The proofs constructed in this work are direct proofs in Classical logic consisting of the inference-rules (2) and (3).

2.4.2.2 Proof Search

In the definition of a formal proof [Definition 2.4.6], the numbering of sentences $S_1, ..., S_n$ where S_n is the formula to be proved, corresponds to a Hilbert Style Forwards proof [6] [Kle64], where the proof of a formula S_n starts from axioms and or assumptions S_1, then inference-rules whose premises match the axioms are applied in a forwards manner (from premises to conclusion) to generate, from the conclusion of the inference-rule, the formulae S_i and eventually the required formula S_n.

The reverse process is the Gentzen-style Backwards-proof [7] whereby the proof search starts from the formula to be proved (i.e. the goal), then an inference-rule whose conclusion matches the existing formula is used to generate the premises of that formula. Usually the structure of the goal, i.e. the connectives in the formula to be proved, determine the inference-rule to be applied in the backwards proof search process. Backwards proof is easier to perform than Forwards proof [Gor88], since in Forwards proof, the axioms and/or assumptions to start from may not be obvious.

Logic or machine induction [FR86] can be used to generate new inference-rules from certain cause-and-effect (i.e. premise(s)-conclusion) examples using plausible (i.e. informal or heuristic) reasoning, where the side-condition is expressed as a measure of confidence, e.g. a percentage, in the effectiveness of the inference rule. However, Logic induction may generate logically incorrect inference-rules if "bad" examples are used, e.g. in the empirical sciences, one can simply try a huge number of cases and conclude that since no counter-examples could be found, the statement must be true. The Gentzen Sequent Calculus is a more viable formal way to derive theorems about mathematical theorems,

[6]This is also known as Bottom-up Proof search or Natural Deduction since this "mimics" the way humans reason from premises towards a conclusion.

[7]This is also known as Goal-oriented or Top-down or Abduction since one is "lead away" from the goal towards axioms.

i.e. Metamathematics.

2.5 The Gentzen Sequent Calculus for Classical logic

The Gentzen Sequent Calculus is a state of the art tool for proof-theoretical investigations. Interactive systems for reasoning about programs are often based on this system as it models ordinary mathematical reasoning more closely than axiom and/or tableaux systems [OS97]. The Gentzen System for Classical Logic (*LK*) is defined as follows:

Definition 2.5.1. (The Gentzen System *LK*) [Gal86]: "The symbols Γ, Δ, Θ, Σ denote arbitrary (possibly empty) sequences of formula, and $A, B, C, ..., P, Q, ...$ are arbitrary formulae. The inference rules of the sequent system *LK* are structural rules (i.e. Weakening, Contraction, Exchange), the Cut rule, and the logical rules for the logic connectives $\wedge$, $\vee, \Rightarrow$, $\neg$, $\forall$, $\exists$." □

2.5.1 Gentzen Sequent Calculus

The Natural Deduction (ND) System (see Definition 2.4.1) formally models the way humans reason. A logical language (see Section 2.4.1) is used to express facts and conjectures about a domain of interest. Formal proofs are constructed by fitting the inference-rules together using backwards proof and/or forwards proof [Section 2.4.2]. As in ordinary human (and mathematical) reasoning, temporary assumptions may be made (and lemmas used) in the course of the proof development, and these have to be discharged for the proof to be completed. The generalizations of formal proof and of provability [Definition 2.4.6] yield the notions of deduction and deducibility to permit the use of any formulas $D_1, \ldots, D_l$ called assumption formulas for the deduction of a goal formula [Kle64]. The structure of a sequent formalises the notion of deducibility under assumptions to localise assumptions to goals in the Natural Deduction System:

Definition 2.5.2. (Sequent) [OS97]: A sequent $\Gamma \vdash_\tau \Delta$, where $\Gamma = A_1, \ldots, A_m$; and $\Delta = C_1, \ldots, C_n$, states that the conjunction of the formulas in Γ implies the disjunction of formulas in Δ, i.e. $A_1, \ldots, A_m \vdash C_1, \ldots, C_n \Leftrightarrow A_1 \wedge \ldots \wedge A_m \Rightarrow C_1 \vee \ldots \vee C_n$. $\Box$

Where $\vdash$ is a Consequence Relational operator, which is read "gives" or "entails"; τ is the context of the formulas; Γ is called the antecedent or assumptions—an empty antecedent ($m = 0$) is equivalent to $\top$ (true); and Δ is called the succedent or consequent—an empty consequent ($n = 0$) is equivalent to $\perp$ (false).

In general, a Gentzen Sequent Calculus is a *meta-language* which directly defines a consequence relation (via the operator $\vdash_\tau$) between formulas of an object language being investigated or used—the relational operator $\vdash$ is not in the alphabet of object language, e.g. $\vdash$ is not in the alphabet of the First-order Classical Logic which is the object language considered in this work. Intuitionistic logic (LJ) restricts the number of formulas n in the succedent to only one formula, i.e. $n \leq 1$, whereas in classical logic (LK) $n \geq 0$ [Kle64]. There is no restriction on the number of formulas m in the antecedent.

Appendix A lists the inference-rules for the Gentzen Sequent Calculus for Classical logic (LK). The first axiom is derived from the notion of deducibility under assumptions, and the other two axioms are derived from the truth table for implication ($\Rightarrow$). The proof of a sequent $\Gamma \Rightarrow \Delta$ completes when the antecedent Γ reduces to *false* or when the succedent Δ reduces to *true*. A sequent in which the antecedent Γ reduces to *true* and the succedent Δ reduces to *false* is unprovable. The symmetries of classical logic are also much better exhibited in sequent formulations of classical logic. Constant logical symbol introduction (or right) rules are the succedent rules: $\vdash \neg, \vdash\wedge, \vdash\vee, \vdash\Rightarrow, \vdash \forall, \vdash \exists, \vdash \perp, \vdash \top$.

Logical symbol elimination (or left) rules are the antecedent rules: $\neg\vdash, \wedge\vdash, \vee\vdash, \Rightarrow\vdash, \forall\vdash, \exists\vdash, \bot\vdash, \top\vdash$.

2.5.2 Metamathematics

The Gentzen Sequent Calculus can be used to reason about other formal systems, and thus yield metamathematical theorems, i.e. mathematical theorems about other mathematical theorems. For example one can use the Gentzen Sequent Calculus system to reason about the definition of correctness to derive metamathematical theorems about correctness.

Definition 2.5.3. (Correctness) [Kne97]: Correctness is the delivery of a proper service that adheres to specified requirements. Formally this requirement can be expressed in either of two ways:

(1) $\forall r : R : \exists s : S : A(r,s)$ (2) $\exists s : S : \forall r : R : A(r,s)$. □

Where R, S are sets of requirements and services respectively, and $A(r,s)$ is the "adheres to" predicate. It thus appears the informal definition of correctness is "ambiguous". The Gentzen Sequent Calculus can be used to reason about whether the two the formal specification statements of the informal correctness requirement are equivalent or not.

The formula to be proved (the goal g) is presented as a conjecture to the Gentzen Sequent Calculus proof system in the form $\vdash g$, i.e. as a sequent whose antecedent list is empty. Figure 2.2 shows the prooftree for the proof of the equivalence of the interpretations of "correctness" [8]. Thus from the prooftree, the two formulations of the informal correctness requirement are not equivalent. Furthermore, the left branch completes with an axiom, thus yielding a finite proof and therefore $\exists s : \forall r : A(r,s) \Rightarrow \forall r : \exists s : A(r,s)$

[8]The PVS Interactive-Theorem-Prover/Proof-Checker system uses the Gentzen Sequent Calculus proof system and the goal-oriented proof search method but displays prooftrees in the Computer Science convention, i.e. with the root at the top [SORSC98a, SORSC98b, SORSC98c].

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A(r_1,s_1)\vdash A(r_1,s_1)}\,[\vdash \mathit{Axiom}]}{A(r_1,s_1)\vdash \exists\, s : A(r_1,s)}\,[\vdash \exists]}{\exists\, s : A(r_1,s)\vdash \exists\, s : A(r_1,s)}\,[\exists \vdash]}{\forall\, r : \exists\, s : A(r,s)\vdash \exists\, s : A(r_1,s)}\,[\forall \vdash]}{\exists\, s : \forall\, r : A(r,s)\vdash \forall\, r : \exists\, s : A(r,s)}\,[\vdash \forall]}{\vdash \exists\, s : \forall\, r : A(r,s) \Rightarrow \forall\, r : \exists\, s : A(r,s)}\,[\vdash\Rightarrow] \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A(r_1,s_2)\vdash A(r_1,s)}{\exists\, s : A(r,s)\vdash A(r_1,s)}\,[\exists \vdash]}{\forall\, r : \exists\, s : A(r,s)\vdash A(r_1,s)}\,[\forall \vdash]}{\forall\, r : \exists\, s : A(r,s)\vdash \forall\, r : A(r,s)}\,[\vdash \forall]}{\forall\, r : \exists\, s : A(r,s)\vdash \exists\, s : \forall\, r : A(r,s)}\,[\vdash \exists]}{\vdash \forall\, r : \exists\, s : A(r,s) \Rightarrow \exists\, s : \forall\, r : A(r,s)}\,[\vdash\Rightarrow]}{\vdash \exists\, s : \forall\, r : A(r,s) \Leftrightarrow \forall\, r : \exists\, s : A(r,s)}\,[\vdash\Leftrightarrow]$$

Figure 2.2: The inequivalence of interpretations of correctness.

The proofscript for the above prooftree is as follows:
([⊢⇔], (([⊢⇒], [⊢ ∀], [∀ ⊢], [∃ ⊢], [⊢ ∃], [*Axiom*]), ([⊢⇒], [⊢ ∃], [⊢ ∀], [∀ ⊢], [∃ ⊢])))

is a theorem of the Gentzen Sequent Calculus LK. The right branch does not complete with an axiom of the LK Sequent Calculus and it requires checking $A(r_1, s_2) \vdash A(r_1, s)$ for every possible r_i, s_i, thus yielding an infinite proof.

The prooftree in Figure 2.2 demonstrates two fundamental results of Proof-Theory: (1) the left branch demonstrates Godel's Completeness Theorem that if a formula is true (valid) then it is provable; and (2) expressions of the form $(\forall\, x : X : \exists\, y : Y : P(x, y))$ are provable and therefore can be used to define correctness criteria in Formal methods. The proof of $\exists\, s : \forall\, r : A(r, s) \Rightarrow \forall\, r : \exists\, s : A(r, s)$ demonstrates that specifications of the form $\forall\, r : \exists\, s : A(r, s)$ can be provable under a model-like assumption $(\exists\, s : \forall\, r : A(r, s))$, hence the advantage of using a model-based specification technique in theorem-proving [Section 2.3.3]. The proof in Figure 2.2 is a constructive proof since the proof yields an algorithm for constructing the provable formula using the inference-rules of the Gentzen System LK.

2.5.2.1 Direct proof via Cut-elimination

The Cut-rule is the only means to introduce a new formula in the current proofstate. The Cut-rule can be viewed as a Rule of Indirect Proof where the cut formula is introduced as a means to shorten the proof. For example, in forwards proof, a cut corresponds to: (1) the transitivity relation, IF $(A \Rightarrow B)$ and $(B \Rightarrow C)$ THEN $(A \Rightarrow C)$; (2) the introduction of lemmas as intermediate steps in proofs in practical mathematics; (3) the deduction of P from $P \vee \neg C$ and $P \vee C$ in resolution C:

$$\frac{C \vdash P \qquad \vdash P, C}{\vdash P}\, Cut(C)$$

In goal-oriented proof (abduction), the cut-formula is used as an assumption in the left premise $(C \vdash P)$, and may have to be proved in the right-premise $\vdash P, C$. Applying the Cut-rule requires creativity in choosing the cut-formula—this may result in a detour in the proof development process if the cut formula is complex or very ingenious. A famous result of Proof Theory is the proof that the *Cut*-rule is redundant in the Gentzen System LK, yielding the famous Gentzen Cut-elimination (or Hauptsatz) Theorem, and the Gentzen System LK without the cut rule, Gentzen System $LK - \{Cut\}$.

One way to show that the cut-rule is redundant in the Gentzen System LK is to demonstrate the logical equivalence of Gentzen Systems G, LK, and $LK - \{Cut\}$, where the System G is the System $LK - \{Cut, Contraction, Weakening, Exchange\}$, i.e. LK without the cut and structural rules [Gal86] (Theorem 6.2.1, page 260). This result is known as a *Normal-form theorem* since it states that a proof not in normal form (i.e. a LK-proof containing the cut-rule(s)) can be reduced to one in a normal form (i.e. a $LK - \{Cut\}$-proof or G-proof without cuts).

On the other hand, a *Normalisation theorem* states the above result, and in addition gives an effective procedure for the reduction of a proof to a normal proof. Gentzen's original Hauptsatz theorem described in [Kle64] (Theorem 48 page 453, Lemma 39 page 454) is a Normalization theorem in that it demonstrates how the *mix-rule* can be permuted upwards with each of the Gentzen Sequent Calculus inference-rules to be eventually eliminated from the proof. The Mix rule is outlined in Figure 2.3, where the sets of formulae $\Pi - \{L\}$ and $\Lambda - \{L\}$ are obtained by deleting all occurrences of formula L from Π and Λ respectively, and the sets of formulae $\Pi + \{L\}, \Lambda + \{L\}$ both contain one *or more occurrences* of a common formula L. A *Mix* is a generalisation of a *Cut* in that a Cut can easily be transformed into a Mix by the use of contractions and exchanges. Conversely a Mix can be transformed into a Cut by using weakenings and exchanges thereby establishing the equivalence of LK with $LK - \{Cut\}$.

The mix-rule is used because of complications in the permutation of the Cut-rule upwards in a prooftree. The Cut-rule proper cannot be permuted above an Exchange-rule involving the cut formula because the application of Gentzen System LK inference rules requires that the formula to be manipulated in a sequent must be at the end of the succedent, or at the beginning of the antecedent. The Cut-rule itself cannot be permuted with the Contraction-rule where the contraction formula is the one introduced by the Cut-rule as the Cut-formula—the contracted formula (L in Figure 2.3) will not be present if the contraction is to occur before the Cut-rule. However when the mix-rule is used instead of the cut-rule, then the Γ and Δ contain one or more occurrences of the mix-formula therefore the contraction after the mix rule can be viewed as redundant. Gentzen proved the cut-elimination theorem *indirectly* by using the Mix-rule instead of the Cut-rule because attempts to demonstrate the redundancy of the cut-rule using the cut-rule proper involve extra complications in proof [BDP00].

$$\dfrac{\dfrac{L, L, \Pi \vdash \Delta}{L, \Pi \vdash \Delta}\, C\vdash \qquad \dfrac{\Gamma \vdash \Lambda, L, L}{\Gamma \vdash \Lambda, L}\, C\vdash}{\Gamma, \Pi \vdash \Lambda, \Delta}\, Cut[L]$$

The Cut cannot be permuted above the Contraction

$$\dfrac{L, L, \Pi \vdash \Delta \qquad \Gamma \vdash \Lambda, L, L}{\Gamma, \Pi - \{L\} \vdash \Lambda - \{L\}}\, Mix[L]$$

Figure 2.3: The Contraction is not required with the Mix-rule

Two corollaries of the Cut-elimination theorem are that the inference-rules in cut-free LK proofs can be permuted without changing the conclusion of the proof, provided: (1) the eigenvariable conditions [9] are not violated; (2) the subformula property is not violated [Sha92]. Since the cut-rule is the only rule by which a new formula can be introduced in the proofstate, all the formulae in a cut-free proof are subformulae of the goal formula:

Definition 2.5.4. (Subformula Property) [Kle64]: "

(1–2) If A is a formula, then A is a subformula of A; and the subformulas of A are subformulas of $\neg A$.

(3–5) If A and B are formulas, then the subformulas of A and the subformulas of B are subformulas of $A \wedge B$, $A \vee B$ and $A \Rightarrow B$.

(5–7) If x is a variable, $A(x)$ is a formula, and t is a term free for x in $A(x)$, then the subformulas of $A(t)$ are subformulas of $\forall\, x : A(x)$ and $\exists\, x : A(x)$.

(8) A formula has only the subformulas required by (1)–(7) above." □

This inductive definition of formulae can be very effective in Automatic Theorem Proving. In general, terms, formulas and proofs can be given by inductive definitions, which define a set S of objects as the smallest set of objects containing a given set X of atoms, and closed under a given set of constructors F [Gal86]. The set S of objects can be conceptualised

[9] Eigenvariable conditions are the side conditions to the applicability of an inference rule

as a free type $S : X \mid f_i\langle\langle x_i \rangle\rangle$ for $1 \leq i \leq n$, where the constructor $f_i : F$ is an injective function from the set X to the set S [WD96]. The inductive definitions can then be proved using the Principle of Mathematical Induction on the formulae and/or the logical connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$). Proofs by mathematical induction can subsequently be implemented as recursive functions, e.g. formal proofs can be represented as prooftrees, which can be defined inductively, and implemented functionally as recursive tactics.

2.5.2.2 Proof complexity

Cut-proofs require creativity in the sense that the cut-formula may not be a subformula of any of the formulae in the conclusion. However proofs which do not use the cut-rule (i.e. cut-free proofs) tend to be exponentially larger than their corresponding proofs where the Cut rule is used (i.e. cut-proofs) [Gal86] (Theorem 6.4.1 page 280). This gives a measure of proof complexity and suggests that: "(1) if some creativity (using cuts) is exercised in proof development, then the proof is considerably shorter than its cut-free counterpart; and (2) there are theorems with no easy proofs in the sense that if the steps are straightforward, then the proofs are very long, or if the proofs are short then the cuts are very creative" [Gal86].

The resultant cut-free prooftree is usually larger than the number of proofsteps in the original prooftree due to the application of structural rules (see Section 3.4.1 for the purposes of structural rules in LK-proof developments). Choosing not to allow structural rules (as in the Gentzen System G) yields a resultant permuted prooftree of the same depth as the original prooftree [10]. Linear Logic, which has been advocated to be more suitable for program verification, disallows the use of contractions and weakenings to mimic the

[10]The Gentzen System G differs from the Gentzen System LK in that for the latter, it is not necessarily true that if the conclusion of a rule is valid, then the premises of the rule are valid [Gal86]. For the System G, the conclusion is valid if and only if the premises are valid.

finiteness of computing resources [Wad93].

2.6 Proof systems for Theorem-proving

The proof system is the inference engine of a logic. Proof systems are amenable for implementation as theorem-provers. The main proof systems considered in this work are:
(1) LCF-Tactics [GMW79] for Gentzen Sequent Calculus [Section 2.5] in Interactive Theorem-Proving and Proof-Checking (ITP/PC) [OS97].
(2) Proof plans [Bun96] for Mathematical Induction and Analogical [MW99] or Case-Based Reasoning [MC98] in Human-Oriented Theorem-Proving (HOTP).
(3) Indirect proof for Resolution and Unification [Rob65] in Machine-Oriented Automatic Theorem-proving (MOATP).

However, since HOTPs model plausible reasoning, which is often based on heuristics which are not entirely formal; and MOATPs often fail to prove putative theorems [Section 2.3.2.1], the preferred proof system in our research is LCF-Tactics [GMW79] for Gentzen Sequent Calculus in Interactive Theorem-Proving and Proof-Checking (ITP/PC).

2.6.1 Tactics

Tactics originated from the work on *Edinburgh LCF* [GMW79], which is essentially a computer program that acts as a proof checker for Scott's Logic of Computable Functions [Sco72]. Tactics support goal-directed proof (or backwards proof), and describe general expression transformations [Mar94]. The formal definition of a tactic is as follows:

Definition 2.6.1. (Tactic) [GMW79]: A (LCF) tactic T, is a function:

$T : \mathit{Goal} \rightarrow \mathit{GoalList} \times (\mathit{ThmList} \rightarrow \mathit{Thm})$. □

Where $g : Goal$ is a formula (or conjecture) to be proved in a Formal system; $l : GoalList$ is a list of subgoal formulas that arise when a tactic $t : T$ is applied to $g : Goal$; and $p : ThmList \rightarrow Thm$ is a forwards proof (or validation function) for the generation of the goal g from the subgoal list l. A simplified view of how tactics work is as follows: for a goal $g : Goal$, subgoals $[g_1, ..., g_n] : GoalList$ and validation $p : ThmList \rightarrow Thm$, if $T(g) = ([g_1, \ldots, g_n], p)$, then if $p_i : [p_1, \ldots, p_n]$ achieve $g_i : [g_1, \ldots, g_n]$ for $1 \leq i \leq n$, then $p(p_1, \ldots, p_n)$ achieves g [RC90].

At its primitive level, a tactic is a single rule of inference applied backwards [11]. Recall that an inference rule consists of one or more premises (i.e. a list of theorems $t : ThmList$) and a single conclusion (i.e. a single theorem $t : Thm$). Thus an inference rule r, when applied in a forwards manner, is a function which takes a list of theorems and returns a single theorem, i.e. $r : ThmList \rightarrow Thm$. Therefore inference-rules can be used as validations for a tactic. The use of inference-rules as validations ensures that the validations can be taken as formal proofs.

At a more complex level, a tactic is a sequential composition of inference-rules. This composition is achieved by tactic language constructs called *tacticals*.

Definition 2.6.2. (Tactical) [GMW79]: A tactical is a function which operates on tactics, and returns a new tactic as a result. □

At an informal level, tacticals are used to compose two or more tactics in various ways to build more complex tactics [SORSC98a], e.g. two or more tactics can be composed to be applied sequentially (i.e. one after another, e.g. the inference-rules in Figure 2.2) or in parallel (i.e. simultaneously, e.g. in Figure 2.2 it is possible to attempt to prove the two branches simultaneously.)

[11] An inference-rule is referred to as a primitive rule [MGW96], or as a primitive tactic [SORSC98a].

There are two possible outcomes when a tactic is applied to a goal expression [MGW96]: (1) if the rule matches the expression (i.e. the expression is in the domain of the rule) then the rule is applied, producing a new expression (i.e. tactic success); or (2) if the rule doesn't match then the rule is said to fail (i.e. tactic failure). In the case of (1), the new expression may be (i) the logical constant *true* ($\top$) or axiom(s) which means the tactic has succeeded in proving the expression (i.e. the tactic is total); or (ii) provable subgoal(s) of the old-expression which means the tactic is a partial proof of the old expression (i.e. the tactic is partial); or (iii) unprovable subgoal(s) of the old expression, e.g. $\perp$ (falsehood). In the case of (2), the old expression remains unchanged. In the cases of (i) and (ii) we say that the tactic is applicable; and in the cases of (iii) and (2), we say that the tactic is not applicable.

Since it may be impossible to prove the new expression(s) in case (iii) above, a safer option is to define tactics so that if they fail to find a proof for a goal expression, then the tactic does not change the proof state, i.e. the goal remains unchanged in its original form. Backtracking is used to revert the goal to its original state [MGW96]. Thus a total tactic succeeds iff the tactic is applicable on a conjecture, and the conjecture is provable with that tactic.

The above interpretation of tactic application is justified by the fact that tactics are supposed to be safe in the sense that they do not give a false proof [GMW79]. For tactics implemented in Angel, "Angelic nondeterminism ensures that "when a tactic presents a choice of possible next steps, the step(s) which will succeed (if any) will be chosen" [MGW96]. It is demonstrated in Chapter 4 Section 4.3, that the PVS instantiation defined-rule `inst?` [12] is not safe nor angelic since it can instantiate with an incorrect term yielding

[12]PVS defined-rules correspond to LCF-type tactics [Table 4.1].

an unprovable proofstate.

2.6.1.1 Implementation of LCF-like tactics

The main idea behind LCF-tactics is to implement backwards proof to generate subgoals from a goal formula (i.e. $Goal \rightarrow GoalList$) and to construct a validation function $ThmList \rightarrow Thm$ for that generation where the subgoals are taken as the list of theorems *ThmList* and the goal as the theorem *Thm*. An inference-rule when applied in a forwards manner can be taken as the validation function, and thus an inference-rule when applied backwards, is a primitive tactic.

This section gives an example implementation of the LK inference-rules Ax, $\vdash\vee$, $\vdash\Rightarrow$ as primitive LCF-tactics. In forwards proof, these LK inference-rules can be defined as functions as follows [13]:

$$Ax : Form \rightarrow Thm = \{a \mapsto \Gamma, a \vdash a, \Delta\}$$

$$OrE : (Thm \times Thm) \rightarrow Thm = \{(\Gamma \vdash a_1, \Delta,\ \Gamma \vdash a_2) \mapsto \Gamma, a_1 \vee a_2 \vdash \Delta\}.$$

$$ImpI : Form \rightarrow Thm \rightarrow Thm = \{a \mapsto (\Gamma \vdash b, \Delta \mapsto \Gamma' \vdash a \Rightarrow b, \Delta)\}$$

Where *Form* is a formula in a sequent, i.e. either an antecedent or consequent formula; Γ' is Γ with all occurrences of a deleted. The primitive tactics corresponding to these inference-rules are then defined as follows:

$$AxTac = \Gamma, a \vdash a, \Delta \mapsto ([\,],\ AxF(a))$$

$$OrETac = \Gamma, a \vee b \vdash \Delta \mapsto ([(\Gamma, a \vdash \Delta), (\Gamma, b \vdash \Delta)],\ \lambda(x, y) : [OrE]\ (x, y))$$

$$ImplTac = \Gamma' \vdash a \Rightarrow b, \Delta \mapsto ([\Gamma, a \vdash b],\ \lambda x : [ImpI]\ a\ x)$$

Where with reference to the definition of a tactic [Definition 2.6.1], and the tactic *OrETac*:

[13]The sequent operators ($\vdash$ and comma ,), and all the logical connectives, bind tighter than $\mapsto$.

$\Gamma, a \vee b \vdash \Delta$ is the goal, $g : \mathit{Goal}$;

$[\Gamma, a \vdash \Delta, \Gamma, b \vdash \Delta]$ is the list of subgoals produced, $[g_1, \ldots, g_n] : \mathit{GoalList}$;

$(\lambda(x, y) : \mathit{OrE}(x, y))$ is the validation function $f : (\mathit{ThmList} \rightarrow \mathit{Thm})$, of the inference-rule.

The sequential composition of tactics to yield a complex tactic in the backwards proof of a goal, i.e. $(t_1 \ ; \ t_2 \ ; \ ...; \ t_n)g = t_1(x) \ \mathit{then} \ t_2(t_1(x)) \ \mathit{then} \ \ldots \ \mathit{then} \ t_n(\ldots(t_1(g))...)$ can be validated by the functional composition of the validation functions, i.e. $(f_1 \circ f_2 \circ \ldots \circ f_n)g = f_1(f_2(\ldots(f_n(g))...))$. For example, the backwards proof of the conjecture $q \vee p \Rightarrow p \vee q$ is given by the following sequential composition of inference-rules:

$$(\mathit{ImplTac};\ \mathit{OrITac};\ \mathit{OrETac};\ (\mathit{AxTac}, \mathit{AxTac}))(\vdash q \vee p \Rightarrow p \vee q)$$
$$= (\mathit{OrITac};\ \mathit{OrETac};\ (\mathit{AxTac}, \mathit{AxTac}))(q \vee p \vdash p \vee q)$$
$$= (\mathit{OrETac};\ (\mathit{AxTac}, \mathit{AxTac}))(q \vee p \vdash p, q)$$
$$= ((\mathit{AxTac}, \mathit{AxTac}))((q \vdash p, q), (p \vdash p, q))$$
$$= [\,], [\,]$$

Where $(\mathit{AxTac}, \mathit{AxTac})((q \vdash p, q), (p \vdash p, q))$ indicates a branching in the prooftree caused by *OrETac* which generates two premises—$(q \vdash p, q)$ and $(p \vdash p, q)$. The empty subgoal list $[\,]$ indicates the successful completion of a proof, i.e. the goal $(\vdash q \vee p \Rightarrow p \vee q)$ is provable and thus a theorem.

2.7 Tactics from proofs state of the art

Approaches to deriving tactics from proofs include: (1) direct encapsulation of proof steps [GMW79, Fel93, FH94]; (2) proof-plans that capture the general direction of a proof [Bun91]; and (3) machine induction [Tar92]. These approaches do not necessarily yield robust tactics since the tactics tend to be specific to the conjecture to be proved, and thus the tactic may fail when applied on a different conjecture.

2.7.1 Direct encapsulation of proof steps

The inference-rules that are applied in the successful proof of a goal conjecture are encoded exactly in the sequence in which the proof steps were executed [GMW79, Fel93]. This was the original idea of deriving tactics from proofs and such tactics are generally called LCF-like tactics. The approach follows naturally in proof-checking where having manually discovered a proof, the human prover maps his/her proof strategy in the language of the ITP/PC of choice, which he/she then uses to ascertain that the proof is correct. Gordon et al [GMW79] use an ITP/PC and the tactic programming language is a subset of ML, whereas Felty et al [Fel93] use an ATP to discover the proof and the tactics are encoded in Lambda Prolog, which is a higher-order logic version of the Prolog programming language [SS86]. Such tactics may fail when the definition of the conjecture is changed, and so are not generally reusable.

This approach is similar to the way the PVS system records a proof for replay—the stored proof is the sequence of proof steps taken to discharge the conjecture, and thus can be viewed as a tactic. However the same stored proof may not be able to prove the same conjecture when the definition of that conjecture is changed.

2.7.2 The encapsulation of proof structures as proof plans

Proof-plans specify LCF-like tactics as a *method*—a "plan is a *method* for one of the top-level tactics, i.e. the specification of a strategy for controlling a whole proof, or a large part of a proof" [BM98] [14].

For example, the mathematical induction proof-plan for an expression $s(x)$ in the Boyer-Moore theorem-prover [BM79] can be encapsulated as the following *methods* (or subplans) [BM98]:

(1) *Induction*—the choice of the induction variable x.

(2) *Ripple-out*—a series of waves that carry the s from one place to another in the base and step cases. The ripple-out subplan consists of two phases: (i) *take-out* which rewrites the recursively defined functions using their base equations; and (ii) *unfolding* which rewrites the recursively defined functions using their step equations.

(3) *Fertilisation*—the induction hypothesis is the 'sperm' that fertilises the step conclusion by making it provable.

(4) *Simplification*—the use of algebraic laws to complete the proof.

However, proof-plans use heuristic knowledge in the design of tactics [Bun91], and heuristics are not entirely formal. Other work which uses heuristics in developing tactics is [Fuc95].

2.7.3 Proofs as programs

This approach has been proposed to formalise the construction of proof-plans [Bun91]. The conventional proofs-as-programs approach [Gre69, MW80, BC85] is concerned with extracting an algorithm from a constructive proof—such an algorithm expressed in a tactic programming language yields a tactic.

In the proofs-as-programs paradigm, a declarative specification, (D, R, I, O) is expressed by the statement that a realization of the specification exists: $\forall(x : D) : \exists(z : R) :$

[14]The authors of PVS do not strictly follow this documentation approach which is very convenient for reasoning about the tactics themselves.

$(I(x) \Rightarrow O(x,z))$, where D is the input data type; R is the output data type; I is the precondition and O is the postcondition [Kre98]. To synthesize an algorithm, a constructive proof of this statement is produced. The constructive proof embodies a method for realizing the specification and this algorithm can then be extracted from the proof.

By analogy with the proof-plan methods, I corresponds to `INPUT`, O corresponds to `OUTPUT`, $x : D$ and $z : R$ are variable instantiations. Thus the constructive proof in the language of the theorem-prover used to construct the constructive proof is a tactic (for that theorem-prover) which takes the `INPUT` predicate, i.e. the conjecture to be proved, and then generates the `OUTPUT` predicate, i.e. the subgoals, which preferably should be axioms meaning that the tactic is total and can successfully complete the proof by itself. Constructive proofs can be performed in the NuPRL theorem-prover [CAB+86] for example. Thus Bundy's *methods* for specifying LCF-tactics can also be useful in the automatic construction of tactics from proofs. Such tactics however tend to be specific to the details of the proof and thus need to be generalized in order to yield a robust proof.

2.7.4 Machine induction

Machine induction (also known as machine learning) [FR86] is an artificial intelligence technique for training computers to extract useful information from past experiences (proofs in this case) and applying the information intelligently themselves to a situation (a conjecture to be proved in this case). The objective of machine learning is to narrow the gap between human experts and automatic theorem provers.

In particular, Tarver uses an adapted form of the genetic algorithm, $M2$, to induce a tactic that solves 14 different conjectures selected from Mendelson's textbook on Mathematical Logic [Tar92]. The induced tactics operate over refinement proofs couched in a sequent calculus format, and the tactics are directly translatable into pure Horn clause programs without negation-as-failure, i.e. the Prolog programming language. The $M2$ algorithm is generic over many kinds of logic and may be applicable over a wide domain of different systems. Similar work on using machine learning to automatically learn proof methods so as to facilitate subsequent reuse include [JKP02, KW94].

However, as in all learning systems, the quality of the induced tactics depends on the quality of the original population of proofs. The machine induction may fail due to tactic interference and negation as failure. Furthermore, because pattern recognition is used in the learning process, teleological (i.e. human) justifications cannot be given for successful proofs. Human justification for proofs is relevant to the social process of reviewing proofs and theorems [DLP79].

2.7.5 Reasoning about tactics

Martin et al [MGW96, Mar94] describe a very general language called Angel, "for expressing tactic programs, making very few assumptions about the form of the expressions (goals) in the target logic, and about the rules which act upon them transforming one expression into another". The work includes up to 92 transformation laws which can be used to improve the efficiency of tactics written in any tactic programming language. Angel does not concern itself with the data structures used to represent conjectures [15], unlike concrete tactic programming languages like that of PVS [SORSC98a] which is more implementation specific in that conjectures are captured using the Common Lisp Object System [Coo00].

Bundy [Bun91] outlines nine criteria for assessing proof-plans:

[15]Whence the generality of the language.

(1) *Correctness:* the tactic for a proof will construct that proof when executed.

(2) *Intuitiveness:* the tactic structures the proof according to human intuitions.

(3) *Psychological validity:* the tactic structures the proof like a mathematician would.

(4) *Expectancy:* there must be a basis for predicting the successful outcome of a proof.

(5) *Generality:* the tactic gets credit from the number of proofs it succeeds on.

(6) *Prescriptiveness:* the tactic generates less search and prescribes rules exactly.

(7) *Simplicity:* a tactic gets more credit for being succinctly stated.

(8) *Efficiency:* a tactic gets more credit when it is computationally efficient.

(9) *Parsimony:* fewer general-purpose tactics are required for some collection of proofs.

2.8 The research problem

In this section we discuss problems in proof development and point the way to the derivation of robust tactics.

2.8.1 Problems in proof theory

The main problems associated with formal methods are [Kne97]:

(1) The limits of mathematics: not everything that is informal can be formalized (and conversely), and not everything is amenable to formal proof. Conjectures expressed in First and/or Higher-order logics can be undecidable.

(2) Scalability: formal methods apply mainly on toy academic examples, i.e. they are generally not suitable for use on industrial scale projects.

(3) Tractability: formal methods techniques are not easily adaptable for use on different projects, i.e. formal methods techniques are generally not reusable.

In general formal methods are not a panacea [BS92, RvH93]; they should be applied judiciously. Formal methods should be applied (1) in particular, on only the critical parts of a system; and (2) in general, on only those parts of a system that are amenable to formal specification and proof.

2.8.1.1 Problems in proof development

Formal proof development tends to be an erudite, error-prone and seemingly interminable task, i.e. the human prover usually has to be an expert in formal proof techniques; human error can be easily introduced when proofs are developed solely by hand; and formal proof usually requires the application of a significant number of inference-rules and thus can take a very long time to complete respectively. Formal proofs of software also tend to be difficult to follow and do not usually undergo social review as with ordinary mathematical proofs [DLP79]. The proofs tend to be specific to the details of the conjecture to be proved and so are not reusable in general, i.e. the proofs are usually not robust [Wil97]. The major challenges in proof development are deciding which proof rule to apply, and which variable instantiations to make that can discharge a conjecture to be proved.

Automated theorem proving is a means to alleviate the problems in proof development—proof search and instantiation are directly available in Prolog as depth-first search and unification respectively [FM87, Fel93]. The popular approach in Interactive Theorem-Proving/Proof-Checking is to use a proof strategy that encodes human expertise in proof search, and to instantiate by pattern matching instantiable variables with those appropriate terms [16] in the current in the proofstate [SORSC98b]. However in the former approach, ATPs may fail to find a solution at all, and may require human guidance to enable the

[16] In instantiation, a variable is substituted by a suitable term, i.e. constants or function symbols (see Definitions 2.4.3, 2.4.4).

search for a suitable instantiation to proceed. In the latter method, the instantiation is not always the correct one, e.g. PVS's most powerful tactic, `(grind)` often performs incorrect instantiations.

2.9 The proposed solution

Though many automated theorem provers (ATPs) can be faster than human experts, there is still a considerable gap between what ATPs can accomplish and what human experts can do. Human experts improve their performance through practice (i.e. learning), and use a whole battery of techniques, past proofs and analogies with past proofs in order to secure a solution [Sch00]. Theorem-proving with tactics can help bridge this gap. In particular, it is possible to formulate tactics that can act as automatic deduction rules for some domain of interest.

Theorem-proving with tactics can be seen as some form of case-based reasoning, i.e. proof by cases, because the tactics are formulated from particular proof cases that are successful. Tactic safety ensures that a tactic will not generate a false proof when it is applied on a particular conjecture. For a particular domain, the tactic may work on one specified conjecture, but may fail when the definition of that conjecture has been slightly changed. An interactive proof development can then be attempted to find a proof for the conjecture on which all the present tactics failed to find a proof. A new tactic can then be derived from that new interactive proof, and the new tactic is composed with the current tactics to yield a composite tactic that can prove all the other previous conjectures and the new (proved) conjecture. Extending current tactics with new tactics in this way can make the resultant composite tactics robust.

Thus for a particular proof-obligation domain D, it is sought a robust tactic $T_D =$

$t_{d_1} \odot t_{d_2} \odot, ..., \odot t_{d_n}$, where $d_i \in D$ is a proof-obligation, $\odot$ is an appropriate tactical, t_{d_i} is a robust tactic for the proof-obligation, and T_D is a robust tactic that can prove any proof-obligation $d_i : D$.

Intuitively, in Interactive Theorem-Proving and/or Proof-Checking it is ideal to derive tactics which perform (all) the creative proofsteps (i.e. those proofsteps that require human ingenuity and can therefore only be partially automated) as early as possible in a proof development so as to leave the rest of the proof development consisting only of mechanical proofsteps (i.e. those that do not require human ingenuity, and can therefore be fully automated). This then dictates an order of application of proofsteps in a proof development and defines a normal form for a formal proof or the tactic which encodes such a proof.

In the case where automatic methods for instantiation may fail, a human-expert with knowledge of the problem domain may be able to introduce instantiation terms via mathematically rigorous creative tactics. In particular the instantiation method of proof in Figure 2.2, where skolemisation is performed as early as possible so that the skolem variables can be used as instantiation terms, is one such mathematically rigorous creative tactic detailed in Chapter 3.

The development of specification templates and tactics can make formal methods scalable and tractable on industrial-scale projects. Specifications that have been proved correct can be reused as components of other larger specifications, or as specification templates for similar problems. The reuse of specification templates and tactics is greatly facilitated by the provision of appropriate software technology.

2.9.1 Robust tactics

The following are two definitions of robustness found in the literature:

Definition 2.9.1. `(Robustness)` Robustness is: (1) the ability of software systems (i.e. tactics) to react appropriately to abnormal conditions [Mey97]; and/or (2) the ability of computer proof systems (i.e. tactics) to demonstrate correctness with minimal human assistance despite modest system or specification changes [Wil97]. □

Robustness complements correctness, where correctness addresses the behaviour of the tactic in cases covered by the specification of the tactic. The first definition is a safety-related issue which is resolved by the fact that LCF-tactics are, and should be, *safe*. The second definition concurs with that given for the argument for proof generalization and maintenance in Section 2.3.1. For example with the straight-forward collation of inference-rules from a proof to yield a composite LCF-tactic (see Section 2.7.1), a change in the specification to yield a new conjecture to be proved may involve the addition of a logical connective, or a new quantified variable. Since this was absent in the original conjecture from whose proof the tactic was derived, that tactic may fail, especially if the quantified variable requires human domain knowledge for the quantifier to be eliminated.

Changes in the specification of a conjecture to be proved are due to the extension of the specification to cater for additional functional/nonfunctional requirements. Extending a tactic with other tactics by the use of tacticals to yield a composite complex tactic is one way of achieving tactic robustness. Therefore Definition (2) requires that the robust tactic must be reusable.

2.9.1.1 Research questions

The major research questions that this work addresses are: (1) Can robust tactics be derived from hand generated proofs?, and (2) Can such robust tactics be incorporated into a state of the art theorem prover? The main goal of this research is to develop a mathematically rigorous method to derive robust tactics from hand-generated proofs.

Chapter 3 addresses the first part; and Chapter 4 addresses the second part of this question. Chapters 5 and 6 applies our solution on the retrenchment development method.

2.10 Summary

Formal methods based on Mathematical Logic techniques can be used to specify the critical properties and prove the correctness of software systems. However, the application of Formal methods tends to be an erudite, error-prone and seemingly interminable task, and in general the formal proofs are not reusable since they tend to be specific to the details of the particular proved goal formula. The major challenges in formal proof development are deciding which proof rule, and variable instantiations/substitutions to apply in proof search. Another major challenge is to generalise a proof into a robust tactic so that the robust tactic can be reusable with minimal human assistance despite modest specification changes in the goal formulae.

Tool support can greatly improve the scalability and tractability of formal methods application. Model-checkers are prone to the state explosion problem, whereas Theorem-provers can safely use of mathematical induction techniques provided that the problem domain is well-ordered. The shortcomings of the three main types of theorem-provers are that: (1) Human-oriented Theorem-Provers (HOTPs) model plausible/informal reasoning which may not be entirely formalisable; (2) Machine-Oriented Automatic Theorem-Provers (MOATPs) often fail to prove putative theorems; and (3) Interactive Theorem-Provers and Proof-Checkers (ITPs/PCs) have been criticized for being too interactive.

Tactics are a means to partially automate formal proof development in ITP/PCs. The current techniques for deriving composite tactics from proofs do not yield sufficiently robust tactics. With the straightforward collation of primitive LCF tactics into composite LCF tactics, the tactic may fail when the definition of the conjecture is changed. The shortcomings of using Machine Learning to induce tactics from proofs include: (1) sensitivity to the quality of the original proofs; (2) the machine induction may fail due to tactic interference and negation as failure; and (3) teleological (formal) justifications cannot be given because pattern recognition is used. Proof-plans also use heuristic knowledge which is not entirely formal, thus such tactics may not be acceptable in an entirely formal setting, e.g. in the verification of High-Integrity or safety-critical systems.

The main goal of this thesis is to develop a mathematically rigorous method for deriving robust tactics from proofs. The major research questions to be addressed in this work are: (1) *Can robust tactics be derived from hand generated proofs?*; and (2) *Can such tactics be incorporated into a state of the art theorem prover?*. The proposed approach to constructing a robust tactic is to use the Gentzen System LK to develop hand-generated proofs for proof obligations that can arise from the specifications of abstract programs, from which can be derived a robust LCF-like tactics with mathematical integrity. If that robust tactic is not applicable when the proof obligation is changed, then another robust tactic can be developed in the same manner for the changed proof obligation, and tacticals are used to extend the other developed robust tactics with the newly developed robust tactic.

Chapter 3

Towards robust tactics from proofs

"Convincing proofs of tactic correctness can be constructed ... Such proofs serve to highlight the properties of the application area which are being exploited: the proof of tactic equivalence generally fails until some property of the basic rules is assumed." [Mar94].

This chapter answers the first research question: *"Can robust tactics be derived from hand-generated proofs?"*. The main goal of this chapter is the formulation of a theory on the construction of robust tactics that can withstand changes in definition of conjectures in Interactive Theorem-Proving/Proof-Checking.

3.1 Introduction

The development of a formal proof requires knowledge of both the proof-theory domain and the application-domain (Section 3.2). A model-based specification method, such as a functional definitional specification style which proceeds from the simplest definition to more complex definitions which in turn can be in terms of the simpler definition (Section 3.2.1), can facilitate discerning patterns in formal proofs. A renowned strategy, which encapsulates the human expertise used in proof search, can also facilitate the hand-generation of the formal proofs of such specifications (Section 3.2.2).

In Section 3.3 a stepwise Tactic Refinement procedure is defined to implement the proposed solution of Chapter 2 Section 2.9. An example is given in Section 3.3.1 on how the ideas above can be used in hand-generated proof development and the straight collation of proof-steps into LCF-tactics (as described in Chapter 2, Section 2.7.1). The example highlights tactic safety, the advantages of an order of precedence in specification and proof, and the potential unreusability of such LCF-like tactics.

A viable way to improve the reusability and thus robustness of LCF-tactics is to classify the proof-steps of the proof system into creative (i.e. proofs-steps which require human ingenuity), or mechanical (i.e. proofsteps which can be performed automatically by a computer) (Section 3.4). It is desirable to perform all creative proof-steps as early as possible in a proof development, thus leaving the rest of the proof consisting of mechanical proof steps which can then be performed automatically by the computer. This proposition gives rise to a subsidiary research question: *"Can creative and mechanical proof-steps be permuted within a proof-tree whilst maintaining the mathematical integrity of the original proof?"*

A permutation analysis of the LK creative inference-rules with the LK mechanical inference-rules is undertaken in Section 3.5 to provide a mathematically rigorous answer to the subsidiary research question above. An algorithm for deriving a normal form of proof based on the results of the permutation analysis is developed in Section 3.7, and an example application of the algorithm is given in Section 3.7.2. The proofs of correctness of this algorithm are given in Section 3.8.

Section 3.9 presents the two main results of this Chapter for the System *LK*: (1) an instantiation proof plan; and (2) a tactic normal form for a goal not in prenex normal form.

These two normal forms are equivalent since logical equivalents can be used as rewrite rules in (1) above, instead of applying the Cut-rule to introduce the prenex normal form of the goal in (2) above. The chapter concludes with a summary of the main results in this Chapter.

3.2 Domain knowledge for proof development

There are two types of domain knowledge: (1) that of the mathematical and meta-mathematical knowledge, i.e. the proof-theory domain; and (2) that of the area in which this mathematical/meta-mathematical knowledge is applied, i.e. the application domain [HJMT95].

In this thesis, the proof theory domain is the Gentzen Sequent Calculus for Classical logic (Section 2.5); and the application domain is the proof obligations that arise from the formal specifications and developments of abstract computer programs expressed in a functional First-Order/Higher-Order Classical logic language. In order to develop a formal proof, a human-prover is expected to have knowledge of both the proof-theory and the application domains. An example of proof-theory domain-knowledge is that it is preferable to perform skolemisation before instantiation to allow the freedom to use the skolem variables as instantiation terms (see Figure 2.2). However in the case that the skolem variables are not suitable instantiation terms, then the human expert can use application domain knowledge to find suitable instantiation terms, e.g. by introducing lemmas about the application domain or defining suitable functions to generate instantiation terms from the skolem variables (see Chapters 5 and 6).

Robust tactics are therefore expected to capture both of these domain-knowledge types to enable a novice human-prover to use the tactics effectively. An Interactive Theorem-Prover/Proof-Checker (IPT/PC) (Chapter 4) can be used as a proof assistant to ensure that the steps in the proof are carried out faithfully without the incidence of human-random error.

3.2.1 A functional definitional specification style

The application and proof-theory domain knowledge can be captured as definitions and theorems, which encapsulate particular concepts in a specification. A functional definitional specification starts with the definitions of the simplest constructs first, and proceeds to the definitions of the more complex constructs which can be in terms of the simpler constructs. For example a functional definitional specification of the set-theoretic operators $\in$, $\notin$, $\subseteq$ and $\not\subseteq$ is in that order of complexity [1]:

$$set : TYPE \rightarrow bool.$$

$$\in, \notin : (TYPE \times set) \rightarrow bool$$

$$\in (x, s) = s(x)$$

$$\notin (x, s) = \neg \in (x, s)$$

$$\subseteq, \not\subseteq : (set \times set) \rightarrow bool$$

$$\subseteq (s_1, s_2) = \forall (x : TYPE) : \in (x, s_1) \Rightarrow \in (x, s_2)$$

$$\not\subseteq (s_1, s_2) = \neg \subseteq (s_1, s_2)$$

The datatype *TYPE* is a given/maximal set or basic type [2], e.g. the integers $\mathbb{Z}$. The signature of a function definition defines the structure (i.e. form or syntax) of the function, e.g. the signatures of set-membership $\in$, $\notin$ are the same but the definitions are different,

[1]These definitions correspond to those given in the PVS prelude file [SORSC98a].

[2]A basic type is a set whose internal structure is invisible—elements of such a set may be introduced, and properties associated with them, but nothing can assumed about the set itself [WD96].

and likewise for the set-inclusion operators $\subseteq$, $\not\subseteq$. The goals or proof obligations to be proved are specified in the order from the simplest to the more complex ones, and proved in that order, since a proof of the simpler proof obligation constitutes a subproof of a more complex proof obligation which may be defined in terms of the former. Although using the definitional specification style aids the brevity of specifications, the overhead is that the definitions may have to be expanded fully for a formal proof to proceed. In developing the proofs from which the LCF-like tactics are to be constructed, a systematic approach which encodes a notion of human expertise in proof development is used.

3.2.2 Human expertise in formal proof development

In the *development* of a formal proof, the application of an inference-rule is either decided: (1) on local hints, i.e. from the information in the proofstate, the human-prover chooses (in an ad-hoc manner) which inference-rule to apply that enables the proof to proceed; or (2) by a well-known proof strategy, e.g. a proof plan [Section 2.7.2]. A renowned interactive proof strategy, variations of which have been used successfully in hardware verifications, e.g. [ALW93, KSK93], encapsulates the human expertise in formal proof development by the following sequence of general proof tasks [COR+95]:

Definition 3.2.1. (ITP/PC Strategy) [COR+95]: The human expertise (in both the proof-theory and application domains) used in the development of a formal proof can be encapsulated by an iteration of the following sequence of proofsteps:

(1) Quantifier elimination (by skolemization, instantiation, mathematical induction).

(2) Unfolding definitions (by expanding definitions, rewriting using definitions, introducing new formulas using assumptions, axioms, and lemmas).

(3) Case analysis (i.e. proof by cases, which splits the proof based on selected boolean

expressions in the current goal; the resulting goals can then be further simplified) □.

This state of the art strategy (or proof plan) is used in developing hand-generated (or interactive) proofs, i.e. given a conjecture to be proved, the inference-rules are tried on the conjecture in the order depicted above. In general, for software verification, an iteration of these proof tasks is required.

3.3 Tactic refinement

An endeavour of constructing robust tactics in the domain of set theory is to derive a tactic that can prove any conjecture which contains any of the set-theoretic operators $\in, \notin, \subseteq, \not\subseteq$, etc. The process starts with a tactic from a proof for $\in$, which is then extended with the tactics from proofs of $\notin, \subseteq, \not\subseteq$, and so on respectively. This stepwise development method for building robust tactics from proofs is coined *Tactic refinement*:

Procedure 3.3.1. *(Tactic refinement:)*

1. For a conjecture g_1, a hand-generated formal proof p_1 is developed.

2. From such a proof, a LCF-tactic t_1, which proves conjecture g_1 automatically, is derived.

3. Tactic t_1 may succeed in the proof of a different conjecture g_2 of similar structure to g_1 but of different definition, or tactic t_1 may fail.

4. In the case tactic t_1 fails, a new hand-generated formal proof p_2, is developed for g_2 and tactic t_2 is derived for the conjecture g_2.

5. The tactics t_1 and t_2 can be composed using tacticals, thus giving a more complex tactic that can prove both g_1 and g_2.

6. This process can be carried out on the conjectures $g_1 \ldots g_n$ that describe some domain D. The composition (using tacticals) of the corresponding tactics $t_1 \ldots t_n$ is then be able

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \top,\bot}[\vdash \top]}{\vdash \top \vee \bot}[\vdash\vee] \qquad \dfrac{}{\vdash \top}[\vdash \top]}{\vdash \top \wedge (\top \vee \bot)}[\vdash\wedge]}{\vdash 4:\mathbb{Z} \wedge (4=4 \vee 4=5)}[\mathit{Arithmetic}]}{\vdash (\lambda(z:\mathbb{Z}):z=4 \vee z=5)(4)}[\lambda E]}{\vdash \{4,5\}(4)}[\mathit{setD}]}{\vdash 4 \in \{4,5\}}[\in D]$$

Figure 3.1: Proof for $\vdash 4 \in \{4,5\}$

The LCF-tactic for the proof above can be formulated as:

$$\mathit{InTac} = (\underline{[\in D]\ ([\mathit{setD}]\ ([\lambda E]}\ ([\mathit{Arithmetic}]\ ([\vdash\wedge]\ (([\vdash\vee]\ ([\vdash \top])),\ ([\vdash \top]))))))$$

to automatically prove any conjecture in that domain. □

Thus for a particular proof-obligation domain D, this results in a "robust" tactic

$$T_D = t_{d_1} \odot t_{d_2} \odot, ..., \odot t_{d_n}$$

Where $\odot$ is an appropriate tactical, t_{d_i} is a tactic for the proof-obligation $d_i \in D$, and T_D is a "robust" tactic that can prove any proof-obligation $d_i : D$.

3.3.1 An example derivation of LCF-like tactics

To facilitate the development of hand-generated proofs, ground terms are used in the specification of the conjectures in this section. Using the notion of human expertise described in Section 3.2.2 above, the proof of $4 \in \{4,5\}$ proceeds as shown in Figure 3.1, where the transformation rule $\underline{([\in D]([\mathit{setD}]([\lambda E])))}$ unfolds the definitions of set membership, set, and the lambda operator respectively ; $[\mathit{Arithmetic}]$ is an arithmetic decision procedure, and $[\vdash\wedge], [\vdash\vee], [\vdash \top]$ are the LK inference-rules.

The LCF-tactic *InTac* fails on the conjecture involving $\notin$. However, since $\notin$ is defined in terms of $\in$, the proof of a conjecture involving $\notin$ contains the underlined segment in

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \bot}{\vdash \bot}\,[\bot \vdash]}{\vdash 5 : T \wedge \bot}\,[\mathit{Algebra}]}{\vdash (5 : T \wedge \bot)}\,[\wedge\vdash]}{\vdash (\lambda(x : T) : \bot)(5)}\,[\lambda\, E]}{\vdash \{x : T \mid \bot\}(5)}\,[\mathit{setD}]}{\vdash 5 \in \{x : T \mid \bot\}}\,[\in D]}{\vdash 5 \in \varnothing}\,[\varnothing D]}{\vdash 5 \in \varnothing}\,[\vdash \neg]}{\vdash 5 \in \varnothing}\,[\not\in D]$$

Figure 3.2: Attempted proof for $5 \in \varnothing$ with NotInTac

the tactic *InTac* in Figure 3.1 above. As expected, the tactic *NotInTac* from the proof of the conjecture $5 \not\in \varnothing$ is applicable to the false statement ($5 \in \varnothing$) and demonstrates that the statement is unprovable as shown in Figure 3.2 [3].

The operators $\in, \not\in$ are of different signature and definition to the operators $\subseteq, \not\subseteq$, but they are of the same parity. The tactic *InTac* and *NotInTac* fail on conjectures involving the operators $\not\subseteq, \subseteq$ because these operators are not in the domain of these tactics. The proof of the conjecture $\{4\} \subseteq \{4, 5\}$ is shown in Figure 3.3. Note that the skolemisation of the universal quantifier by $[\vdash \forall]$ requires the generation of a free (Skolem) variable y for the bound variable x.

The conjecture $\{4\} \not\subseteq \{5, 6\}$ is defined in terms of a negation of the operator $\subseteq$ and this causes the proof of $\{4\} \not\subseteq \{5, 6\}$ to require instantiation (instead of skolemisation) of the bound variable x (see Figure 3.4, where $[\lambda\, \mathit{setE}]$ is the tactic $([\mathit{setD}]([\lambda\, E])))$. The instantiation term for the variable x in the subgoal $\forall(x : \mathbb{Z}) : x \in \{4\} \Rightarrow x \in \{5, 6\} \vdash$ can be easily deduced from the expression $x \in \{4\}$ as $x = 4$ and thus the substitution

[3]$\vdash \bot$ is equivalent to *true* $\Rightarrow$ *false* which is false by the truth table for $\Rightarrow$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y : T, y = 4 \vdash y = 4, y = 5}[Ax] \qquad \dfrac{}{y : T, y = 4 \vdash y : T}[Ax]}{y : T \wedge y = 4 \vdash y : T \wedge (y = 4 \vee y = 5)}[\vdash\wedge]}{y : T \wedge y = 4 \Rightarrow y : T \wedge (y = 4 \vee y = 5)}[\vdash\Rightarrow]}{\vdash (\lambda(z : T) : z = 4)(y) \Rightarrow (\lambda(z : T) : (z = 4 \vee z = 5))(y)}[\lambda E]}{\vdash \{4\}(y) \Rightarrow \{4,5\}(y)}[setD]}{\vdash y \in \{4\} \Rightarrow y \in \{4,5\}}[\in D]}{\vdash \forall(x : \mathbb{Z}) : x \in \{4\} \Rightarrow x \in \{4,5\}}[\vdash \forall]}{\vdash \{4\} \subseteq \{4,5\}}[\subseteq D]}{}$$

Figure 3.3: Proof for $\vdash \{4\} \subseteq \{4,5\}$

The LCF-tactic from the proof above is:

$$SubsetTac = ([\subset D]([\vdash \forall]([\vdash\Rightarrow](\underline{[\in D]([setD]([\lambda E]}([\vdash\wedge](([Ax]),([Ax])))))))))$$

$[4/x]$. Thus the instantiation involves application-domain knowledge of set theory, and the proof-theoretic knowledge that instantiation is required for the subgoal $\forall(x : \mathbb{Z}) : x \in \{4\} \Rightarrow x \in \{5,6\} \vdash$.

Since $\not\subseteq$ is defined in terms of $\in$ and $\subseteq$, the tactic *NotSubsetTac* can be used to prove conjectures involving these operators, but not conjectures involving $\notin$ since *NotSubsetTac* does not involve the $\notin$ operator [4]. The composite tactic

SetTac = *InTac*; *NotInTac*; *SubsetTac*; *NotSubsetTac*, where the semicolon (;) is the sequential tactical operator, is sufficiently robust for the set-theory operators $\in, \notin, \subset, \not\subseteq$.

However, the tactics *NotSubsetTac*, *NotInTac*, *SubsetTac*, *NotSubsetTac* all contain the tactic *InTac* therefore tactic *NotSubsetTac* is not in its normal form. The next sections propose a theory for deriving a normal form tactic for *SetTac* and for LCF-like tactics in general.

[4] A theorem-prover such as PVS, which can apply the negation inference-rules automatically, will enable the tactics *InTac* and *SubsetTac* to also work on conjectures involving $\notin, \not\subseteq$ respectively.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bot \vdash}\,[\bot \vdash]}{4=6 \vdash}\,[\mathit{Algebra}] \qquad \dfrac{\dfrac{}{\bot \vdash}\,[\bot \vdash]}{4=5 \vdash}\,[\mathit{Algebra}]}{4=5 \vee 4=6 \vdash}\,[\vee\vdash]\ (2)}{\{5,6\}(4) \vdash}\,[\lambda\, \mathit{setE}]}{4 \in \{5,6\} \vdash}\,[\in D] \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\vdash \top}\,[\vdash \top]}{\vdash 4=4}\,[\mathit{Algebra}]}{\vdash (\lambda(x:\mathbb{Z}) : x=4)(4)}\,[\lambda\, \mathit{setE}]}{\vdash 4 \in \{4\}}\,[\in D]}{\dfrac{4 \in \{4\} \Rightarrow 4 \in \{5,6\} \vdash}{\dfrac{\forall(x:\mathbb{Z}) : x \in \{4\} \Rightarrow x \in \{5,6\} \vdash}{\dfrac{\{4\} \subseteq \{5,6\} \vdash}{\dfrac{\vdash \neg(\{4\} \subseteq \{5,6\})}{\vdash \{4\} \not\subseteq \{5,6\}}\,[\not\subseteq D]\ (0)}\,[\vdash \neg]}\,[\subseteq D]}\,[\forall \vdash]}\,[\Rightarrow\vdash]\ (1)$$

Figure 3.4: Branch-labeled prooftree for $\{4\} \not\subseteq \{5,6\}$

3.4 An abstraction of LK inference-rules

The inference-rules involved in a proof development exercise can be typically characterized as (1) *creative* (i.e. high-level or strategic steps, which require ingenuity and thus may only be partially automated); or (2) *mechanical* (i.e. low-level or straightforward steps, which require no ingenuity and thus can be automated). The main goal here is to reorder the inference-rules in a hand-generated proof (or LCF-tactics as derived in Section 3.3.1 above) so that creative proofsteps are performed as early as possible thus leaving the rest of the proof consisting of the mechanical rules which can then be applied automatically. This then defines a normal form for encoding tactics which work best in Interactive Theorem-Proving/Proof-Checking whereby the human user performs the creative rules first, and leaves the ITP/PC to generate the rest of the prooftree [5].

[5]This is analogical to planting a genetically-modified seed (a robust tactic derived from a proof) and then watching the seed grow on its accord into a tree of the required shape (a prooftree) under the nourishment of the environment (the ITP/PC).

3.4.1 Mechanical LK inference-rules

The mechanical (low-level/straightforward/automatic) LK inference-rules are defined below:

Definition 3.4.1. (Mechanical LK proof rules) A LK inference-rule is mechanical if the rule does not require application domain-specific knowledge in its application. The LK mechanical proof rules are: *LKMechanical* ::= *Contraction* | *Weakening* | *Interchange* $|\vdash \neg \mid \neg \vdash | \vdash \wedge | \wedge \vdash | \vdash \vee | \vee \vdash | \vdash \Rightarrow | \Rightarrow \vdash | \vdash \forall \mid \exists \vdash$. □

Structural rules (Contraction, Weakening and Interchange) concern sequents in general rather than particular connectives. In goal-oriented proof, contraction rules ($\vdash C$ and $C \vdash$) allow a single occurrence of a formula to be replaced by multiple occurrences of the *same formula*, thus allowing a formula to be used more than once. However, when sequents are considered as sets of formula, the set $\{A\}$ is identical to the set $\{A, A\}$ and therefore contractions are redundant. Formulae are rarely used more than once except in the case of instantiation ($[\forall \vdash]$ and $[\vdash \exists]$), where applying contraction before these rules makes the Gentzen System LK complete [Gal86]. In goal-oriented proof, Weakening rules ($\vdash W$ and $W \vdash$) are used to delete unwanted formulae from the antecedent or consequent of a sequent. However, Linear Logic [Wad93], which has been proposed as most appropriate for software verification, disallows the weakening and contraction rules to mimic the finiteness of computing resources such as memory.

The Exchange rules ($\vdash E$ and $E \vdash$) allow formulas in a sequent to be reordered, thus asserting that the order of the formulas in the antecedent and consequent parts of the sequent is not important [OS97]. If sequents are considered as sets of formulae instead of lists of formulae, the Exchange rule is redundant since order is irrelevant in Set Theory. According to the definition of the original Gentzen System LK inference-rules, the Exchange rule is used to put a formula to be manipulated at the front (end) of the antecedent (succedent) respectively so that the inference-rules can manipulate the formula. Therefore the Exchange rule accounts for the complexity of the proof in manipulating a sequent into the required form so that an inference rule can be applied to that sequent. However, the Exchange rules can be made redundant by allowing the logical connectives to be manipulated in-line where the logical-connectives are in the scope of the inference-rules (parentheses and operator-precedence scope the area of operation of a logical operator).

Skolemisation generates new free terms to be used for eliminating quantifiers in expressions by using the rules $[\vdash \forall]$ and $[\exists \vdash]$. The new variables can be constructed or created without any application-domain knowledge by collecting all variables in the current proofstate, and then generating a new variables that do not currently occur in that current proofstate. This can be done automatically without using any application-domain knowledge, by maintaining this eigenvariable condition. Note that in the Resolution method skolemisation refers to the elimination of the existential quantifier by using Herbrand (or Skolem) functions. This is because Resolution is a refutation method—to prove a goal $(\vdash \exists x : A(x))$ it is required to show that $(\neg \exists x : A(x))$ is *unsatisfiable* with respect to the defined non-logical axioms of the application domain, i.e. there is no self-evident x (in the axioms) such that $A(x)$ becomes true. If the conjunction of $(\neg \exists x : A(x))$ and the axioms derives a contradiction (or there is no model for the conjunction), then by consistency, $(\exists x : A(x))$ is a theorem, otherwise it is not. In Gentzen Sequent Calculus, this assumed negated goal $(\mathit{Axioms} \vdash \neg \exists x : A(x))$, becomes $(\mathit{Axioms} \wedge \exists x : A(x) \vdash)$ by the application of $[\vdash \neg]$, and thus skolemisation $([\exists \vdash])$ is required to eliminate the existential quantifier. Similarly, existential quantifiers in the non-logical *Axioms* are in the antecedent, and therefore require skolemisation. Dually in resolution, the universal quantifiers are simply dropped, which is equivalent to instantiation by the term(s) in the current proofstate in direct proof.

Therefore LK mechanical rules can be considered robust primitive tactics since they are successfully applicable on any formula involving the logical connective in that mechanical rule.

3.4.2 Creative LK inference-rules

These are the creative choices in proof development that usually require application-domain knowledge as well as proof-theory-domain knowledge from the human-user.

Definition 3.4.2. (Creative LK proof rules) A LK inference-rule is creative if the rule requires both proof-theory and application domain-knowledge in its application and thus can only be partially automated. In LK, the creative proof rules are:
LKCreative ::= *Cut* | *Mathematical Induction* $|\vdash \exists \mid \forall \vdash$. □

Mathematical induction ($\vdash \forall$, $\exists \vdash$) requires the choice of a suitable induction variable from a well-ordered (i.e. either numerically or lexicographically) domain; the construction of an induction hypothesis; and the proof of the base and step cases. The choice of these attributes can be quite challenging, thus this is a creative task which requires a proof-theoretically (or heuristically) justified classification of the application-domain knowledge. Section 2.7.2 in Chapter 2 briefly describes the partial automation of the Rippling heuristic induction scheme [BM98] as an LCF-like tactic. The PVS Theorem-Prover/Proof-Checker (see Chapter 4) also comes with a partially automated defined-rule for mathematical induction.

Instantiation [$\vdash \exists$], [$\forall \vdash$] may require ingenuity in the choice of the instantiation terms. For example, it was also demonstrated in Chapter 2, Section 2.5 that goals of the form

$\exists\, s : \forall\, r : A(r, s)$ are not theorems of the Gentzen Sequent Calculus LK largely due to the difficulty of finding a instantiation term m as a model of all the possible r, where m is of the same type as s.

The cut rule is the only viable means of introducing a *new formula* (i.e. the cut-formula), which does not occur in the current proofstate. The application of the Cut-rule requires the choice of the cut-formula, usually with an insight towards shortening the length of a proof, e.g. already proven results can be introduced as intermediate steps in a proof development in the traditional practice of mathematical proof. An example of such lemma introduction is in proof by mathematical induction where the induction hypothesis (which has been proved for the base-case) is applied as a lemma in the proof of the step-case. In that respect, the cut rule is approximated by the transitivity relation (*IF* $(A \Rightarrow B)$ *and* $(B \Rightarrow C)$ *THEN* $(A \Rightarrow C)$), where $(A \vdash C)$ is the goal, $(A \vdash B)$, $(B \vdash C)$ are the subgoals, and the required cut formula is B. Another typical example of the application of the cut-rule is introducing case-formulae, which are boolean expressions that define the cases in the proof-by-cases approach (Step (3) in Definition 3.2.1). For example, the conditional statement, *IF cond THEN* $a1$ *ELSE* $a2$, can be proved by considering two cases: (1) when *cond* is true, $a1$ should hold; and (2) when *cond* is false $a2$ should hold. Thus in general, a case-analysis leads to branching in the prooftree. Ingenuity is required to ensure that the cases chosen complete the problem domain, and that the introduced case expressions are provable; otherwise the case proof is not exhaustive. The ingenuity entails detailed knowledge of both the proof-theory and application domains, as well as creative insight to choose the appropriate concepts to introduce as cut formulae in order to aid the proof search.

Therefore LK creative rules are not robust primitive tactics in that the incorrect choice of the cut-formula, induction variable/hypothesis and the instantiation terms in the creative rules (cut, induction, instantiation respectively) can easily lead to an unprovable proofstate.

3.5 The permutability of LK inference rules

Given a sequent formula to be proved, the proof can proceed by manipulating those connectives available for manipulation. In particular, in most ITP/PCs, e.g. PVS (see Chapter 4), quantifiers can only be manipulated when they are at the front of the sequent-antecedent or end of the sequent-consequent formula, and logical connectives are manipulated when they are in scope. Therefore in the generation of a proof, creative and mechanical rules can be mixed when the mechanical rules are necessary to bring the formula in the scope of the creative rules, e.g. when definitions contain quantifiers (see Figures 3.4, 3.3, Section 3.3.1), or when the goal formula contains some inline quantifiers (see Example 3.5.1 below). The main goal of this section is to give a rigorous mathematical analysis of the reordering (or permutability) of the inference-rules in a hand-generated proof.

Given a collection of n distinct objects, any (linear) arrangement of these objects is called a permutation of the collection [Gri99]. When the objects are proofsteps in a prooftree, the following definition is more relevant:

Definition 3.5.1. (Permutation) [LH01]: The permutation of two adjacent inference rules of a given proof is reversing their order in the proof but without disturbing the rest of the proof (modulo some duplication of proof branches and a renaming of certain variables) as a result of which we get a proof equivalent to the given one. □

For two inference-rules to be permutable, the principal formulas (i.e. the formula each

Example 3.5.1.

$$\begin{array}{cl}
(Q(y_1), R(x_1, y_1)), P(y) \vdash P(x_1), P(x_2) & \\ \hline
(Q(y_1) \wedge R(x_1, y_1)), (P(y) \vdash P(x_1), P(x_2)) & \wedge\vdash \\ \hline
\exists\, y : (Q(y) \wedge R(x_1, y)), (P(y) \vdash P(x_1), P(x_2)) & \exists\vdash \\ \hline
P(y) \vdash P(x_1), \neg\exists\, y : (Q(y) \wedge R(x_1, y)), P(x_2) & \vdash\neg \\ \hline
P(y) \vdash (P(x_1) \vee \neg\exists\, y : (Q(y) \wedge R(x_1, y))), P(x_2) & \vdash\vee) \\ \hline
P(y) \vdash \forall x : (P(x) \vee \neg\exists\, y : (Q(y) \wedge R(x, y))), (\forall x : P(x)) & \vdash\forall\ (\textit{twice}) \\ \hline
(P(y), \neg\forall x : P(x)) \vdash \forall x : (P(x) \vee \neg\exists\, y : (Q(y) \wedge R(x, y))) & \neg\vdash \\ \hline
(P(y) \wedge \neg\forall x : P(x)) \vdash \forall x : (P(x) \vee \neg\exists\, y : (Q(y) \wedge R(x, y))) & \vdash\wedge \\ \hline
\vdash \forall x : (P(x) \vee \neg\exists\, y : (Q(y) \wedge R(x, y))), \neg(P(y) \wedge \neg\forall x : P(x)) & \vdash\neg \\ \hline
\vdash \forall x : (P(x) \vee \neg\exists\, y : (Q(y) \wedge R(x, y))) \vee \neg(P(y) \wedge \neg\forall x : P(x)) & \vdash\vee
\end{array}$$

of the inference rules manipulates) have to be present as two distinct formulas in the sequent. In Example 3.5.1, the quantifier rules cannot be permuted with those logical required to bring the quantifiers to the front for manipulation. Even with the use of geographic/structural tactics [Mar94, MGW96] that target quantifiers first (i.e. can perform inline processing of quantifiers), the presence of negation can cause problems, since the negation changes the semantics of quantifiers, e.g. instead of a skolemisation, an instantiation would be required and vice versa (see Section 3.3.1). Therefore to enable all quantifiers to be eliminated as the first step in a proof development (Definition 3.2.1), a suitable first step is to "rewrite" the goal formula to a form where all quantifiers are at the beginning of the goal formula, i.e. prenex normal form.

3.5.1 Prenex Normal Form

The conversion to prenex normal form corresponds to the Reduction phase in the Formal Methods Lifecycle (Procedure 2.3.1); the goal formula is an Abstraction representing the proof obligation that has to be verified from a Specification.

Example 3.5.2.

$$\begin{array}{cl}
Q(w_1), R(x_1, w_1), P(y) \vdash P(x_1), P(w_1) & \\
\hline
Q(w_1), R(x_1, w_1), P(y), \neg P(w_1) \vdash P(x_1) & \neg\vdash \;\textit{twice} \\
\hline
(Q(w_1) \wedge R(x_1, w_1)), (P(y) \wedge \neg P(w_1)) \vdash P(x_1) & \wedge\vdash \;\textit{twice} \\
\hline
\vdash P(x_1), \neg(Q(w_1) \wedge R(x_1, w_1)), (\neg(P(y) \wedge \neg P(w_1))) & \vdash\neg\textit{twice} \\
\hline
\vdash (P(x_1) \vee (\neg(Q(w_1) \wedge R(x_1, w_1)))) \vee (\neg(P(y) \wedge \neg P(w_1))) & \vdash\vee \;\textit{twice} \\
\hline
\vdash \forall x : \forall w : \forall z : (P(x) \vee (\neg(Q(w) \wedge R(x, w)))) \vee (\neg(P(y) \wedge \neg P(w))) & \vdash\forall\,\textit{thrice} \\
\hline
\vdash \forall x : \forall w : (P(x) \vee (\neg(Q(w) \wedge R(x, w)))) \vee (\forall w : \neg(P(y) \wedge \neg P(w))) & A \vee \forall y : B(y) \equiv \forall y : A \vee B(y)) \\
\hline
\vdash \forall x : \forall w : (P(x) \vee (\neg(Q(w) \wedge R(x, w)))) \vee (\neg\exists w : (P(y) \wedge \neg P(w))) & \neg\exists y : B(y) \equiv \forall y : \neg B(y)) \\
\hline
\vdash \forall x : \forall w : (P(x) \vee (\neg(Q(w) \wedge R(x, w)))) \vee (\neg(P(y) \wedge \exists w : \neg P(w))) & A(x) \wedge \exists y : B(y) \equiv \exists y : (A(x) \wedge B(y)) \\
\hline
\vdash \forall x : (P(x) \vee (\forall w : \neg(Q(w) \wedge R(x, w)))) \vee (\neg(P(y) \wedge \exists w : \neg P(w))) & A(x) \vee \forall y : B(y) \equiv \forall y : (A(x) \vee \neg B(y)) \\
\hline
\vdash \forall x : (P(x) \vee \forall w : \neg(Q(w) \wedge R(x, w))) \vee (\neg(P(y) \wedge \neg\forall z : P(z))) & \neg\forall z : A(z) \equiv \exists z : \neg A(z) \\
\hline
\vdash \forall x : (P(x) \vee \neg\exists w : (Q(w) \wedge R(x, w))) \vee \neg(P(y) \wedge \neg\forall z : P(z)) & \neg\exists w : A(w) \equiv \forall w : \neg A(w) \\
\hline
\vdash \forall x : (P(x) \vee \neg\exists y : (Q(y) \wedge R(x, y))) \vee \neg(P(y) \wedge \neg\forall x : P(x)) & \textit{variable renaming}
\end{array}$$

Logical equivalents are used to rewrite a goal formula into prenex normal form. In addition, bound variables of the same name as some free variables need to be renamed to avoid variable capture—a formula in which all variables are distinct is called a rectified formula [Gal86]. Converting a goal formula to prenex normal form: (i) gives a unique representation of a formula as $Q_1x_1, ... Q_nx_n : P(x_1, ..., x_n)$ where $x_1, ..., x_n$ are distinct variables bound by the quantifiers $Q_i \in \{\exists, \forall\}$, and $P(x_1, ..., x_n)$ (called the matrix) is a quantifier-free formula; and (ii) is useful for eliminating quantifiers first and thus is a useful *Reduction* technique (Step 3 of Procedure 2.3.1) for automated theorem proving. "For every formula A, a prenex normal form formula B can be constructed so that $A \Leftrightarrow B$ is valid (i.e. A is equivalent to B)" [Gal86] (Theorem 7.2.1 page 307). The proof in Example 3.5.2 first rewrites the goal formula into prenex normal form, and is equivalent to the proof in Example 3.5.1 in that the same subgoals are achieved.

Instead of rewriting the goal formula by using equivalents as in Example 3.5.2 above,

Example 3.5.3.

$$\dfrac{P_1 \qquad P_2}{\vdash \forall x, y : (P(x) \vee \neg \exists y : (Q(y) \wedge R(x,y))) \vee \neg(P(y) \wedge \neg \forall x : P(x))}\ \mathit{Cut}$$

Where P_1 is the proof of the equivalence of the original goal to its prenex normal form as follows:

$$\dfrac{}{(P(x_1) \vee (\neg(Q(y_2) \wedge R(x_1,y_2)))) \vee (\neg(P(y_1) \wedge \neg P(x_2))), Q(y_2) \wedge R(x_1,y_2), P(y_1) \vdash P(x_1), P(x_2)}\ \mathit{PropSimp}$$

$$\dfrac{(P(x_1) \vee (\neg(Q(y_2) \wedge R(x_1,y_2)))) \vee (\neg(P(y_1) \wedge \neg P(x_2))), Q(y_2) \wedge R(x_1,y_2), P(y_1) \vdash P(x_1), P(x_2)}{\forall x, y, w, z : (P(x) \vee (\neg(Q(w) \wedge R(x,w)))) \vee (\neg(P(y) \wedge \neg P(z))), Q(y_2) \wedge R(x_1,y_2), P(y_1) \vdash P(x_1), P(x_2)}\ \forall\vdash [4]$$

$$\dfrac{\forall x, y, w, z : (P(x) \vee (\neg(Q(w) \wedge R(x,w)))) \vee (\neg(P(y) \wedge \neg P(z))), Q(y_2) \wedge R(x_1,y_2), P(y_1) \vdash P(x_1), P(x_2)}{g_{PNF}, \exists y : (Q(y) \wedge R(x_1,y)), P(y_1) \vdash P(x_1), P(x_2)}\ \exists\vdash$$

$$\dfrac{g_{PNF}, \exists y : (Q(y) \wedge R(x_1,y)), P(y_1) \vdash P(x_1), P(x_2)}{g_{PNF}, \exists y : (Q(y) \wedge R(x_1,y)), P(y_1) \vdash P(x_1), \forall x : P(x)}\ \vdash\forall$$

$$\dfrac{g_{PNF}, \exists y : (Q(y) \wedge R(x_1,y)), P(y_1) \vdash P(x_1), \forall x : P(x)}{g_{PNF} \vdash P(x_1) \vee \neg \exists y : (Q(y) \wedge R(x,y)), \neg(P(y_1) \wedge \neg \forall x : P(x))}\ \mathit{PropSimp}$$

$$\dfrac{g_{PNF} \vdash P(x_1) \vee \neg \exists y : (Q(y) \wedge R(x,y)), \neg(P(y_1) \wedge \neg \forall x : P(x))}{g_{PNF} \vdash \forall x, y : (P(x) \vee \neg \exists y : (Q(y) \wedge R(x,y))) \vee \neg(P(y) \wedge \neg \forall x : P(x))}\ \vdash\forall$$

P_2 is the proof of the prenex normal form of the goal as follows (the original goal is deleted by a weakening since it has been proved to be "equivalent" to the prenex normal form, which now needs to be proved):

$$\dfrac{Q(w_1), R(x_1,w_1), P(y) \vdash P(x_1), P(z_1)}{Q(w_1), R(x_1,w_1), P(y), \neg P(z_1) \vdash P(x_1)}\ \neg\vdash\ \mathit{twice}$$

$$\dfrac{Q(w_1), R(x_1,w_1), P(y), \neg P(z_1) \vdash P(x_1)}{(Q(w_1) \wedge R(x_1,w_1)), (P(y) \wedge \neg P(z_1)) \vdash P(x_1)}\ \wedge\vdash\ \mathit{twice}$$

$$\dfrac{(Q(w_1) \wedge R(x_1,w_1)), (P(y) \wedge \neg P(z_1)) \vdash P(x_1)}{\vdash P(x_1), \neg(Q(w_1) \wedge R(x_1,w_1)), (\neg(P(y) \wedge \neg P(z_1)))}\ \vdash\neg\mathit{twice}$$

$$\dfrac{\vdash P(x_1), \neg(Q(w_1) \wedge R(x_1,w_1)), (\neg(P(y) \wedge \neg P(z_1)))}{\vdash (P(x_1) \vee (\neg(Q(w_1) \wedge R(x_1,w_1)))) \vee (\neg(P(y) \wedge \neg P(z_1)))}\ \vdash\vee\ \mathit{twice}$$

$$\dfrac{\vdash (P(x_1) \vee (\neg(Q(w_1) \wedge R(x_1,w_1)))) \vee (\neg(P(y) \wedge \neg P(z_1)))}{\vdash \forall x : \forall w : \forall z : (P(x) \vee (\neg(Q(w) \wedge R(x,w)))) \vee (\neg(P(y) \wedge \neg P(z)))}\ \vdash\forall\,\mathit{thrice}$$

$$\dfrac{\vdash \forall x : \forall w : \forall z : (P(x) \vee (\neg(Q(w) \wedge R(x,w)))) \vee (\neg(P(y) \wedge \neg P(z)))}{\vdash g_{PNF},\ (\forall x : (P(x) \vee \neg \exists y : (Q(y) \wedge R(x,y))) \vee \neg(P(y) \wedge \neg \forall x : P(x)))}\ \vdash W\ ;\ [g_{PNF} D]$$

PropSimp is a tactic which applies the LK rules for the logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$. When the above proof is checked in the PVS ITP/PC, *PropSimp* is available in PVS as the tactic `(prop)`. In P_1, the powerful instantiation PVS tactic `inst?` correctly guesses the first two instantiations of variables x, w with the skolem variables x_1, y_2 respectively. However for the instantiation of variables y, z, `inst?` tries skolem constants x_1, y_1 respectively, which yields two unprovable subgoals. The correct human instantiation of variables y, z with skolem variables y_1, x_2 yields the provable proofstate as above. Therefore the PVS `inst?` tactic can be considered as *unangelic* since the tactic can often instantiate incorrectly thus yielding an unprovable proofstate.

one possible application of the cut rule is to introduce the prenex normal form of the goal formula to be proved as demonstrated in Example 3.5.3. Comparing proofs in Example 3.5.3 with the original proof of the same goal formula in Examples 3.5.1 and 3.5.2, it is evident that the prooftrees in Example 3.5.3 are in a normal form. There is a clear distinction between quantifier rules and propositional rules in the subproof P_1 (the deduction of the goal formula from its prenex normal form). In the proof of P_2 (the prenex normal form of the goal), first the quantifiers are eliminated, and the rest of the proof consists of application of the propositional inference-rules. In the prooftree in Example 3.5.1, the quantifier rules are mixed with the logical rules since the proof is constrained by bringing quantifiers to the front before they are eliminated. In Example 3.5.2, the reduction of the original goal to its prenex normal form using logical equivalents is replaced in Example 3.5.3 by the application of the cut-rule, the proof of subtree P_1 and the weakening of the original goal formula in subtree P_2. All three examples use the idea of skolemizing quantifiers first and using the skolem variables as potential instantiation terms (as demonstrated in Figure 2.2, Section 2.5, Chapter 2).

The next step is to justify the proposition that the LK creative rules should be applied as early as possible, therefore leaving only the mechanical rules to be applied afterwards. The crucial criteria in rearranging logical inference rules is that a different order of rules is used but the same conclusion is reached. The analysis of the permutability of the inference rules of LK with each other may proceed by considering all cases of the permutation of all logical rules with each other, all quantifier rules with either a logical rule, or a structural rule or the cut rule. However by the Gentzen Hauptsatz Theorem, the Cut-rule permutes with all the other LK rules, and the $\vdash \exists$ and $\exists \vdash$ rules are dual to the $\forall \vdash$ and $\vdash \forall$ rules respectively. Tables A.2, A.3, A.1 in Appendix A.1 typifies the permutation of the creative rules with all the other LK rules.

3.5.2 Permutability of the cut-rule with other LK-rules

The Cut-rule mostly used in the literature [Gal86, Kle64] is

$$\frac{\Lambda \vdash \Theta, L \qquad L, \Gamma \vdash \Delta}{\Gamma, \Lambda \vdash \Delta, \Theta}\ \mathit{AtomicCutB}$$

In the form of the Cut-rule above, the antecedent (Λ, Γ) and succedent (Θ, Δ) formulas are split between the two branches. In this section, the simpler form of the Cut rule [OS97, Pau99] is used:

$$\frac{L, \Gamma \vdash \Delta \qquad \Gamma \vdash \Delta, L}{\Gamma \vdash \Delta}\ \mathit{AtomicCutL}$$

The same antecedent (Γ) and succedent (Δ) formulas are repeated in both the two branches, which makes clearer the conception of the cut-rule as a way to introduce lemmas or cases in proof since the original sequent formulas are both involved in the assumption (left branch) and proof (right branch) of the lemma/case formula L.

The Gentzen Cut Elimination (Hauptsatz) Theorem (Section 2.5.2.1) demonstrates that the Cut-rule permutes with all the other LK rules. Definition 3.4.2 classifies the cut-rule is as a creative proof rule, therefore it is permuted down the prooftree. Applying the cut rule before the contraction avoids the problem of permuting cut above the contraction, which forced Gentzen to use the mix rule in his permutation analysis.

The cut-rule is a branching rule, i.e. an inference-rule with more than one premise, thus when a cut is permuted below a rule, that rule can then duplicated in the branches of the cut-rule in order to produce the same leaves as in the original prooftree. In the permutation of the cut-rule with skolemisation ($\vdash \forall$, $\exists \vdash$): (1) the eigenvariable v (the skolem variable) should not appear in the conclusion, e.g. in Δ, otherwise the proof would violate the eigenvariable condition; and (2) $A[v/x]$ cannot be the cut formula since the eigenvariable condition would also be violated in that case.

3.5.3 Permutability of quantifier-rules with other LK-rules

Since the $\forall \vdash$ and $\vdash \forall$ rules are dual to the $\vdash \exists$ and $\exists \vdash$ rules respectively, it suffices to consider only the $\vdash \exists$ (instantiation) and $\vdash \forall$ (skolemisation or induction) rules in this case. For those permutations not involving Cut, all formulae in the permuted prooftree are subformula of the goal formula (see Section 2.5.2.1). The permutation of instantiation ($\vdash \exists$) with skolemisation/induction ($\vdash \forall$) satisfies the eigenvariable conditions that the skolem/induction terms should be new variables which do not occur in the conclusion.

3.6 Results from the permutation analysis

Tables A.1, A.2, A.3 in Appendix A.1 show the permutation analysis of these quantifier with the other LK-rules including Cut. The permutation analysis shows that in Gentzen Sequent Calculus for Classical logic (LK) it is possible to rearrange the order of inference-rules in a developed proof without prejudicing the validity of the proof [6].

3.6.1 Creative compositions of proofsteps

As demonstrated in Example 3.5.3, the Cut introduces the prenex normal form of the goal formula; the original form of the goal is then deleted in the right branch by a Weakening; and the prenex normal form of the goal is contracted to allow completeness of the Gentzen System LK. Thus a creative combination of the inference rules is *Cut*; $\vdash W$; $\vdash C$, which yields a normal form of proof for goals which are not in prenex normal form.

[6]In Intuitionistic logic (LJ) it is not possible to do so [Sha92].

Definition 3.6.1. (Tactic-proof normal form) A normal form of tactic-proof is a finite proof where all the creative rules are applied as early as possible in a proof thus leaving the rest of the proof consisting of mechanical rules possibly automatic. For a goal g where g is not in prenex normal form, the tactic-proof normal form is a proof of the form:

$$\dfrac{\dfrac{\begin{matrix}\text{- }(Axiom)\\ \vdots\end{matrix}}{g_{PNF} \vdash g}\,LKrules \qquad \dfrac{\dfrac{\dfrac{\begin{matrix}\text{- }Axiom\\ \vdots\end{matrix}}{\vdash g_{PNF}, g_{PNF}}\,Creative;\ Mechanicalsteps}{\vdash g_{PNF}}\vdash C}{\vdash g, g_{PNF}}\vdash W}{\vdash g}\,Cut_{g_{PNF}}$$

□

This provides an alternative way to reason about the reduction of a goal formula to prenex normal form. However this reduction can be replaced by definitional equivalences, and thus eliminating this use of the cut-rule.

When skolemization is performed after instantiation, the eigenvariable condition requires that the skolem variable z must not occur in the conclusion. Hence the skolem variable z must be a new variable (or constant) and cannot be the instantiation term t. When skolemisation is performed after instantiation, the instantiation term t can be a variable (Definition 2.4.4), e.g. t can be the skolem variable z and the eigenvariable condition is not violated. Therefore performing skolemisation before instantiation can lead to more freedom in the choice of instantiation terms since the skolem variables can be used as instantiation terms pending type correctness conditions, i.e. the instantiation term should be of the same type as the skolem variable. Thus $\vdash \forall$; $\vdash \exists$ (which is equivalent to $\exists \vdash$; $\forall \vdash$) is a creative composition of proofsteps. These claims are demonstrated in the permutation Cut with $\exists \vdash$ (Tables A.3) and in Figure 2.2.

Theorem 3.6.1. *(Instantiation with skolem terms): Given a sequent $\Gamma \vdash \Delta$ where each* *formula in* Γ, Δ *is in prenex normal form* $\forall^n x_i : X : \exists^m y_j : Y : P(x, y)$ *for* $i, j, n, m \geq 1$ *then instantiation terms* t_j *for existential variables* y_j *can be generated from the skolem variables* v_i *for the universal variables* x_i *by a function* $f : X \rightarrow Y$*, i.e.* $t_j = f(v_i)$ *where the function* f *generates instantiation terms whilst ensuring type-correctness conditions between the sets* X, Y.

Proof: *by the permutation analysis of skolemisation with instantiation [Section 3.5].* □

3.6.2 Factorizing inference-rules

The permutation analysis above also shows that for the LK branching rules (Cut, $\vee\vdash$, $\Rightarrow\vdash$, $\vdash\wedge$), performing the branching rule after a non-branching rule has the effect of factoring out all common proofsteps among subtrees at the same level, so that only one instance of the non-branching rule is applied in the *stem/trunk* of the prooftree (see Appendix A.1).

Definition 3.6.2. (Common inference-rules): For a given proof, an inference-rule is common if the same inference-rule occurs among different subtrees at the same level of the prooftree. □

A common inference-rule can be either a creative-rule, e.g. the $\vdash \exists$ in the permutation analysis in Appendix A.1; or a mechanical-rule, e.g. the inference-rules $[\in], [\lambda\, setE]$ in Figure 3.4. This factorization of common inference-rules can be achieved by repeated permutation.

Theorem 3.6.2. *(Rule Factorization): Given a prooftree* P *in which a branching rule is applied before a non-branching rule, if the two rules are permutable, then applying the non-branching rule before the branching rule yields a prooftree* P' *in which the non-branching rule is factored out from the branches of prooftree* P *to be applied only once.*

Proof: *by repeated permutation.* □

The permutation of the branching cut-rule upwards the prooftree (Gentzen Cut-elimination Theorem) can be seen as a way of Factorizing the prooftree. The purpose of performing branching rules as late as possible is to reduce the duplication of proofsteps that the user may have to endure if the branching rules are applied as early as possible.

3.6.3 A strategy for choosing which proofrule to apply

The analysis of the permutations of the LK rules with each other can determine which rules are to be applied first, i.e. the order of application of the rules. The example proof which involves instantiation (see Figure 3.4, Section 3.3.1) shows that by first rewriting the goal (where the goal involves definitions and negation), the instantiation terms can be more easily deduced, and in addition, the goal would be in its normal form. The permutation analysis demonstrates that it is possible to permute downwards the prooftree the LK creative proofsteps (*Cut*, $\vdash \exists$, $\forall \vdash$) with all the other LK rules provided that: (1) the formulae are in prenex normal form; (2) the eigenvariable conditions are satisfied; and (3) inline processing of logical connectives is allowed. Definition 3.6.1 provides an alternative means of introducing the prenex normal form of a goal formula; Theorem 3.6.1 provides a viable means for eliminating quantifiers; and Theorem 3.6.2 states that it is preferable to perform non-branching rules before the branching ones. Therefore Theorem 3.6.3 below depicts an order of application of LK proof rules which can be used as a strategy in an ITP/PC to yield a formal proof in normal form.

Theorem 3.6.3. *(Tactic-Proof Normal Form): If a conjecture $\vdash g$ in predicate calculus is LK-provable by a proof P, then there is a normal form proof P' where:*

(1) creative compositions of LK rules and the mechanical rules ($\vdash\Rightarrow$, $\vdash \neg$, $\neg \vdash$) are used to rewrite $\vdash g$ into sequent subgoals $\Gamma_i \vdash \Delta_i$ where each formula in Γ_i, Δ_i is in Prenex

Normal form [by Definitions 2.5.2, 3.6.1 and Appendix A].

(2) Creative rules (instantiation, induction) are used to eliminate the quantifiers in $\Gamma_i \vdash \Delta_i$ to yield quantifier-free sequents $\Gamma'_i \vdash \Delta'_i$ [by Theorem 3.6.1].

(3) Each $\Gamma' \vdash \Delta'$ is provable by LK mechanical rules where branching rules are applied as late as possible [by Theorem 3.6.2].

This produces a normal form of proof where creative proofsteps are applied as early as possible by the human prover to leave the rest of the proof to be performed automatically by the theorem prover via decision procedures for propositional logic and arithmetic.

Proof: *from the permutation analysis and results in Section 3.5 and Appendix A.1 respectively.* □

In applying the Cut-rule to introduce the prenex normal form of the goal, the reduction in proofsteps required of the human-prover is afforded in that there is no overhead in applying mechanical-rules to rewrite the goal formula so that all quantifiers are at the front, in particular when the proof of the left branch is ignored. If the goal formula is already in prenex normal form, the reduction in proofsteps required of the human-prover is afforded in that the creative proofsteps required of the human-prover are applied first, thus leaving the rest of the proof automatic. In general the Cut-rule can be used to introduce new formulae, e.g. definitions, lemmas, and cases in goal-oriented proof with mathematical integrity. Thus the unfolding of definitions in Definition 3.2.1 can be regarded as a form of Cut-rule application.

Theorem 3.6.3 can be used in the Generalization and Maintenance phase of the Formal Analysis Lifecycle to transform a given formal proof into the normal form of proof. This normal form of proof can the be encoded as a tactic which can be composed with other tactics in such a normal form to yield a tactic-proof each of whose subtrees (or subproofs) is in normal form.

3.7 A procedure for constructing robust tactics

In this section an algorithm that can transform a proof into the normal form given by Theorem 3.6.3 is presented and proved. A method for deriving a normal form of tactic-proof can be conceptualised as follows:

Procedure 3.7.1. *(Deriving a normal form of tactic-proof) A normal form of tactic-proof can be derived from a finite proof (which is not in normal form) by distinguishing the inference-rules used in the proof into creative and mechanical inference-rules, Factorizing the proof description, and then permuting the inference-rules in accordance to Theorem 3.6.3.* □

The procedure is applied iteratively (or recursively) starting from the root of the developed prooftree and proceeding upwards towards the axioms or leaves of the prooftree. The factorization involves factoring out the prooftree into subtrees (which are effectively branches of the prooftree), where each subtree is a subproof of the complete proof. A further factorization involves factoring out the inference-rules that are common between all the subtrees at the same level. These common inference-rules can be distinguished into creative and/or low-level inference rules.

The permutation involves two phases. The first phase involves permuting the inference-rules of the subtrees so that the common inference-rules are applied just before the inference-rule that generated the subtrees. This yields a new subproof for the subtrees at the same level where the common inference-rules are 'bubbled down' towards the root of the prooftree. The second phase involves permuting the inference-rules in the resultant proofstem, so that the creative inference-rules in that proofstem, which are lower in the creative lattice, are applied first. This bubbles down the creative inference-rules (lower in the creative-lattice) towards the root of the prooftree.

3.7.1 An algorithm for constructing robust tactics

Procedure 3.7.1 can be conceptualized into an iterative algorithm as shown in Figure 3.5. The $\{P\}\ S\ \{Q\}$ notation [Hoa69, Bac86] is used in writing the algorithm in order to verify the correctness of the algorithm, i.e. $\{P\} = \{branch = 0\}$ is the precondition, S is the program fragment, and $\{Q\} = \{branch = n\}$ is the postcondition. *LatticePermute* uses the Theorem 3.6.3 to permute inference-rules. *Permute* does not use Theorem 3.6.3, but permutes common inference-rules with the inference-rule that caused the branching. *Distinguish* separates creative inference-rules from mechanical inference-rules. Collecting inference-rules at a branch-label i is defined as in Algorithm CollectBranchRules, where *CollectRules*(i) collects inference-rules at branch i to the next branch $(i + 1)$ for each subgoal.

3.7.1.1 Labeling the prooftree

Given a prooftree and/or its corresponding proofscript, the branches produced by inference-rules with more than one premise in the prooftree are labeled as follows.

Definition 3.7.1. (Labeled Prooftree) A generic labeled prooftree can be conceived as:

$$\dfrac{\dfrac{\dfrac{\cdots}{\cdots}\,[e](3)}{\overset{\vdots}{\vdash g.1}}\,[d](2) \qquad \dfrac{}{\overset{\vdots}{\vdash g.2}}\,[c] \qquad \cdots \qquad \dfrac{\cdots}{\vdash g.i}\,[b](4)}{\overset{\vdots}{\vdash g}(0)}\,[a](1)$$

Algorithm 3.7.1. *(Normalize: an algorithm for transforming a formal proof into normal form)*

```
{branch = 0}
FOR branch := 0 TO n DO %the branches are numbered from 0 up to n.
    IF branch = 0 then begin %Define the root as at branch 0
        LatticePermute(Distinguish(CollectBranchRules(branch)))
        branch := branch + 1
    ELSE
        Permute(FactorCommonRules(CollectBranchRules(branch)))
        LatticePermute(Distinguish(CollectBranchRules(branch − 1)))
        branch := branch + 1
    END
END
{branch = n + 1}
```

Algorithm 3.7.2. *(CollectBranchRules(i))*

```
FOR goal := g.1 TO g.n DO %these are the subgoals at that branch-level
    CollectRules(i)
END
```

Figure 3.5: An algorithm for constructing tactics from proofs.

Where the g is the goal; $g.1, g.2, g.i$, the horizontal dots $\ldots$, and the vertical dots $\vdots$, are subgoals. The numbers in the parentheses—$(0), (1), (2), (3)$—are branch-labels; and $[a], [b], [c], [d], [e]$ are the branching inference-rules.

The branching of the prooftree is identified as instances of double opening brackets in a proofscript—the individual subtrees at a branch are separated by commas. The linear representation of the above prooftree is:

$$
\begin{aligned}
&{}_{(0)}(* * * \; ([a] \; ({}_{(1)}(* * *([d] \; ({}_{(2)}(*, *, *,) \; ([e] \; ({}_{(3)}(*, *, *)))),\\
&\qquad (* * * \; ([c])),\\
&\qquad * * *,\\
&\qquad ([b] \; ({}_{(4)}(*, *, *)))))))
\end{aligned}
$$

Where $* * *$ are the inference-rules for the subgoals $\ldots$; the numbers $(1), (2), (3)$ denote the branches; and the commas separate the subtrees. □

For each subgoal, the branch-labeling proceeds from that subgoal to the leaves of that subgoal, i.e. depth-first search [Hop93]; and then from left to right for subgoals at the same branch-level, i.e. breadth-first search [Hop93]. For example, with reference to the above tree, the branches of the whole subtree for the first subgoal $g.1$ is labeled completely (depth-first search), before proceeding to label the branches of the subsequent subgoals $g.2, \ldots, g.i$ which are on the same-branch-level as $g.1$, (breadth-first-search). See Section 3.7.2 for how the prooftree in Figure 3.4 would be labeled.

3.7.1.2 Collecting proofrules

For each subgoal, the collection of all the inference-rules up to the next branching level is called a proofstem. The inference-rules in each proofstem are collected into a list and the lists for each subgoal are concatenated to give a list of all inference-rules at that branch up to the next branch for all the subgoals at the same level (see Section 3.7.2 for an example). A proofstem is effectively a subproof, or a complete proof for a subgoal, e.g. the proof of $5 \notin \varnothing$ in Figure 3.2 consists of one proofstem which is the complete proof for that conjecture. The proof in Figure 3.1 consists of three proofstems: (1) $([\in D]\ ([\mathit{setD}]\ ([\lambda\, E]\ ([\mathit{Arithmetic}]\ ([\vdash\wedge])))))$; (2) $([\vdash\vee]\ ([\vdash\ \top]))$; and (3) $([\vdash\ \top])$. Thus in Figure 3.1 the inference-rules at branch 1 are $\langle[\vdash\vee], [\vdash\ \top], [\vdash\ \top]\rangle$.

3.7.1.3 Factoring out common inference-rules

Factoring out common inference-rules among subtrees means that each of these common inference-rules only has to be applied once instead of n times where n is the number of subtrees in which they occur, thus reducing the time taken to complete the proof by $(n-1)t$ where t is the time taken by each of these inference-rules. Furthermore, the factorization maintains the order of the inference rules in the original prooftree.

Only the inference-rules that are not axioms can be factored out because axioms are the final derivations which require the intermediate proofsteps in order to be generated. For example, in Figure 3.1, at branch (1), the inference-rule [$\vdash \top$] is common among the two subtrees at that branch but [$\vdash \top$] is an axiom, therefore it cannot be factored out since this axiom can only be deduced from the branching rule [$\vdash\wedge$].

3.7.1.4 Permuting the factored inference-rules

The common inference-rules form a new proofstem for all the subtrees at the same branch-level. The permutation includes the inference-rule that caused the branching. This is illustrated in Section 3.7.2.

Theorem 3.6.3 defines which inference-rules may be applied before others. For example, in Figure 3.4, the creative-lattice dictates that the definition expansion [$\in D$] (which is a form of Cut application (see Section 3.6.3)) should come before the instantiation [$\forall \vdash$].

The effect of this algorithm is to (1) capture commonalities between subproofs at the same level, and thus improve efficiency; (2) bubble creative inference-rules down the prooftree so that they can be performed as early as possible; and (3) enable robustness of the encoded resultant tactic by allowing the creative proofsteps to be passed as actual parameters when the tactic is invoked by the human-prover. Algorithm 3.7.1 can be applied recursively on the resultant prooftree that it produces until there is no change in subsequent prooftrees that the algorithm generates.

3.7.2 An example application of Algorithm 3.7.1

We apply Algorithm 3.7.1 on the prooftree for conjecture $\{4\} \not\subseteq \{5, 6\}$ in which the branching rule [$\Rightarrow\vdash$] is applied earlier in the proof development.

1. The prooftree is labeled as in Figure 3.4.

2. First iteration; *branch* = 0, and so the *If*-part is executed:

   (a) *CollectBranchRules*(0) gives the inference-rules:

   $\langle [\not\subseteq D], [\vdash \neg], [\subseteq D], [\forall \vdash] \rangle$.

   (b) *Distinguish* gives the creative inference-rules:

   $\langle [\not\subseteq D], [\subseteq D], [\forall] \vdash \rangle$.

   (c) *LatticePermute* returns the same sequence:

   $\langle [\not\subseteq D], [\subseteq D], [\forall] \vdash \rangle$.

   (d) *branch* := 1

3. Second iteration—*branch* = 1, and so the *Else*-part is executed:

   (a) *CollectBranchRules*(1) gives the inference-rules:

   $\langle \langle [\in D], [\lambda\, setE] \rangle, \langle [\in D], [\lambda\, setE], [Algebra], [\vdash \top] \rangle \rangle$.

   (b) *FactorCommonRules* gives the common inference-rules:

   $\langle [\in D], [\lambda\, setE] \rangle$

   (c) *Permute* gives the prooftree in Figure 3.6.

   (d) *CollectBranchRules*(0) gives the inference-rules:

   $\langle [\not\subseteq D], [\vdash \neg], [\subseteq D], \vdash \forall], [\in D], [\lambda\, setE] \rangle$.

   (e) *Distinguish* gives the creative inference-rules:

   $\langle [\not\subseteq D], [\subseteq D], [\vdash \forall], [\in D] \rangle$.

   (f) *LatticePermute* returns the new sequence:

   $\langle [\not\subseteq D], [\subseteq D], [\in D], [\vdash \forall] \rangle$.

   This gives the prooftree in Figure 3.7

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bot \vdash}\,[\bot \vdash]}{4=6 \vdash}\,[\mathit{Algebra}] \qquad \dfrac{\dfrac{}{\bot \vdash}\,[\bot \vdash]}{4=5 \vdash}\,[\mathit{Algebra}]}{4=5 \lor 4=6 \vdash}\,[\lor\vdash]\ (2) \qquad \dfrac{\dfrac{}{\vdash \top}\,[\vdash \top]}{\vdash 4=4}\,[\mathit{Algebra}]}{4=4 \Rightarrow (4=5 \lor 4=6) \vdash}\,[\Rightarrow\vdash]\ (1)}{(\{4\}(4) \Rightarrow \{5,6\}(4)) \vdash}\,[\lambda\, \mathit{setE}]}{4 \in \{4\} \Rightarrow 4 \in \{5,6\} \vdash}\,[\in D]}{\forall (x:\mathbb{Z}) : x \in \{4\} \Rightarrow x \in \{5,6\} \vdash}\,[\forall \vdash]}{\{4\} \subseteq \{5,6\} \vdash}\,[\subseteq D]}{\vdash \neg\{4\} \subseteq \{5,6\}}\,[\vdash \neg]}{\vdash \{4\} \not\subseteq \{5,6\}\ (0)}\,[\not\subseteq D]
$$

Figure 3.6: Factorization and permutation of the prooftree in Figure 3.4 at branch (1)

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bot \vdash}\,[\bot \vdash]}{4=6 \vdash}\,[\mathit{Algebra}] \qquad \dfrac{\dfrac{}{\bot \vdash}\,[\bot \vdash]}{4=5 \vdash}\,[\mathit{Algebra}]}{4=5 \lor 4=6 \vdash}\,[\lor\vdash]\ (2) \qquad \dfrac{\dfrac{}{\vdash \top}\,[\vdash \top]}{\vdash 4=4}\,[\mathit{Algebra}]}{\top \Rightarrow \bot \vdash}\,[\Rightarrow\vdash]}{(\{4\}(4) \Rightarrow \{5,6\}(4)) \vdash}\,[\lambda\, \mathit{setE}]}{\forall (x:\mathbb{Z}) : \{4\}(x) \Rightarrow \{5,6\}(x) \vdash}\,[\forall \vdash]}{\forall (4 \in \{4\} \Rightarrow 4 \in \{5,6\}) \vdash}\,[\in D]}{\{4\} \subseteq \{5,6\} \vdash}\,[\subseteq D]}{\vdash \neg\{4\} \subseteq \{5,6\}}\,[\vdash \neg]}{\vdash \{4\} \not\subseteq \{5,6\}}\,[\not\subseteq D]
$$

Figure 3.7: Lattice permutation of the prooftree in Figure 3.4 at branch (1)

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bot\vdash}\,[\bot\vdash] \quad \dfrac{}{\bot\vdash}\,[\bot\vdash]}{\bot\vee\bot\vdash}\,[\vee\vdash]\ (2)}{4=5\vee 4=6\vdash}\,[\mathit{Algebra}] \qquad \dfrac{\dfrac{}{\vdash\top}\,[\vdash\top]}{\vdash 4=4}\,[\mathit{Algebra}]}{4=4\Rightarrow(4=5\vee 4=6)\vdash}\,[\Rightarrow\vdash]\ (1)}{(\{4\}(4)\Rightarrow\{5,6\}(4))\vdash}\,[\lambda\, setE]}{\forall(x:\mathbb{Z}):\{4\}(x)\Rightarrow\{5,6\}(x)\vdash}\,[\vdash\forall]}{\forall(x:\mathbb{Z}):x\in\{4\}\Rightarrow x\in\{5,6\}\vdash}\,[\in D]}{\{4\}\subseteq\{5,6\}\vdash}\,[\subseteq D]}{\vdash\neg\{4\}\subseteq\{5,6\}}\,[\vdash\neg]}{\vdash\{4\}\not\subseteq\{5,6\}\ 0}\,[\not\subseteq D]}{}
$$

Figure 3.8: Factorization and permutation of the prooftree in Figure 3.7 at branch (2)

(g) *branch* $:= 2$

4. *branch* $= 2$ is satisfied by the *for*-statement *forbranch* $:=$ 0 *to* 2, and so we have a third iteration. The ELSE-part of Algorithm 3.7.1 is executed:

   (a) *CollectBranchRules*(2) gives the inference-rules:

   $\langle\langle[\mathit{Algebra}],[\bot\vdash]\rangle,\langle[\mathit{Algebra}],[\bot\vdash]\rangle\rangle$.

   (b) *FactorCommonRules* gives the common inference-rules:

   $\langle[\mathit{Algebra}]\rangle$. The rule $[\bot\vdash]$ is an axiom and thus should not be factored out.

   (c) *Permute* gives the prooftree in Figure 3.8.

   (d) *CollectBranchRules*(1) gives the inference-rules:

   $\langle[\mathit{Algebra}]\rangle,\langle[\mathit{Algebra}]\rangle$.

   (e) *Distinguish* gives the empty list of creative inference-rules since [*Algebra*] is a low-level inference-rule.

   (f) *LatticePermute* returns the same proofstem $\langle[\mathit{Algebra}]\rangle$:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\bot\vdash}[\bot\vdash]\quad\dfrac{}{\bot\vdash}[\bot\vdash]}{\bot\vee\bot\vdash}[\vee\vdash]\ (2)\qquad\dfrac{}{\vdash\top}[\vdash\top]}{\top\Rightarrow(\bot\vee\bot)\vdash}[\Rightarrow\vdash]\ (1)}{4=4\Rightarrow(4=5\vee 4=6)\vdash}[\mathit{Algebra}]\ \ (\mathit{completion})}{(\{4\}(4)\Rightarrow\{5,6\}(4))\vdash}[\lambda\, \mathit{setE}]\ \ (\mathit{completion}}{\forall(x:\mathbb{Z}):\{4\}(x)\Rightarrow\{5,6\}(x)\vdash}[\vdash\forall]\ \ (\mathit{instantiation})}{\forall(x:\mathbb{Z}):x\in\{4\}\Rightarrow x\in\{5,6\}\vdash}[\in D]\ \ (\mathit{rewrite})}{\{4\}\subseteq\{5,6\}\vdash}[\subseteq D]\ \ (\mathit{rewrite})}{\vdash\neg\{4\}\subseteq\{5,6\}}[\vdash\neg]\ \ (\mathit{rewrite})}{\vdash\{4\}\not\subseteq\{5,6\}\ (0)}[\not\subseteq D]\ \ (\mathit{rewrite})
$$

Figure 3.9: The normal form of the prooftree in Figure 3.4

This gives the same prooftree as in step 4(c) above.

(g) *branch* := 3 makes *branch* = 3.

5. This is not satisfied by *for branch* := 0 *to* 2, and so the algorithm terminates with *branch* = 3 as required.

A second application of Algorithm 3.7.1 on the prooftree in step 4(c) above factors out the inference rule [*Algebra*] at branch 1 to give the following final prooftree in Figure 3.9.

Thus the tactic that results from the application of our algorithm is:

$\mathit{NotSubsetTac}' =$

$([\not\subseteq]\ ([\vdash\neg]\ ([\subseteq]\ ([\in D]\ ([\forall\vdash]\ ([\lambda\, \mathit{setE}]\ ([\Rightarrow\vdash]\ (([\vdash\top]),\ ([\bot\vdash])))))))))$

For a proofstem consisting of quantifier elimination, the resultant proofstem consists of three regiments: (1) a rewrite phase from the root to the quantifier elimination inference-rule prepares the conjecture for quantifier elimination; (2) a quantifier elimination phase where the quantifier is eliminated; and (3) a completion phase, which involves mechanical inference-rules to complete the proof.

Comparing the prooftree in Figure 3.9 to the one in Step 1 Figure 3.4 above, the inference-rules $[\lambda\, setE], [\in D]$ have been factored out from the two proof branches into the proofstem to be applied once in the proof development. Furthermore, by application of the creative-lattice, the inference-rule $[\in D]$ has been bubbled down the prooftree below the instantiation inference-rule $[\forall \vdash]$, and this unfolding of the definition of set membership facilitates the finding of the instantiation term, 4, as a witness to the proof. Note that $[\lambda\, setE]$ requires this witness in order for the boolean computation of $\{4\}(x) \Rightarrow \{5,6\}(x)$ to complete. We can view the inference-rules $([\not\subseteq D]([\vdash \neg]([\subseteq D]([\in D]))))$ as the rewrite phase; the inference-rules $[\forall \vdash]$ as the instantiation phase; and the inference-rules $([\lambda\, setE]([\Rightarrow\vdash]((\vdash \top), ([\bot \vdash]))))$ as the completion phase of the proof. The tactic *NotSubsetTac′* can thus be seen as a proof plan for instantiation proofs.

3.8 Proofs of correctness of Algorithm 3.7.1

The proof consists of two parts: (1) a proof by induction on the termination of the algorithm; and (2) a proof of the properties of the algorithm's output.

3.8.1 Proof of termination of the algorithm

We are interested in the termination of and the properties of the resultant tactic given by Algorithm 3.7.1, which is iterative on the number of branches in the prooftree. The branches are labeled using natural numbers, and natural numbers constitute a well-ordered domain. The mathematical correlation between the subgoal generation and the branch-labeling in a given prooftree is that the root is labeled as branch 0, and the last subgoal generation in the prooftree is labeled n, where n is the number of branches in the whole proof tree. Thus we can prove the correctness and termination of Algorithm 3.7.1 for any prooftree with n branches by using mathematical induction.

The FOR-loop terminates when the guard condition is not satisfied, i.e. $\neg(0 \leq b \leq n)$, which simplifies to $b > n$ since $b : \mathbb{N}$. The IF-THEN-ELSE statement within the FOR-loop can be formalised as $((b = 0 \Rightarrow S(b)) \wedge (b > 0 \Rightarrow T(b-1)))$ where $S(b)$ is the IF-part, and $T(b-1)$ is the ELSE-part. Putting this all together, the termination condition for Algorithm 3.7.1 for any tree with n branches is:

$$\vdash \forall n : \exists(b \mid (b = 0 \Rightarrow S(b)) \wedge b > 0 \Rightarrow T(b-1)) : b > n).$$

By induction on n, the base case $n = 0$, i.e. a tree with no branches, requires to prove the goal:

$$\vdash \exists(b \mid (b = 0 \Rightarrow S(b)) \wedge (b > 0 \Rightarrow T(b-1))) : b > 0)$$

The statement *branch* := *branch* + 1 in the FOR-loop yields the instantiation $[0 + 1/b]$. For the step case, the induction hypothesis, $(\exists(b \mid (b = 0 \Rightarrow S(b)) \wedge (b > 0 \Rightarrow T(b-1))) : b > j)$, is assumed to hold for a tree with $n = j$ branches, and it is required to prove the step case $n = j + 1$ as follows:

$$\vdash \forall j : (\exists(b \mid (b = 0 \Rightarrow S(b)) \wedge b > 0 \Rightarrow T(b-1)) : b > j)$$
$$\Rightarrow (\exists(b \mid (b = 0 \Rightarrow S(b)) \wedge b > 0 \Rightarrow T(b-1)) : b > j + 1).$$

Skolemisation, simplification, and the statement *branch* := *branch* + 1 in the FOR-loop establishes the instantiation $[((j + 1) + 1)/b]$.

Note that since the proof above is valid for any tree with n branches, then Algorithm 3.7.1 can also be applied on an infinite prooftree, or a prooftree while the prooftree is in the process of being developed. To complete the induction proofs above, it is also required to establish that in the base case, the IF-part of the FOR-loop holds, i.e. $\vdash S(0)$ holds; and in the step case, the ELSE-part also holds, i.e. $b > j \vdash T(j+1)$. These properties are established by the proofs in the following section.

3.8.2 Proof of correctness properties of the algorithm's output

Algorithm 3.7.1 takes a formal proof P encoded as an LCF-like tactic t_1, and returns a tactic t_2, where t_2 is expected to: (1) preserve the correctness of t_1 and thus the proof P; (2) be more robust than t_1; and (3) be more reusable than t_1. However, robustness and reusability are nonfunctional requirements which are difficult to subject to verification since they both cannot be expressed succinctly using mathematical formula—both these properties can be validated by suitable case studies (see Section 3.7.2 and the ensuing chapters).

Tactic t_2 is a syntactic variation of tactic t_1 in that the proofsteps in tactic t_1 are factorized and permuted according to the creative lattice to yield t_2 . To verify correctness preservation, we can argue in terms of the properties of the abstractions used to transform a proof P to the tactics t_1 and t_2. Theorem 3.8.1 describes the correctness preservation of proof P by tactics t_1 and t_2.

Theorem 3.8.1. *(Correctness preservation of formal proof P by LCF-like tactic t_1 and robust-LCF-like tactic t_2) The robust-LCF-like tactic t_2 produced by Algorithm 3.7.1 preserves the correctness of the LCF-like tactic t_1 derived from the formal proof P, i.e. $\forall\, g : Goals : t_2(g) = t_1(g)$.* $\square$

3.8.2.1 Proof of Theorem 3.8.1

The proof consists of the proofs of the four lemmas below. First it is shown that the direct collation of the proofsteps used in the development of a proof P into an LCF-tactic t_1 maintains the integrity of the proof P.

Lemma 3.8.2. *Correctness Preservation of formal proof P by LCF-like tactic t_1: The LCF-like tactic t_1 preserves the correctness of the proof P from which the tactic t_1 is derived, i.e. $\forall g : Goals : t_1(g) = P$*

Proof:

1. By Definition 2.4.6, a formal proof is a cartesian product of a set of sentences with a set of inference rules, i.e. $\{S_1, ..., S_n\} \times \{R_1, ..., R_{n-1}\}$.

2. By Definition 2.6.1, a simple tactic is a function from a goal S_n : Goal to a cartesian product of a list of goals, S_i : GoalList with a validation function R_i : (ThmList $\rightarrow$ Thm). Tactic t_2 is a composition of such simple tactics which generates the list of subgoals $[S_{n-1}, ..., S_1]$ = GoalList from the goal S_n using the inference-rules $R_{n_1}, .., R_1$: (ThmList $\rightarrow$ Thm)

3. Therefore $t_1(S_n)$ gives a formal proof P of the goal S_n. □

Algorithm 3.7.1 factors out common inference-rules among branches at the same level. This factorization process can be seen as repeated permutation.

Lemma 3.8.3. *Factorization as repeated Permutation: The factorization of proofsteps common among branches at the same level can be achieved by repeated permutation according to the permutation lattice.*

Proof:

By the permutation analysis (see Sections 3.5, 3.6). In addition, the factorization involves *only those inference-rules that are not axioms, i.e. all the inference-rules except* $[Ax]$, $[\bot \vdash]$, $[\vdash \top]$. □

The validity of Theorem 3.8.1 depends on the validity of re-ordering (permuting) inference-rules in a cut-free proof:

Lemma 3.8.4. *(Lattice permutability) Mechanical (low-level) inference-rules and creative (high-level) inference-rules can be permuted in a prooftree according to Theorem 3.6.3.*

Proof:

By the permutation analysis in Section 3.5 and Theorem 3.6.3. □

Algorithm 3.7.1 uses a tree-labeling system to process a prooftree:

Lemma 3.8.5. *(Prooftree labeling): The labeling of the branching of the prooftree is valid.*

Proof : *by the proof for termination of the Algorithm 3.7.1 (see Section 3.8.1).* □

3.9 Tactic-proof normal forms

Algorithm 3.7.1 yields a normal form for proofs involving instantiation as demonstrated by the example in Section 3.7.2. The normal form is conceptualized as consisting of the following regiments.

Definition 3.9.1. `(Instantiation proof plan)` Proofs involving instantiation can be proved using the following phases:

(1) Rewrite—this involves bringing the $\exists$ in the consequent, or the $\forall$ in the antecedent to the front of the formula by unfolding definitions, skolemizing, induction and/or creative tactics (but not instantiation).

(2) Instantiate—this is the elimination of the $\exists$ in the consequent, or the $\forall$ in the antecedent, using $[\vdash \exists]$ and $[\forall \vdash]$. Deciding the expression to instantiate with may be

achievable automatically but in general, this may require human guidance.

(3) Completion—this corresponds to the application of LK mechanical rules and decision procedures to finish the proof. □

In the tactic coding of such a normal form prooftree, the arguments to the creative inference-rules, e.g. the definitions and instantiations used in the proof development can be taken as formal parameters to the tactic. A conjecture of similar structure but different semantics may require a different instantiation of these parameters, which is correctly derived by a different instantiation of the formal parameters, hence deriving a tactic that can prove that conjecture. This makes the tactic robust in that it can cater for different conjectures of same structure but possibly different meaning. Furthermore in the proof in Figure 3.4, the rewrite phase makes the instantiation trivial.

In Section 3.5.2 it was demonstrated how the cut rule can be applied as the first creative rule in a proof development to introduce the prenex normal form of a goal which is not in prenex normal form, thus yielding the Tactic-Cut-proof normal (Definition 3.6.1). The rewrite phase is equivalent to introducing the prenex normal form of the goal formula using the Cut-rule, proving the deduction of the prenex normal form of the goal from the original form of the goal formula (i.e. proof of the left branch), and weakening the right-branch to delete the original form of goal formula.

3.10 Summary

In this chapter we have developed a formal method for deriving robust tactics from hand-generated formal proofs. The method is based on the application of the Formal Methods Lifecycle (Procedure 2.3.1). The state of the art formal method used is the Gentzen

Sequent Calculus LK and the proofs are those of conjectures that may arise from the formal specifications of computer programs in Classical Logic (Section 3.2). The summary of the main results in this chapter are:

1. The definition of Procedure 3.3.1 for proof obligations (POs) which are specified in a functional definitional style (Section 3.2.1) for the purpose that the proofs of the simpler POs are subproofs of the more complex POs. Teleological justifications for patterns in proof obligations (see Section 2.7.4) can then be given in terms of this specification design and purpose.

2. The application of Procedure 3.2.1 to the proofs of set-theoretic operators $\in$, $\notin$, $\subseteq$, $\subseteq$ using a state of the art Interactive Proof Procedure (Definition 3.2.1) demonstrates that the LCF-tactics from the straightforward collation of formal proofsteps are (see Section 3.3.1): (1) safe in that they will not generate a false proof; (2) unreusable when the definition of the conjecture is changed; and (3) extendible using tacticals to yield a *robust* tactic $RobustTac = T_1;\ T_2;\ \ldots;\ T_n$ (where ; is the tactical for combining tactics in sequence), which may be reusable to prove other conjectures in that domain.

3. The abstraction of LK inference-rules into mechanical (which can be completely automated, Definition 3.4.1) and creative (which can only be partially automated, Definition 3.4.2). The mechanical rules are robust primitive tactics since they are reusable on any conjecture, whereas the creative rules (cut, induction, instantiation) can easily introduce terms and formulae which may result in an unprovable proofstate.

4. The justification of the abstraction above by a mathematically rigorous permutation

analysis of the creative rules with the mechanical rules (Section 3.5), which also gives the following results:

(a) Performing the creative steps as soon as possible leaves the rest of the proof consisting of mechanical steps which are easily performed automatically, thus the rest of the proof is automatic (Section 3.4).

(b) The creative Cut-rule can be used to introduce the prenex normal form of the goal formula, which leads to the elimination of quantifiers first, followed by an automatic proof completion using the mechanical LK rules. This gives a Normal form of Proof (Definition 3.6.1) for goals which are not in prenex normal form.

(c) Performing creative steps first can reduce proof complexity, e.g. the proof of the left branch in Definition 3.6.1 is equivalent to using definitional equivalences to rewrite the original goal formula into prenex normal form (Section 2.5.2.2).

(d) Performing non-branching rules first rather than performing the branching rules first has the effect of factoring out common proofsteps among branches in a prooftree, e.g. ($\vdash \exists$; $\vdash\Rightarrow$) has the effect of factoring out common instantiation instances from the ($\vdash\Rightarrow$; $\vdash \exists$) (Section 3.6.2).

(e) Complex tactics can be derived as creative compositions of proofsteps, e.g. (Skolemisation;Instantiation) gives the option of using skolem variables as instantiation terms, and (Cut;Weaken;Contraction) makes the LK system complete for the prenex normal form of the goal formula (Section 3.6.1).

(f) Permutation analysis gives a classification of proofsteps (Theorem 3.6.1, Theorem 3.6.2, Theorem 3.6.3), which can be used as an order of application of proofrules to yield a formal proof in normal form.

5. The derivation of Algorithms 3.7.1, 3.7.2 from the permutation analysis of LK inference-rules, which were demonstrated in Section 3.7.2 to yield a normal form of proofs involving instantiation (Definition 3.9.1).

6. The proofs of correctness of these Algorithms (Section 3.8), thus justifying the use of this algorithm to construct a normal form of proof, which can then be encoded as a tactic.

The next chapter looks at encoding tactics from proofs in the state of the art PVS Interactive Theorem Prover and Proof Checker.

Chapter 4

Robust tactics from proofs in PVS

PVS is an interactive environment for writing formal specifications and checking formal proofs. [COR+95].

This chapter addresses the second research question: *"Can the tactics derived using Algorithm 3.7.1 be incorporated into a state of the art theorem-prover?"*.

4.1 Introduction

A theorem-prover/proof-checker that supports interactive proof development, the Gentzen System LK, and the encoding of tactics, can be used to facilitate the development of hand proofs, and the incorporation into that ITP/PC of those tactics yielded by the application of Algorithms 3.7.1, 3.7.2 on a developed proof. This chapter describes the encoding of robust tactics in such a state-of-the-art theorem-prover/proof-checker, PVS [SORSC98a, SORSC98b, SORSC98c].

The PVS specification language is based on Classical Higher-Order logic and specifications are defined in a functional style. The proof system is based on Gentzen Sequent Calculus LK, the *defined-rules* (or commands) are analogous to LCF-tactics, and the strategies of the tactic language are analogous to LCF-tacticals. The most powerful PVS tactic, `(grind)`, can sometimes complete a whole proof development without human assistance, but sometimes can fail due to the use of heuristic instantiation, which results in an unprovable proofstate. Therefore the `(grind)` is not safe in the LCF sense.

The definition of `(grind)` is made to conform to the normal form of proofs yielded by Theorem 3.6.3. A robust formulation of `(grind)` is introduced which uses backtracking to return the original goal formula when `(grind)` fails, thus allowing the interactive development of a proof for that goal. Thus this chapter also serves as validation for Algorithm 3.7.1, and sets PVS as the software tool for further validation of our theory for constructing and encoding robust tactics from proofs in other domains.

Section 4.2 gives an overview of PVS in terms of the specification language (Section 4.2.1), the proof system (Section 4.2.2), and the tactic language (Section 4.2.3). The encoding of safe and robust LCF-like tactics based on `(grind)` is introduced in Section 4.3. A motivating example on interacting with PVS to develop robust tactics in the normal form yielded by Algorithm 3.7.1 is given in Section 4.4.

4.2 An overview of the PVS theorem-prover

PVS [SORSC98a, SORSC98b, SORSC98c, OS97] is a general-purpose interactive theorem-prover which uses a functional classical higher-order logic specification language [Section 2.4.1], the Gentzen Sequent Calculus proof system [Section 2.5], and the Lambda Calculus [Acz98, Bar94] [Section 2.6.1.1] as the computational/evaluation mechanism for the proof system. The PVS ITP/PC is implemented in Allegro Common Lisp [Fra88, Ste84].

In comparison with other theorem-provers such as HOL [GM93], Isabelle [Pau94], Boyer-Moore [BM79], the PVS specification language has a richly expressive type system, and the interactive theorem-prover is very effective. The system also comes with a tactic language which is a subset of Allegro Common Lisp [Fra88], and in which the proof rules defined within the prover as well as additional user-defined proof rules are implemented.

4.2.1 PVS specification language

The PVS specification language is used in the *Specification* phase of the Formal Methods Lifecycle (Procedure 2.3.1). The highest specification encapsulation construct in PVS is called a THEORY, and this can contain, in the following order, formal parameters, other PVS THEORYs, assumptions, type declarations, constant declarations, variable declarations, functional definitions, and conjectures to be proved. The order of specification of constructs is important, since no statement may reference a variable or definition that has not been previously declared or defined.

The PVS Prelude file [OS03a] consists of theories that are built into PVS, which are then available for use in other user-defined specifications. For example the Prelude theory booleans:THEORY defines the type bool:NONEMPTY_TYPE = boolean; the Prelude theory defined_types[t]:THEORY defines the basic type for sets setof:TYPE = [t -> bool], where t is some type; and the Prelude theory set:THEORY defines the operators $\in$, $\subseteq$ as:

```
set: TYPE = setof[T]

x,y: VAR T

a,b,c: VAR set

member(x, a): bool = a(x)

subset?(a,b):bool = forall (x:TYPE) : member(x, a) implies member(x, b)
```

Figure 4.1 uses these Prelude definitions in the PVS specification NotSubsetProp of the conjecture $\{4\} \not\subseteq \{5, 6\}$. The theory NotSubsetProp does not include parameters, other

```
NotSubsetProp  % [ parameters ]
: THEORY
  BEGIN
  % IMPORTING
  % ASSUMING
   % assuming declarations
  % ENDASSUMING
  set:TYPE = setof[number]
  nsubset?(s1:set,s2:set):bool = NOT(subset?(s1,s2))
  NotSubsetConj: CONJECTURE nsubset?({x:number|x=4}, {y:number|y=5 OR y=6})
  END NotSubSetOfProp
```

Figure 4.1: Specification for $\{4\} \not\subseteq \{5,6\}$ in PVS.

theories nor assumptions hence the slots for these constructs are commented out using the percentage sign %—the PVS compiler ignores any statements preceded by the % sign. The type `setof[number]` is used to define the set of numbers, and `nsubset?` is the functional definition of $\not\subseteq$. The goal formula to be proved, `NotSubsetConj`, is specified with the keyword `CONJECTURE`, and in this case set comprehension is used to formulate the goal formula. Other keywords that can be used to introduce a goal formula are `THEOREM, CHALLENGE, CLAIM, COROLLARY, FACT, FORMULA, LEMMA, SUBLEMMA, PROPOSITION, LAW`. Defining the goal formula in a specification corresponds to the *Abstraction* phase in the Formal Methods Lifecycle [Procedure 2.3.1].

4.2.2 The PVS proof system

The PVS proof system is used for the *Reduction* and *Verification* phase of the Formal Methods Lifecycle [Procedure 2.3.1]. The PVS Prover system consists of a parser and a typechecker which automatically detect syntax errors and type correctness conditions (TCCs) respectively in a specification. For syntactically incorrect specifications, the parser returns useful messages which the user can use to write the PVS specification in the correct syntax. The PVS tactic `tcp` for TCCs can be invoked to automatically prove the TCC proof obligations—those TCCs that cannot be proved automatically (due to the undecidability of higher-order logic), are tagged as `unfinished` and the user is then obliged to prove the TCCs interactively; those TCCs which have been proved are tagged `proved-complete` or `proved-incomplete` (see Definition 2.4.2).

After parsing and typechecking the PVS specification, the user can then attempt to prove the goal formula(s) using PVS proof commands which correspond to the Gentzen Sequent Calculus inference-rules and decision procedures. In interactive theorem-proving and proof-checking a given goal formula, the user develops a formal proof by typing in the name of a defined-rule of the ITP/PC, which the ITP/PC then executes on the current proofstate. The defined-rule works by reducing the sequent goal formula to simpler subgoals which can be discharged automatically using the information contained in the current proofstate. Compared to other interactive theorem-provers such as HOL [GM93] and the Boyer-Moore prover [BM79], in PVS, all of the LK inference-rules are automated, and the propositional logical connectives ($\neg$, $\wedge$, $\vee$, $\Rightarrow$) are manipulated in-line where they are in scope.

There are two facets to this automation: (1) the user has to type in the name of the appropriate defined-rule, and so the execution of the defined-rule occurs only after the human-prover inputs the name of the inference-rule; and (2) the system automatically applies the rule by itself without any input from the human-prover to "tidy" up the proofstate, e.g. the inference-rules for negation, axioms, and lambda evaluation, operate in this manner. The second kind of automation is effected by the inclusion of simplification procedures in the PVS defined rules, which helps to minimize the amount of interactivity that the user may have to endure by having to input a proof command for each proofstep, even the trivial ones.

4.2.2.1 PVS proof rules

There are 26 inference rules documented in [OS97] as Gentzen Sequent Calculus proof rules, nine of which are axioms. Appendix A lists the natural deduction inference-rules for classical logic in Gentzen Sequent Calculus, and the rule for the principle of mathematical induction, which is not listed in [OS97] but is formulated as a strategy in [Sha01]. For each LK inference-rule, the name is given of the corresponding PVS defined proof-rule that is used in an interactive session with PVS. In this way hand proof steps conceptualized by the human expert user are mapped to the machine proof steps used by the PVS theorem-prover/proof-checker. Those inference rules named `auto` are invoked automatically by the PVS system as opposed to the rule being typed-in by the human user.

In addition to the implementation of the Gentzen LK inference-rules as primitive defined-rules, the PVS prover consists of other automated prover-commands known as defined-rules and strategies that execute a sequence of steps to achieve a task in a proof development exercise. These involve among others [Sha01]: rewriting goal formula (`(rewrite)`), BDD (Binary Decision Diagram)-based Boolean simplification (`(bddsimp)`), the arithmetic and equality decision procedures (`(assert)`), model-checking (`(model-check)`), and the most powerful PVS proof command `(grind)` which can be invoked by a novice user to attempt to prove a goal formula automatically in PVS. These strategies are defined using the PVS tactic language.

4.2.3 PVS tactic language

The PVS tactic language is for the *Generalisation and Maintenance* phase of the Formal Methods Lifecycle (Procedure 2.3.1). The tactic language enables the user to define more powerful proof-rules (which correspond to LCF-like tactics), by using predefined proof strategies (which correspond to LCF-like tacticals) to compose proof-rules together. The PVS tactic language is a subset of Allegro Common Lisp, which is the language in which the PVS system is implemented.

4.2.3.1 A synopsis of the implementation of tactics

The PVS tactic language consists of 16 tacticals [SORSC98a]. PVS tacticals can only combine proof rules sequentially since there are no strategies for parallel composition of tactics in PVS at present, probably due to the lack of a parallel programming construct in Common Lisp [Fra88, Ste84].

The syntax for a strategy expression (or tactical) in PVS is:

Definition 4.2.1. `(PVS tactic language)` [SORSC98b]: ⟨*step*⟩ :=

⟨*primitive-step*⟩ | %LCF-like LK inference-rules

⟨*defined-rule*⟩ | %LCF-like composite tactics

⟨*defined-strategy*⟩ | %LCF-like tacticals

`(quote` ⟨*step*⟩`)` | %Identity strategy

`(try` ⟨*step*1⟩⟨*step*2⟩⟨*step*3⟩`)` | %Backtracking and subgoaling tactic

`(if` ⟨*lisp-expression*⟩⟨*step*⟩⟨*step*⟩`)` | %Conditional tactic

`(let` $(\{(\langle symbol\rangle\langle lisp\text{-}expression\rangle)\}^+)$⟨*step*⟩`)` | %Evaluates/bind Lisp expression/values

`(branch` *step steplist*`)` | %assigning strategies to subgoals

`(else` *step*1 *step*2`)` | %Simple Backtracking Strategy

`(query*)` | %Basic Interaction Strategy

`(repeat` *step*`)` | %Iterate Along Main Proof Branch; `(repeat*` *step*`)` for all branches

`(rerun &` *OPTIONAL proof*`)` | %Rerun a Proof or Partial Proof

`(spread` *step steplist*`)` | %Assigning Strategies to Subgoals; variations are `(spread@)`, `(spread!)`

`(then &` *REST steps*`)` | %Sequencing Strategy `(time` *strategy*`)` | %Time a Given Strategy

`(try-branch` *step*1 *steplist* *step*2`)` %Branch or Break

The core of PVS tacticals are `(quote)`, `(try)`, `(if)`, `(let)`—all the other tacticals can be defined in terms of these four. The syntax for a PVS defined rule (or tactic) is as follows:

Definition 4.2.2. (PVS `defined-rule form`) [SORSC98b]: A PVS defined-rule has the form:

`(defstep` *name*

(*required-parameters* &optional *optional-parameters* &rest *parameters*)

strategy-expression

documentation-string

format-string) □

Definition 4.2.2 defines a *(blackbox) defined-rule*, `name` and a *(glassbox) strategy* `name$`, i.e. if the user appends the dollar sign to the name of the defined rule on invocation at the PVS prover prompt, the resulting proof is a trace of each of the proof-rules in *strategy-expression*, otherwise the prover just returns the resulting subgoal(s).

Alternatives to the definition form `defstep` are `defhelper` which defines strategies that are only meant to be used in the definition of other strategies and are not likely to be invoked by the user directly; and `defstrat` which defines only a glassbox strategy *name*, but not the blackbox version, and does not use the final *format-string* argument given in `defstep`. The differences and similarities between PVS defined-rules and strategies is summarized in Table 4.1 [SORSC98c].

Defined rule	Strategy
Analogous to LCF tactics	Analogous to LCF tacticals
Named and invoked as (*name*), e.g. (prop).	Named and invoked as (*name*$), e.g. (prop$).
Is atomic like a primitive rule (LK inference-rules) when invoked	Can expand to the application of several atomic rules when invoked.
Has blackbox behaviour—internal behaviour is not visible to the user.	Has glassbox behaviour—internal behaviour is visible to the user.
Saved and rerun in its expanded form	Only the expanded form is saved to be rerun.
Returns the unproven subgoals	Returns the expanded proof tree.
Can be recursive and involve the application of a number of primitive proof steps to achieve an effect.	Same.

Table 4.1: Defined rules and strategies

4.3 Encoding LCF-like tactics in PVS

The most powerful tactic in PVS, (grind) is defined as follows:

Definition 4.3.1. PVS grind tactic [Sha01]: The PVS super duper strategy (grind) is defined as follows:

```
(defstep grind (&optional (defs !); NIL, T, !, explicit, or explicit!
                theories rewrites exclude (if-match T) (updates? T)
                polarity? (instantiator inst?))
  (then (install-rewrites$ :defs defs :theories theories
                           :rewrites rewrites :exclude exclude)
   (then (bddsimp)(assert))
   (replace*)
   (reduce$ :if-match if-match :updates? updates? :polarity? polarity?
            :instantiator instantiator))
   "A super-duper strategy ...")
```

However the PVS (grind) (Definition 4.3.1) can sometimes yield an unprovable proofstate if the proof involves instantiation. The PVS tactic (grind) works by first unfolding all the definitions in the goal (i.e. (install-rewrites$)); simplifying all expressions (i.e. (bddsimp) (assert)); replacing the old expressions with the simpler ones (i.e. (replace*)); and finally reducing the proofstate using the LK inference rules and decision procedures ((reduce$)). This (reduce$) tactic invokes the PVS defined-rule for instantiation via the argument ((instantiator inst?)). The automatic instantiator (inst?) is defined to heuristically find a suitable instantiation term by simply matching the type of the variable to be instantiated with the types of the skolem variables and/or other terms in the current proofstate. It is this heuristic which can lead to instantiation with an incorrect term thus yielding incorrect and unprovable subgoals; or the heuristic may fail to find any instantiation term at all. Thus the PVS inst? is not a safe LCF-like tactic (see Chapter 2, Section 2.6.1), in that its application may result in a false proofstate.

Nevertheless, the definition of grind conforms to the normal form of proof given by Theorem 3.6.3. The proofsteps (then(install-rewrite$)(then(bddsimp)(assert))) can be seen as the rewrite phase in Definition 3.9.1; or as a form of *Cut*-rule application to introduce normal forms of the definitions in the goal formula to be proved in Definition 3.6.1; or as the reduction phase in the Formal methods lifecycle Definition 2.3.1 to reduce the formula to be proved to a form where LK inference-rules can be applied by the (reduce$) tactic. Once the rewrite phase is complete, by Theorem 3.6.3, the subgoal generated is now in the form where an instantiation can be applied (if required) by tactic (reduce$). Theorem 3.6.1 can be used to generate instantiation terms from skolem variables where an instantiation is required. By Theorem 3.6.3, the resulting subgoal is a quantifier-free formula, on which only the mechanical inference-rules can be applied recursively by the tactic (reduce$) to complete the proof automatically.

Since (grind) can sometimes find the correct instantiation automatically, as well as complete a whole proof development by itself, it is persuasive to try the (grind) tactic as the first attempt at a proof development in PVS. If (grind) fails to prove the conjecture, then backtracking can be used to return to the proofstate just before the automatic instantiation invoked by (inst?), and then the user can instantiate the proofstate manually.

4.3.1 Encoding robust tactics in PVS

Therefore in accordance with the instantiation proof plan (Definition 3.6.1), and the idea to try (grind) first in a proof, Definition 4.3.2 describes a template for encoding robust tactics in PVS as follows:

Definition 4.3.2. (RobustGrind): In proving a goal formula, (grind$) can be tried first, and if (grind$) fails to find a proof, then backtrack to the proofstate just before an instantiation to enable manual instantiation by the user to reduce the proofstate to Propositional logic which is decidable; and thus completion of that proofstate by the decision procedures of Propositional Logic.

```
(defstep RobustGrind (&optional (defs !))
   (try (try (grind)(fail)(skip))
        (skip)
        (grind :defs defs :if-match nil))
   "tries (grind) on the goal formula, and if (grind) fails,
   perform creative proofsteps first to reduce proofstate to
   Propositional Logic for decision procedures to complete proof.")
```

The robustness of Definition 4.3.2 is argued for as follows. The tactic `(try (grind)(fail)(skip))` enables `(grind)` to try to find a proof, and if `(grind)` produces any subgoals, the backtracking tactic `(fail)` returns the original goal formula because the resultant subgoals my be unprovable due to incorrect instantiation(s) invoked by `(grind)`. The outermost `(try)` tactical then invokes `(grind :defs defs :if-match nil)` which simplifies the original goal without automatic instantiation (to avoid incorrect instantiations). If the actual parameter for definitions (`defs`) is given as `nil`, then the definitions in the goal are not rewritten (to avoid overwhelming the user with too much detail). This simplification yields one or more subgoals, each of which can be discharged by the user performing the creative proofsteps (Cut (`(case)`), Instantiation (`(inst)`), Induction (`(induct)`)) to reduce the proofstate to Propositional Logic to enable the completion of the proofstate by the decision procedures for propositional logic and arithmetic in `(grind)`.

Theorem 3.6.1 yields a method by which suitable instantiation terms may be arrived at or generated. In particular, the user can code the function f to generate suitable instantiation terms for the proofstate—this is used in Chapter 5 where instantiation terms are generated from constructive definitions of the operations in abstract programs. Alternatively an ATP may be used to find the instantiation terms, which can then be fed back into proof state to complete the proof [Section 2.3.2.1]. The ingenuity in choosing a suitable cut formula may assisted by considering a cut as a transitive relation $((A \Rightarrow B) \wedge (B \Rightarrow C)) \Rightarrow (A \Rightarrow C)$, where B is the cut formula, and $A \Rightarrow C$ is the goal/conjecture to be proved $A \vdash C$ [Section 2.5.2.1]. Note that the antecedent part A may be empty, e.g. Definition 3.6.1 encodes the transitive relation $(\vdash g_{PNF}) \wedge (g_{PNF} \vdash g) \Rightarrow (\vdash g)$, where g_{PNF} is the Cut formula, and $\vdash g$ is the original goal. Given a suitable induction variable, the PVS tactics for induction can generate the induction hypotheses automatically, or the user can define their own induction scheme, e.g. using Bundy's induction proof plan [Section 2.7.2].

4.4 Interacting with the PVS proof system

The theory `NotSubsetProp.pvs` in Section 4.2.1 is proffered to the PVS proof system by opening the specification in PVS. To prove the conjecture `NotSubsetConj`, the mouse cursor is clicked on that definition, and then the PVS prover is invoked by clicking on `PVS` on the Emacs menu bar, selecting `Prover Invocation`, and then clicking on `Prove`. User-defined tactics saved in the PVS file called `pvs-strategies` are automatically loaded and the Conjecture is made available for the user to start inputting proof commands.

The user types in the name of a proof command (i.e. a tactic) at the `Rule?` prompt, and on pressing the Keyboard Enter key, PVS executes the proof command. If the proof command is applicable to the current proofstate, new subgoal(s) are generated from the current goal formula; otherwise if the proof command is not applicable, then the proof state does not change (this is akin to what happens in tactic application described in Chapter 2, Section 2.6.1).

The PVS system uses the `time` strategy to record the time taken in a proof development. The `Run time` refers to the time the PVS system takes to execute proof rules typed interactively into the system by a human use; the `Real time` refers to the actual time the user spends on the proof development process. Since the `Real time` is dependent on the human-user, the `Run time` is a more accurate measure of the improvement on efficiency that can be afforded by the application of Algorithm 3.7.1 when the normal form of proof yielded by the algorithm is used in place of the interactively developed proof using the

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{4=5\vdash}\,[\bot\vdash]\texttt{(assert)}\qquad\dfrac{}{4=6\vdash}\,[\bot\vdash]\texttt{(assert)}}{4=5\vee 4=6\vdash}\,[\vee\vdash]\texttt{(split)}}{\forall(4\in\{4\}\Rightarrow 4\in\{5,6\})\vdash}\,[\in D]\texttt{(expand "member")}}{\{4\}\subseteq\{5,6\}\vdash}\,[\subseteq D]\texttt{(expand "subset?")}}{\vdash\{4\}\not\subseteq\{5,6\}}\,[\not\subseteq D]\texttt{(expand "nsubseteq")}$$

Figure 4.2: The prooftree from the application of Algorithm 3.7.1

The tactic is defined as follows in PVS

```
(defstep NotSubsetOfTac (fnum &rest terms)
  (then (expand* "nsubseteq" "subset?" "member")
        (inst fnum terms) (split) (assert) (assert))
  "INPUT = NOT(subset?(x,y))
   OUTPUT = true"
  "proving NotSubsetConj ...")
```

strategy Definition 3.2.1.

4.4.1 An application of Algorithm 3.7.1 in PVS

The proof of conjecture `NotSubsetConj` can be developed interactively using the strategy Definition 3.2.1, which yields a formal proof consisting of the sequence of proof steps `((expand "nsubset?") (expand "subset?") (inst?) (expand "member"))`. The rest of the proof consists of propositional and arithmetic simplification, which is completed automatically by the PVS proof system. In this case PVS is able to correctly automatically instantiate `((inst?))` with the constant 4. An interactive step by step proof, i.e. with the user typing each primitive proof rule separately at the `Rule?` prompt, one after the system executes another, takes approximately 0.13 seconds `Run time` and 8.41 seconds `Real time`.

The prooftree yielded by the application of Algorithm 3.7.1 on the interactive proof steps `((expand "nsubset?")(expand "subset?")(inst?)(expand "member"))` above is shown in Figure 4.2, which conforms to the normal form of the Instantiation Proof Plan (Definition 3.9.1). The proofsteps [$\vdash \neg$] and [λ *setD*] are invoked automatically by the PVS proof system without the user having to type in any explicit commands. Thus it appears expanding the definition of set-membership, [$\in D$], makes redundant the instantiation step `inst?` in the interactively developed proof—PVS performs this step automatically without prompting the user. The tactic resulting from the application of Algorithm 3.7.1 on the interactive proof was coded in PVS, and applied on the conjecture—this proof took a `Run time` of 0.05s. Thus the human proof strategy took almost 3 times as much time as the tactic developed using our method. The `Real time` taken by a user in applying the strategy Definition 3.2.1 is just under a minute. Thus tactics are very useful for reducing the time taken to develop a proof, as well as the amount of interaction the user has to engage with the proof system.

Since PVS automatically finds a correct instantiation in the proof above, PVS's most powerful tactic `(grind)` completes the proof successfully in 0.08s. However in the more complex examples, such as those for the proof of the refinement and retrenchment proof obligations in the following chapters, `(grind)` is not able to find a correct instantiation; the human-prover has to use domain specific knowledge to perform an instantiation manually.

4.5 Summary

This chapter has demonstrated the application of the Theory developed in Chapter 3 to encode robust tactics in the PVS ITP/PC. PVS is a general-purpose theorem prover with a functional classical higher-order logic specification language, a Gentzen Sequent Calculus proof system, a rewrite engine based on the Lambda Calculus, and a tactic language which is a subset of Allegro Common Lisp.

PVS defined proof rules are analogous to LCF tactics and PVS strategies are analogous to LCF tacticals [Table 4.1]. However the most powerful tactic in PVS `(grind)` can sometimes automatically instantiate incorrectly due to the heuristic nature of the PVS `(inst?)` tactic which uses pattern matching to decide possible instantiation terms from the proofstate. Therefore the `(grind)` and `(inst?)` tactics are not safe in that they may produce a false proof state.

The instantiation proof plan [Definition 3.9.1] can be used to make the task of finding manual instantiations easier for the human-user by applying the necessary reduction steps to transform the goal formula, e.g. in a goal formula involving definitions, unfolding the definitions first may facilitate the task of finding the instantiation term automatically [Section 3.3.1].

The example application in Section 4.4.1 shows that Algorithm 3.7.1 can work in PVS to reduce the number of proofsteps the user has to enter and thus to reduce the time taken to perform a proof, which makes interactive proof development more efficient for both the human user and the ITP/PC. In the next chapters Algorithm 3.7.1 is validated on a more complex example, that of constructing tactics for the Retrenchment method in PVS.

Chapter 5

Architectural retrenchment

"As assumptions are strengthened in the Retrenchment Proof Obligation towards refinement, the models become more refinement like." [Pop01].

In this chapter RobustGrind (Definition 4.3.2) is applied to derive manageable subgoals and robust tactics for Architectural Retrenchment, which is our decomposition of the Banach-Poppleton *simple retrenchment method* [BP99a, Ban98], in order to highlight the modest changes in specification. Some of the material in this chapter has been previously published in [HG01].

5.1 Introduction

We noted in the Chapter 3 that the successful application of a tactic on a new conjecture depends on whether the structure of the original conjecture, from whose proof the tactic was constructed, is the similar to the structure of the new conjecture to be proved. An example of structured conjectures in formal software development are the *refinement* and *retrenchment* proof obligations for specification development [BP99b], given in Figure 5.1.

We explore the derivation of tactics for retrenchment in PVS, using a numerical example of a B machine adapted from [Pop01]. The example involves the retrenchment of subtraction in the infinite reals domain to subtraction in reals that are bounded above by some biggest expressible real *MaxReal*. The purpose of this example is to further validate our proposition for the construction of robust tactics, as well as derive tactics for the issue of overflow in computer arithmetic. Section 5.3 describes the nature of this retrenchment in the B-method. Section 5.4 looks at the specification of the retrenchment in PVS—a widely-used specification and verification system with ample facilities for constructing proof tactics. Section 5.5 describes the derivation of tactics from proofs of the retrenchment proof obligations and describes a general tactic for proving retrenchment proof obligations in PVS. A further abstraction to derive a general tactic that can automatically prove any of the proof obligations in this example is described in Section 5.5.4. Section 5.6 discusses the merits and limitations of our approach and points to further work.

5.2 Approaches to formal program development

Formal program development entails the application of formal methods in the development of software programs, and this has been advocated for critical system construction [NASb, NASa]. Program synthesis and refinement are well known formal program development techniques. Retrenchment is a liberalization of refinement that has been proposed as a formal development step prior to applying refinement. In particular, retrenchment enables the construction of a concrete specification from an abstract one. Concrete specifications are then implemented using a refinement calculus [BP99b].

5.2.1 Program synthesis

Program synthesis deals with aspects of the software development process which can, at least in principle, be automated [Kre98] and thus this provides software tools for use in automated software engineering. Research in this area focusses on (1) the development of expressive calculi which support formal reasoning about specifications and algorithms, e.g. *Abstract State Machine Notation* [Abr96], *Specification Statement* [Mor88], *Constructive Type Theories* [Mar84, CAB$^+$86, GLT89, CH88, NPS90]; and (2) the implementation of deductive strategies which derive programs from their specifications, e.g. *Hoare Logic* [Hoa69], *Morgan's Refinement Laws* [Mor94], *Refinement Calculus* [Bac80].

There are three approaches to automated program synthesis. The *Proofs-as-Programs* approach [Gre69, MW80, BC85] sees the derivation of a program as a proof of the statement $\forall(x : D) : \exists(z : R) : (I(x) \Rightarrow O(x, z))$ [Kre98] which implicitly constructs a program (see Section 2.7.3). This appears to be the most popular approach, e.g. the NuPRL system. *Transformational Synthesis* approach [MW75, BD77] uses general rewriting techniques in order to transform a specification into a form that can straightforwardly be converted into a program. The *Knowledge-Based Program Synthesis* approach [Smi85, Dol95] involves the analysis of algorithmic classes and strategies in order to derive the parameters of program schemata and instantiate them accordingly. Originally, approaches to program synthesis were developed in the framework of automated deduction.

5.2.2 Program refinement

Definition 5.2.1. (Refinement) [Pop01]: A refinement, $S_0 \sqsubseteq S_1$, [1] is "...a correctness preserving transformation...between possibly abstract, non-executable programs which is

[1] $S_0 \sqsubseteq S_n$ means S_0 is refined by S_1.

transitive, thus supporting stepwise refinement, and is monotonic with respect to program constructors, thus supporting piecewise refinement." $\square$

Stepwise refinement allows one to move gradually from a specification to an implementation and this is possible because refinement is reflexive and transitive.

Definition 5.2.2. (Stepwise refinement) [Abr96]: If $S_0 \sqsubseteq S_1 \sqsubseteq \ldots \sqsubseteq S_n$ then $S_0 \sqsubseteq S_n$. $\square$

Piecewise refinement enables compositional/top-down development [Ros98] and this is possible because refinement is monotonic.

Definition 5.2.3. (Piecewise refinement) [Abr96]: If $S[T]$ is a program containing a subprogram or statement T then $T \sqsubseteq T' \Rightarrow S[T] \sqsubseteq S[T']$. $\square$

There are two formal styles of refinement, both of which support piecewise and stepwise refinement [But98]. In the Invent-and-Verify (Posit-and-Prove) style, a refined specification R is developed intuitively and then checked against the original specification S via the standard refinement relation $S \sqsubseteq R \Leftrightarrow R \subseteq S$ [WD96], i.e. some of the functionality that is possible in the specification S may not be possible in the implementation R. Methods that support this technique include the *B-Method* [Abr96], *VDM* [Jon86], *Z* [WD96].

Transformational (Strict) refinement applies a refinement preserving transformation rule to all or part of the current specification to produce a refined specification automatically. Methods that support this technique include the refinement calculus for imperative programs [Bac80, Mor94, Mor97] and the transformational design approach for functional programs [BD77]. The refinement relation in this case is $S \sqsubseteq R \Leftrightarrow S \Leftrightarrow R$, i.e. everything that the specification S does should be possible in the implementation R.

The standard (transformational) refinement calculi of Back [Bac88], von Wright [vW94], Morgan [Mor94], and Morris [Mor97] are a formalization of Wirth's stepwise program development method [Wir71] and Dijkstra's weakest precondition semantics approach [Dij75]. The *B-Method* has been interpreted in standard refinement calculus [Rou99], and one conclusion of this interpretation was that the improper use of the B-Method *SEES* clause can lead to unsafe programs. On a similar token, some inconsistencies have been found in Morgan's refinement laws [CSJ99].

One of the drawbacks to the uptake of formal methods in industry, is the radical *revolution* that formal methods imposes on the software development process [Bro87]. The invent-and-verify refinement technique mirrors conventional programming practice albeit verification is used instead of informal testing/debugging techniques. A major criticism of transformational refinement is that it is too strict—some informal steps usually have to be taken when applying refinement in program development.

5.2.3 Retrenchment

Retrenchment is a liberalization of the strict transformational refinement calculus to enable the development of realistic specifications from idealized specifications using the invent and verify approach. Some of the strict transformation steps produce verification conditions (proof obligations) which must be discharged for the transformational refinement to be valid, and these proof obligations have been encapsulated as the retrenchment proof obligations defined in Figure 5.1 [Pop01] [2]. In particular, the retrenchment method can be used to formalize and reason about the relationship between an idealized (also known as divine, abstract) specification and its realizable (also known as mundane, concrete) representation, which can then be successfully implemented by using the transformational refinement method.

[2] An earlier criticism of this method was that it did not have a mathematical basis [Smi99].

Definition 5.2.4. (Retrenchment) [BP99b]: The Retrenchment method is governed by the refinement initialization and invariant preservation proof obligations, and the retrenchment proof obligations shown in Figure 5.1. □

The variables used in a program specification are distinguished into state u, v, input i, j and output o, p for the abstract and concrete machines respectively, and the logical variables A which hold before-values of the state and input variables so that they may be referred to in the after-state if necessary. The refinement concrete invariant $J(u, v)$ is separated into the retrieves relation $G(u, v)$ and the concrete invariant $J(v)$. The retrenchment initialization proof obligation requires satisfaction of the retrieves relation, and the operation retrenchment proof obligation requires satisfaction of the retrieves relation or the concedes relation $C(u, v, o, p, A)$ which relates state and logical variables in the after-state thus weakening the postcondition. The conjunct $P(i, j, u, v, A)$ is the *WITHIN* clause which relates state variables, input, and logical variables A, in the before-state thus strengthening the precondition.

The initialization and invariant preservation proof obligations are the same as those for the B-refinement method, but the retrenchment initialization and the retrenchment operation proof obligations differ from those corresponding to the B-refinement method. The operation-retrenchment proof obligation has the termination condition of the realistic machine, $trm(T)(v, j)$, in the antecedent whereas the operation-refinement proof obligation has the termination condition of the idealistic machine, $trm(S)(u, i)$, in the antecedent. Thus for retrenchment, it is assumed that the realistic machine terminates and it has to be proved that the idealistic machine terminates. Since the idealistic machine is the basis of the development, it has more merit to prove everything about the idealistic machine, in particular to prove that it terminates rather than to assume that it terminates. The

- Initialization proof obligations: the initialization of each machine (i.e. the before-state) must satisfy the invariant.
  $\underline{[X(u)]I(u)}$
  $\underline{[Y(v)]J(v)}$
- Invariant preservation proof obligations: if the before-state satisfies the invariants and the termination conditions then the after-state given by the operations shall satisfy the invariants.
  $I(u) \wedge trm(S(u,i)) \Rightarrow \underline{[S(u,i,o)]I(u)}$
  $J(v) \wedge trm(T(v,j)) \Rightarrow \underline{[T(v,j,p)]J(v)}$
- Retrenchment initialization PO: the initializations of the two machines must satisfy the retrieves relation.
  $\underline{\underline{[Y(v)]\neg[X(u)]\neg G(u,v)}}$
- Operation retrenchment PO: if the before-state satisfies the invariants and the concrete termination condition, then the abstract termination condition shall be satisfied, and there shall not be a situation where the after-states of the concrete and abstract operations should not satisfy the retrieves or concedes relations.
  $(I(u) \wedge G(u,v) \wedge J(v) \wedge trm(T)(v,j) \wedge P(i,j,u,v,A))$
  $\Rightarrow (trm(S)(u,i) \wedge \underline{\underline{[T(v,j,p)]\neg[S(u,i,o)]\neg(G(u,v) \vee C(u,v,o,p,A))}})$

Figure 5.1: The retrenchment proof obligations.

operation retrenchment proof obligation enables to reason about whether a retrenchment preserves the retrieve relation $G(u,v)$—we call this a *subrefinement*—or on failing to do so gives an acceptable degradation of service depicted by the concession $(C(u,v,o,p,A))$.

Retrenchment is particularly suited to reasoning about systems involving continuous variables such as hybrid systems, and for changes in the systems architecture [Pop01], e.g. there is no formalization of the real datatype in the B-Method [Abr96], and refinement in the B-Method is only defined for machines with operations of the same signature [HG01]. Various notions of retrenchment have been proposed, e.g. simple retrenchment [BP99a, Ban98] which is described in Figure 5.1; sharp retrenchment [BP99b, Pop01] which strengthens the retrieves clause with the nevertheless clause [3], and evolving retrenchment [PB02] in which the retrieve clause G becomes variant [4]. We are concerned with unsharp retrenchment in this chapter; the next chapter deals with a form of evolving retrenchment. In [Pop01], the Retrenchment method is formalized using the B-Method Generalized Substitution Language [Abr96], and the provision of software tools akin to the linking of Automatic Theorem Provers with Computer Algebra Systems [HT93, BJ01] is advocated in order to reason about discrete and continuous datatypes, i.e. floats and reals respectively.

5.2.3.1 Patterns in the retrenchment proof obligations

The initialization proof obligations have the same structure as the implied conclusions of the invariant proof obligations, and the retrenchment initialization proof obligation has the same structure as the second conjunct implied in the operation retrenchment proof obligation. This is depicted by the underlines in Figure 5.1. However, the definitions of the constructs of the proof obligations, i.e. $I(U)$, $G(u,v)$, $J(v)$ etc, may differ from specification to specification according to what is being specified.

Thus for a particular retrenchment, the Tactic Refinement method (Definition 3.3.1 and Theorem 3.6.3) can be applied for the retrenchment proof obligations defined in the order depicted in Figure 5.1. Since these proof obligations govern the whole retrenchment, the robust tactic formulated for the operation retrenchment proof obligation should be able to prove other retrenchments in the same application domain.

[3]The nevertheless clause $V(u,v,o,p,A)$ describes nontrivial relationships between the idealized and realizable after-state variables u, v and the idealized an realizable output variables o, p and the logical variables A. That is the after-state predicate becomes $((G(u,v) \vee C(u,v,o,p,A)) \wedge V(u,v,o,p,A))$.

[4]G becomes mediated with some precision parameters α and β and it is usually (but not always) expected that $\alpha \leq \beta \not\Rightarrow (G_\alpha \Rightarrow G_\beta)$

5.3 Specification and proof of retrenchment in B

Consider an idealistic B machine, *DSub* with state variables a, b, and an operation S which subtracts b from a and puts the result in a, provided $a \geq b$. The realistic machine *MSub* has a state variable aa, and a corresponding operation T which subtracts an input bb from aa and puts the result in aa but does so only for $aa < Ov$ and $bb < Ov$, where Ov is some threshold value that the variables should not exceed. In addition, *MSub* gives an output *resp* to signal whether or not the subtraction has been successfully—*ok* means that $aa := aa - bb$, whereas *fail* means that the subtraction was not possible for the values of aa and bb given to *MSub*, i.e. there was an overflow. This retrenchment of *DSub* by *MSub* is shown in Figure B.1 in Appendix B. This is the notion of retrenchment formulated by Banach and Poppleton [Pop01], and we call it a *Banach-Poppleton retrenchment* or *BPRet*.

5.3.1 Architectural retrenchment

Looking at the retrenchment *DSub* by *MSub* in Figure B.1, we can see three components of two different forms of the retrenchment of *DSub* as summarized in Figure B.1: (1) input-architectural retrenchment *IAMach*; (2) data representation retrenchment *DRMach*; and (3) output-architecture retrenchment *OAMach*. Note that for concise specification, the state, input and output variables $u0, u1, u2, u3, i1, i1, i3, o3,$ *etc*, are specified as a record structure, whose components are the individual variables of the machine. For example, with reference to the *DSub* retrenchment shown in figure B.1, the type of $U0$ is a record $[\#a : \mathbb{R}, b : \mathbb{R}\#]$, and $u0$ is a variable of this type [5].

[5] $[\#x : X, y : Y, ...\#]$ is the PVS syntax for defining record types.

MACHINE	*DMach*	*IAMach*	*DRMach*	*OAMach*
RETRENCHES		*DMach*	*IAMach*	*DRMach*
VARIABLES	$u0 : U0$	$u1 : U1;\ \ i1 : I1$	$u2 : U2;\ \ i1 : I2$	$u3 : U3;\ \ i3 : I3$
INVARIANT	$inv(u0)$	$inv(u1)$	$inv(u2)$	$inv(u3)$
RETRIEVES		$G(u0, u1)$	$G(u1, u2)$	$G(u2, u3)$
INIT	$init(u0)$	$init(u1)$	$init(u2)$	$init(u3)$
OPERATIONS	$S0(u0)$	$S1(u1)(i1)$	$S2(u2)(i2)$	$o3 \longleftarrow S3(u3)(i3)$
LVAR		$A1$	$A2$	$A3$
WITHIN		$W(u0, A1, u1, i1)$	$W(u0, A1, u1, A2, u2, i, i2)$	$W(u0, A1, u1, A2, u2, A3, u3, i1, 12, i3)$
CONCEDES		$C(u0', u1', A)$	$C(u1', u2')$	$C(u2', u3', A3)$

Figure 5.2: Architectural retrenchment in B

5.3.1.1 The nature of an architectural change

In this architecture, state variables are moved to input, and/or output variables are introduced to indicate exceptional behaviour thus resulting in a change of operation signature which refinement cannot deal with. For an architectural change to occur, we need to characterize the architectural change, and to alter the *RETRIEVES*, *LVAR*, *WITHIN*, and *CONCEDES* clauses to reflect this.

From Figure 5.2 we see that the input architecture retrenchment is achieved as follows:

- Transform some state variables to input variables, e.g. in the retrenchment of *DSub* by *IASub*, state variable b changes to input variable bb. The architecture (signature) of the function changes from $S0 : U0 \rightarrow U0$ to $S1 : U1 \rightarrow (I1 \rightarrow U1)$, where $U1 = U0 \backslash I1$.

- Ensure that the input variable is of the appropriate type for the operation $S1$ to be type-correct, e.g. $bb \in \mathbb{R}$. This condition becomes part of the precondition to the operation $S1$.

- Change the individual clauses as follows. The *RETRIEVES* clause is an identity relation between the state variables $u0$ of the idealistic machine *DMach* and the state variables $u1$ of the more realistic machine *IAMach*. The *LVAR* clause remains empty, since the variables are of the same type hence everything that is possible in *DMach* should be possible in *IAMach*. The *WITHIN* clause is an identity relation between the new input variables $i1$ and the state variables in $u0$ these input variables correspond to. The *CONCEDES* clause becomes *false*, i.e. we do not envisage a situation in which the *RETRIEVES* relation cannot be satisfied in this case since the variables of both machines are of the same type.

The output architectural change is achieved as follows:

- Alter the signature of the operation $S2$ to introduce output variables $o3$, i.e. the signature of the operation becomes $o3 \longleftarrow S3(u3, i3)$, which is equivalent to $S3 : U3 \rightarrow (I3 \rightarrow (U3, O3))$.

- The *RETRIEVES* is the same identity relation between $u2$ and $u3$. The *LVAR* clause declares the logical variables $A3$. The *WITHIN* clause gives values to $A3$ and $i3$ in terms of the *OAMach* and *DRMach* respectively. The *CONCEDES* clause defines a weaker postcondition to that described by *RETRIEVES*.

5.3.1.2 The nature of a datatype change

Here, an idealistic datatype is implemented as a realistic datatype type—the infinite reals $\mathbb{R}$, becomes the finite reals $\mathbb{FR}$. For a datatype change to occur we need to identify variables whose types may change, and for each such variable, we need to characterize the type change(s), i.e. to define the relationship between idealistic and realistic variables and alter the *RETRIEVE*, *LVAR*, *WITHIN*, and *CONCEDES* clauses to reflect this. From Figure 5.2 above we see that the data change is as follows:

- The type of the state variables changes from $u1 : U1$ to $u2 : U2$.
- This datatype change is described by the retrieves relation. For example the relationship between infinite reals and finite reals in Figure B.1 is defined by $R(x : \mathbb{R}, y : \mathbb{FR}) = (y < Ov \Rightarrow x = y) \wedge (y = Ov \Rightarrow x \geq Ov)$.
- The "structure" of the operation remains the same, i.e. $S2 : U2 \rightarrow (I2 \rightarrow U2)$ has the same structure as $S1 : U1 \rightarrow (I1 \rightarrow U2)$, but the signatures are different in the sense that the datatypes $U2, I2$ are different from those of $U1, I1$ respectively.
- The changes to the individual clauses are that the *RETRIEVES* becomes the predicate that relates the two datatypes. The *LVAR* clause declares the logical variables $A2$. The *WITHIN* clause gives values to the input variables $i2$ and the logical variables $A2$ in terms of the variables of *IAMach* and *DRMach* respectively. The *CONCEDES* clause defines a weaker postcondition in terms of the state variables of the *IAMach* and *DRMach*.

5.3.2 Proof procedure for architectural retrenchment

Each of the above architectural transformations is a retrenchment in its own right, therefore we can use the retrenchment proof obligations to reason about the correctness of each of these transformations. We can unfold a Banach-Poppleton retrenchment as an input-architecture retrenchment followed by a data-representation retrenchment, and finally by an output-architecture retrenchment:

$$\begin{tikzcd} \mathit{DMach} \arrow[r, "\mathit{BPRet}"] \arrow[d, "\mathit{IARet}"'] & \mathit{OAMach} \\ \mathit{IAMach} \arrow[r, "\mathit{DRRet}"'] & \mathit{DRMach} \arrow[u, "\mathit{OARet}"'] \end{tikzcd}$$

This requires us to prove the following vertical composition or transitivity result:

Theorem 5.3.1. *(Architectural Retrenchment)*

$\mathit{DMach} \lesssim \mathit{IAMach} \lesssim \mathit{DRMach} \lesssim \mathit{OAMach} \Rightarrow \mathit{DMach} \lesssim \mathit{OAMach}$

Proof:

$\mathit{DMach} \lesssim \mathit{IAMach} \lesssim \mathit{DRMach} \lesssim \mathit{OAMach}$

$\Leftrightarrow$ *%By Theorem RetCompGS$_w$ [Pop01],Chapter 6, page 107.%*

$inv(u0) \wedge \exists\, u1 \bullet (G(u0, u1) \wedge inv(u1) \wedge G(u1, u2)) \wedge inv(u2) \wedge trm(S2(u2, i2))$

$\wedge\, \exists\, u1, i1, A1 \bullet (G(u0, u1) \wedge inv(u1) \wedge G(u1, u2) \wedge W(i1, u0, u1, A1)$

$\wedge\, W(i1, i2, u1, u2, A2)) \Rightarrow trm(S0(u0))$

$\wedge\, [S2(u2, i2)]\neg[S0(u0)]\neg\,\exists\, u1 \bullet (G(u0, u1) \wedge inv(u1) \wedge G(u1, u2))$

$\vee\, \exists\, u1 \bullet (G(u0, u1) \wedge C(u1, u2, A2))$

$\vee\, \exists\, u1, A1 \bullet (C(u0, u1, A1) \wedge G(u1, u2))$

$\vee\, \exists\, u1, A1 \bullet (C(u0, u1, A1) \wedge C(u1, u2, A2)) \lesssim \mathit{OAMach}$

$\Leftrightarrow$ *%By Theorem RetAssoc$_w$ [Pop01], Chapter 6, page 111.%*

$inv(u0) \wedge \exists\, u1, u2 \bullet (G(u0, u1) \wedge inv(u1) \wedge G(u1, u2) \wedge inv(u2)$

$\wedge\, G(u2, u3)) \wedge inv(u3) \wedge trm(S3(u3, i3)) \wedge \exists\, u1, u2, i1, i2, A1, A2 \bullet$

$(G(u0, u1) \wedge inv(u1) \wedge G(u1, u2) \wedge inv(u2) \wedge G(u2, u3) \wedge W(i1, u1, i1, A1)$

$\wedge\, W(i1, i2, u1, u2, A2) \wedge W(i2, i3, u2, u3, A3)) \Rightarrow trm(S0(u0))$

$\wedge\, [S3(u3, i3, o3)]\neg[S0(u0)]\neg(\exists\, u1, u2 \bullet (G(u0, u1) \wedge inv(u1)$

$\wedge\, G(u1, u2) \wedge inv(u2) \wedge G(u2, u3)) \vee (\exists\, u1, u2, o3, A1, A2 \bullet (G(u0, u1)$

$\vee\, C(u0, u1, A1)) \wedge (G(u1, u2) \vee C(u1, u2, A2)) \wedge (G(u2, u3) \vee C(u2, u3, A3))))$

$\Leftrightarrow$ *%u3 = u2 = u1 ⊆ u0; unifying/instantiating u1, u2 with u0.%*

$inv(u0) \wedge G(u0, u0) \wedge inv(u0) \wedge G(u0, u0) \wedge inv(u0) \wedge G(u0, u3)$

$\wedge\ inv(u3) \wedge trm(S3(u3, i3)) \wedge G(u0, u0) \wedge inv(u0) \wedge inv(u0) \wedge G(u0, u3)$

$\wedge\ W(i1, u1, i1, A1) \wedge W(i1, i2, u1, u2, A2) \wedge W(i2, i3, u2, u3, A3) \Rightarrow trm(S0(u0))$

$\quad \wedge\ [S3(u3, i3, o3)]\neg[S0(u0)]\neg((G(u0, u0) \wedge inv(u0) \wedge G(u0, u0)$

$\quad \wedge\ inv(u0) \wedge G(u0, u3)) \vee (G(u0, u0) \vee C(u0, u0, u0) \vee G(u0, u0)$

$\quad \vee\ C(u0, u0, u0) \wedge G(u0, u3) \vee C(u0, u3, o3, A3)))$

$\Leftrightarrow$ *%propositional simplification: G(u0,u0)=true; p ∧ p = p; p ⇒ p = true.%*

$inv(u0) \wedge G(u0, u3) \wedge inv(u3) \wedge trm(S3(u3, i3)) \wedge W(i1, u1, i1, A1) \wedge W(i1, i2, u1, u2, A2)$

$\wedge\ W(i2, i3, u2, u3, A3) \Rightarrow trm(S0(u0)) \wedge [S3(u3, i3, o3)]\neg[S0(u0)]\neg(G(u0, u3) \vee C(u0, u3, o3, A3))$

$\Leftrightarrow$ *%Definition of retrenchment.%*

$DMach \precsim OAMash$ □

In retrenchment (in comparison to refinement), the retrieves relation and the invariant are separated. A retrenchment specification should satisfy the retrieves relation *or* the concedes relation.

Theorem 5.3.2. *(Maximally Abstract Retrenchment)*

The proof of the operation retrenchment proof obligation requires satisfaction of the retrieves relation or the concedes relation. When the retrieves relation is satisfied, we have a subrefinement. When the concedes relation is satisfied, we have a concession.

Proof:

$(inv(S)(u) \wedge G(u, v) \wedge inv(T)(v) \wedge \underline{trm(T)(v,j)} \wedge W(i, j, u, v, A))$

$\quad \Rightarrow (\underline{trms(S)(u, i)} \wedge [T(v, j, p)]\neg[S(u, i, o)]\neg(G(u, v) \vee C(u, v, o, p, A)))$

$\Leftarrow$ *%[X](A ∨ B) ⇐ [X]A ∨ [X]B ∴RHS is a sufficient but not necessary condition%*

$(inv(S)(u) \wedge G(u, v) \wedge inv(T)(v) \wedge \underline{trm(T)(v,j)} \wedge W(i, j, u, v, A))$

$\Rightarrow (\underline{trms(S)(u,i)} \wedge [T(v,j,p)]\neg[S(u,i,o)]\neg G(u,v))$ %subrefinement

$\vee$

$(inv(S)(u) \wedge G(u,v) \wedge inv(T)(v) \wedge \underline{trm(T)(v,j)} \wedge W(i,j,u,v,A))$

$\Rightarrow (\underline{trms(S)(u,i)} \wedge [T(v,j,p)]\neg[S(u,i,o)]\neg C(u,v,o,p,A))$ % concession. □

In some cases, a concession may contain a *subrefinement* [Ban98]. For example, in the vanilla Banach-Poppleton retrenchment shown in Figure B.1, the *CONCEDES* clause has the same semantics as the *IF-THEN-ELSE* construct that models the operation in MSub, and similarly in DRSub and OASub: *IF a THEN b ELSE c* $\Leftrightarrow (a \Longrightarrow b)[](\neg a \Longrightarrow c) \Leftrightarrow (a \wedge b) \vee (\neg a \wedge c) \Leftrightarrow (a \Rightarrow b) \wedge (\neg a \Rightarrow c)$. Thus satisfaction of the concession is actually a *subrefinement* in this case.

5.4 Specification and proof of retrenchment in PVS

Mathematical functions provide a convenient way of modeling programming languages. The BMethod Generalized Substitution Language (BGSL) is based on the imperative paradigm. The specification language of PVS is based on the functional paradigm. The example in Figure B.1 is in terms of retrenching the idealized specification which involves the infinite set of all real numbers to a subset of the reals $\mathbb{FR}$. The Reals are formalised as the standard Abelian group in the PVS system [OS03a]. Both PVS and the B-Method use the substitution model of evaluation [SORSC98a].

5.4.1 Specification method for B machines in PVS

The style of PVS specification used is a shallow embedding of the B-Method in PVS, where the B-Method Generalized Substitution Language is translated into the PVS functional

Classical Higher-order specification language. A B-*MACHINE* corresponds to a PVS `THEORY`, and the *REFINES* clause is effected by the PVS `IMPORTING` statement. The state (u, v), input (i, j), output (o, p), and logical (A) variables are represented as the record structures U, V, I, O, A respectively. The machine operations X, Y, S, T are specified constructively, e.g. initialization is record construction; an assignment operation is a record update in PVS; and a continuation-style semantics [Ste98] can be effected by the PVS `LET` construct [6]. The machine assertions *RETRIEVES* $(G(u, v))$, *INVARIANT* $(I(u), J(v))$, *WITHIN* $(W(...))$, *CONCEDES* $(C(...))$ are represented by boolean definitions, and the retrenchment proof obligations in Figure B.1 are represented by PVS `THEOREMS`.

5.4.2 High Integrity translation from B to PVS

The retrenchment proof obligations listed in Figure 5.2 have to be converted from the B-Method Generalized Substitution Language into PVS classical higher-order logic functional specification language. The following theorem derives a predicate logic formula corresponding to the generalized substitution $[T(v, j, p)\neg[S(u, i, o)]\neg R(v, p, u, o)$:

Theorem 5.4.1. *(B GSL in Higher-Order Logic) [BP99b, Pop01]:*

$[T(v,j,p)]\neg[S(u,i,o)]\neg R(v,p,u,o)$

$\Leftrightarrow \mathit{trm}(T)(v,j) \wedge (\mathit{trm}(S)(u,i) \Rightarrow (\forall \tilde{v},\tilde{p} \bullet \mathit{prd}(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow$

$\quad(\exists \tilde{u},\tilde{o} \bullet \mathit{prd}(S)(u,i,\tilde{u},\tilde{o}) \wedge R(\tilde{v},\tilde{p},\tilde{u},\tilde{o}))))$

Proof:

$[T(v,j,p)]\neg[S(u,i,o)]\neg R(v,p,u,o)$

$\Leftrightarrow \%[S]R = \mathit{trm}(S)(u,i) \wedge (\forall \tilde{u},\tilde{o} \bullet \mathit{stp}(S)(u,i,\tilde{u},\tilde{o}) \Rightarrow [u,o := \tilde{u},\tilde{o}]R)\ \mathit{if}\ \tilde{u},\tilde{o}\backslash R\%$

[6]As in conventional programming practice, first the variables are initialized, and then the operations act on this initial state to give some after-state. The PVS statement `(LET x:int=2, y:int=x*x IN x+y)` is equivalent to the functional Lambda Calculus expression `(LAMBDA (x:int):(LAMBDA (y:int):x+y)(x*x))(2)` [SORSC98a].

$trm(T)(v,j) \land (\forall \tilde{v},\tilde{p} \bullet stp(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow [v,p := \tilde{v},\tilde{p}]$

$\quad \neg(trm(S)(u,i) \land (\forall \tilde{u},\tilde{o} \bullet stp(S)(u,i,\tilde{u},\tilde{o}) \Rightarrow [u,o := \tilde{u},\tilde{o}]\neg R(v,p,u,o))))$

$\Leftrightarrow$ *%$[u,o := \tilde{u},\tilde{o}]R(v,p,u,o) = R(\tilde{v},p,\tilde{u},o)$ twice%*

$trm(T)(v,j) \land (\forall \tilde{v},\tilde{p} \bullet stp(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow \neg(trm(S)(u,i) \land$

$\quad (\forall \tilde{u},\tilde{o} \bullet stp(S)(u,i,\tilde{u},\tilde{o}) \Rightarrow \neg R(\tilde{v},\tilde{p},\tilde{u},\tilde{o}))))$

$\Leftrightarrow$ *%$A \Rightarrow B \equiv \neg A \lor B$; $\neg\forall x \bullet Q \equiv \exists x \bullet \neg Q$%*

$trm(T)(v,j) \land (\forall \tilde{v},\tilde{p} \bullet stp(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow (\neg trm(S)(u,i) \lor$

$\quad (\exists \tilde{u},\tilde{o} \bullet \neg(\neg stp(S)(u,i,\tilde{u},\tilde{o}) \lor \neg R(\tilde{v},\tilde{p},\tilde{u},\tilde{o})))))$

$\Leftrightarrow$ *%$\neg(\neg A \lor \neg B) = A \land B$; $\neg A \lor B = A \Rightarrow B$%*

$trm(T)(v,j) \land (\forall \tilde{v},\tilde{p} \bullet stp(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow (trm(S)(u,i) \Rightarrow$

$\quad (\exists \tilde{u},\tilde{o} \bullet stp(S)(u,i,\tilde{u},\tilde{o}) \land R(\tilde{v},\tilde{p},\tilde{u},\tilde{o}))))$

$\Leftrightarrow$ *%$trm(S)(u,i)$ is unbound under $\forall \tilde{v},\tilde{p}$%*

$trm(T)(v,j) \land (trm(S)(u,i) \Rightarrow (\forall \tilde{v},\tilde{p} \bullet stp(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow$

$\quad (\exists \tilde{u},\tilde{o} \bullet stp(S)(u,i,\tilde{u},\tilde{o}) \land R(\tilde{v},\tilde{p},\tilde{u},\tilde{o}))))$

$\Leftrightarrow$ *% $stp(S)(u,i,u',o) = trm(S)(u,i) \land prd(S)(u,i,u',o)$;*

$\quad stp(T)(v,j,v',p) = trm(T)(v,j) \land prd(T)(v,j,v',p)$%:

$trm(T)(v,j) \land (trm(S)(u,i) \Rightarrow (\forall \tilde{v},\tilde{p} \bullet prd(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow$

$\quad (\exists \tilde{u},\tilde{o} \bullet prd(S)(u,i,\tilde{u},\tilde{o}) \land R(\tilde{v},\tilde{p},\tilde{u},\tilde{o}))))$

$\Leftrightarrow$ *%Converting to prenex normal form (PNF)—if x does not occur free in A then*

$\quad A \Rightarrow \forall x : B \Leftrightarrow \forall x : A \Rightarrow B$ and $A \Rightarrow \exists x : B \Leftrightarrow \exists x : A \Rightarrow B$%:

$\forall \tilde{v},\tilde{p} \bullet \exists \tilde{u},\tilde{o} \bullet trm(T)(v,j) \land (trm(S)(u,i) \Rightarrow (prd(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow$

$\quad (prd(S)(u,i,\tilde{u},\tilde{o}) \land R(\tilde{v},\tilde{p},\tilde{u},\tilde{o}))))$

$\Leftrightarrow$ *%Generalising the formula by universally quantifying the unquantified variables u,i,v,j %*

$\forall u,i,v,j,\tilde{v},\tilde{p} \bullet \exists \tilde{u},\tilde{o} \bullet trm(T)(v,j) \land (trm(S)(u,i) \Rightarrow (prd(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow$

$\quad (prd(S)(u,i,\tilde{u},\tilde{o}) \land R(\tilde{v},\tilde{p},\tilde{u},\tilde{o})))) \ \square$

B	PVS	$stp(S) \Leftrightarrow trm(S) \wedge prd(S)$
$x := E$	initialization: `(# x := E #)` assignment in Op: `r WITH [x := E]`	$x' = E$
$skip$	`r WITH [x := x]`	$x' = x$
$P \mid S$	`OpName(r:R | P) = S`	$(P \wedge trm(S)) \wedge (P \Rightarrow prd_x(S))$
$S \mathbin{[]} T$	`IF true THEN S ELSE T ENDIF`	$(trm(S) \wedge trm(T)) \wedge (prd_x(S) \vee prd_x(T))$
$P \Longrightarrow S$	`COND P -> S ENDCOND`	$(P \Rightarrow trm(S)) \wedge (P \wedge prd_x(S))$
$@\, z \cdot S$	`FORALL (z:TYPE):trm(S) AND EXISTS (z:TYPE): prd(S)`	$(\forall\, z \cdot trm(S)) \wedge (\exists\, z \cdot prd_x(S))$ *if* $z \backslash x'$
$P \mid @\, x' \cdot (Q \Longrightarrow x := x')$	`P AND (P IMPLIES Q)`	$P \wedge (P \Rightarrow Q)$
$x :\in E$	`member(y,E) AND member(xp,E)`	$(y \in E) \wedge (x' \in E)$
$x : P$	`x:P`	$[x_0, x := x, x']P$

Table 5.1: B to PVS

The formulation of the B-Method General Substitution Language in Higher-Order Logic above conforms to the Proofs as Programs (or Program Synthesis) proof obligation $\forall(x : D) : \exists(z : R) : (I(x) \Rightarrow O(x, z))$ [Kre98] with the difference that the proof obligation is in terms of one machine (the idealistic machine), whereas the Theorem 5.4.1 is in terms of two machines (both the idealistic and realistic machines).

The operations S, T can be defined constructively or declaratively in PVS by using the correspondences in Table 5.1 and the PVS LET statement. Constructive definitions can be valuable in proof development, particularly for computing values for instantiating existential variables in the retrenchment proof obligations. This provides the mathematically rigorous mechanism of Lambda Calculus to check the correctness of the retrenchment of an operationally defined abstract specification by an operationally defined concrete specification. In this way logical errors in the operational definition of the operations can be caught by verification thus reducing the likelihood of such errors, which have been found to account for the greatest percentage of faults in the software development process, and consequently the implementations [YBH97].

5.4.2.1 PVS specification of the architectural retrenchment in Figure B.1

The specification of a particular Banach-Poppleton vanilla retrenchment of a divine (idealistic) machine by a realistic machine consists of the following PVS `THEORY`s: (1) the divine machine *Dmach* [Figure B.2]; (2) the input architecture machine *IAmach* which imports *Dmach* [Figure B.2] [7]; (3) the data-representation machine *DRmach* which imports *IAmach* [Figure B.3]; (4) the output-architecture machine *OAmach* which imports *DRmach* [Figure B.4]; (5) input-architecture proof obligations *IAPOs* which imports *IAmach* [Figure B.5]; (6) the data-representation proof obligations *DRPOs* which imports *DRmach* [Figure B.6]; and (7) the output-architecture proof obligations *OAPOs* which import *OAmach* [Figure B.7].

The identifiers in *Dmach* are indexed by the numerical 0, e.g. `u0:U0,S0:[U0 -> U0]`; those in *IAmach* by 1, e.g. `u1:U1,i1:I1, S1:[U1->[I1->U1]]`; those in *DRmach* by 2, and those in *OAmach* by 3 [8]. The operations are specified constructively and declaratively using the correspondences in Table 5.1.

Note that the *WITHIN* clause is augmented to include the initialization of the machine, and the *WITHIN* clauses of the previous retrenchments, e.g. the `WITHIN` clause for `DRSub` is given as:

```
W(u0:U0, u1,A1:U1, u2,A2:U2, i1:I1, i2:I2): bool = W(u0,u1,A1,i1)
  & a1(u1)>=b1(i1) & (b2(i2)<=10 => b1(i1)=b2(i2)) & a2(u2)=a2(A2)
```

[7]Note in PVS these are specified as two separate files. They are conjoined here for brevity.

[8]In [Ban98], for a series of retrenchments, the machine variables and operation names are lexically ordered from machine to machine , i.e. for `DSub` we have variables `u:U, i:I, o:O` and operation `S`; correspondingly for `IASub` we have variables `v:V, j:J, p:P` and operation `T`; for `DRSub` we would have variables `w:W, k:K, q:Q` and operation `U`; and variables `x:X, l:L, r:R` and operation `V` for `OASub`.

Where `W(u0,u1,A1,i1)` is the *WITHIN* clause for `IASub`. This is to ensure that the previous conditions 'within' which the previous retrenchments were valid, carry through to the present retrenchment. In particular, in the idealistic machine `DSub`, all variables are initialized since all variables are state variables in this case. On the other hand, `IASub` does not initialize input variables, e.g. `b1(i1)` is not initialized, thus statements like `a1(u1)>=b1(i1)` would not be decidable. This is because in the input-architecture, data-representation and output-architecture retrenchments, the input variables are not initialized with actual values since the `initDef` operation only works on state variables. In the input-architecture retrenchment, the *WITHIN* clause statement `b1(u1)=b0(u0)` provides a rewrite rule where `b0(u0)` has an actual value from the initialization in DSub, i.e. the initializations of the more realistic machine are in terms of the initialization of the idealistic machine, which is included in the *WITHIN* clause. Thus the *WITHIN* clause can act as a repository where certain proof obligations, such as the machine initializations to satisfy the TCCs for a machine specification, can be added so that the retrenchment proof obligations can be made provable. As such the *WITHIN* clause can be interpreted as a "contract" under which a certain retrenchment is valid.

The specification of the input architectural retrenchment proof obligations in classical higher-order logic is given in Figure B.5. The invariant preservation proof obligations (excluding the invariant `AInvPO` for the divine (idealistic) machine) are also augmented with the *WITHIN* clause of the `THEORY` being retrenched as a means of chaining the initialization (of all variables) in the idealistic machine to later retrenchments. This is similar to the idea of maximally abstract retrenchments [Ban98]. Note that `RetInitPO`, `SubrefinementPO` and `ConcessionPO` are specified in the original Banach-Poppleton formulation, i.e. the proof obligations are not in prenex normal form. The proof obligations can easily be transformed into prenex normal form by performing the last two steps in Theorem 5.4.1.

Each PVS machine `THEORY` has slots for the relevant B-machine attributes in which definitions can be entered to specify a particular machine, e.g. the slots `U0, initDef, init, S0Def, inv, trm, prd` for the divine machine *Dmach*. The retrenchment proof obligations are general proof obligations for all retrenchment specifications, and thus are static for all machine definitions. Thus the seven PVS `THEORY`s above can be used as specification templates on which the user need only change the variable types, attribute definitions and the `IMPORTING` clause to specify a particular Banach-Poppleton Vanilla Retrenchment to be verified.

5.4.3 Proof of the retrenchment specifications in PVS

The PVS system comes with a parser to ensure that no syntax errors are present in the specification, a typechecker to check that the types of variables used in the specification are type-consistent, and a prover to prove the correctness of the specification under the Gentzen Sequent Calculus LK. The parsing and typechecking of the specifications can be viewed as the Exploration phase in the Formal Methods Lifecycle (Procedure 2.3.1), for the verification of the retrenchment proof obligations specifications.

5.4.3.1 Proof procedure for architectural retrenchment POs

The proofs are for the particular initialization made in the idealistic machine [9]. This is consistent with conventional programming practice where variables are initialized before

[9]In particular, we only need to modify the initialization of the idealistic machine to define a new proof case. This is convenient for the retrenchment of reals to floats for which there are 5 distinct cases of initializations: (1) underflow, (2) overflow, (3) exact representation, (4) approximation, and (5) not-a-number.

being passed to operations. In particular, the generalisation in Theorem 5.4.1 to have all free variables universally quantified ensures that the predicate is valid for all initial states:

$$\forall\, u, i, v, j, \tilde{v}, \tilde{p} \bullet \exists\, \tilde{u}, \tilde{o} \bullet trm(T)(v,j) \wedge (trm(S)(u,i) \Rightarrow (prd(T)(v,j,\tilde{v},\tilde{p}) \Rightarrow (prd(S)(u,i,\tilde{u},\tilde{o}) \wedge R(\tilde{v},\tilde{p},\tilde{u},\tilde{o}))))$$

For a specification where u, i, v, j are from a well-ordered domain, e.g. $u, i, v, j : \mathbb{N}$, the principle of mathematical induction can be used to reason about the predicate above for all such initial states. However in this work u, i, v, j are reals which are not well-ordered and hence an induction scheme for real numbers does not exist. Therefore Theorem 3.6.1 is used instead in this work.

The initialization proof obligations ensure that the initializations $X(u)$, $Y(u)$ etc, satisfy their machine invariants I, J respectively, and the invariant-preservation proof obligations ensure that the operation(s) S, T etc, satisfy their machine invariants. Thus these proof obligations ensure the correctness of an individual machine. The retrenchment initialization, subrefinement and concession proof obligations ensure the correctness of the 'realization' of the idealistic machine by the realistic machine. The retrenchment initialization proof obligation is concerned with the retrenchment of the initializations of the two machines, and the other retrenchment proof obligations are concerned with the retrenchment of the other operations defined in the machine [10]. Thus to ensure a correct retrenchment, one first has to prove that each machine is correct, the retrenchment initialization is correct, and then finally the subrefinement proof can be attempted; if the subrefinement is unprovable, then the concession must be provable.

The method of Tactic refinement (Procedure 3.3.1) where each tactic is derived according to Theorem 3.6.3 is used in the proof of each architecture proof obligations `THEORY`s

[10]The machines considered in this work contain only one defined operation but it is possible for a machine to have more than one defined operation, each of which can be proved by the method outlined herein.

IAPOs, *DRPOs*, *OAPOs*. The initialization proof obligations have the same structure as the invariant proof obligations; the retrenchment initialization POs constitute a subproof of the subrefinement and concession proof obligations, both of which have the same structure (see Figures B.5, B.6 and B.7 in Appendix B). Thus a robust tactic from the proof of the initialization PO may be reusable to prove an invariant proof obligation, and similarly a robust tactic from the proof of the subrefinement PO may be reusable to prove the concession proof obligation. It is likely that the tactics from proofs on one architectural retrenchment may fail on the proof obligations in the next architecture due to changes in operation signature and datatypes, e.g. since the input-architecture (and output-architecture) retrenchment involves an operation signature change, the tactics from proofs of the proof obligations in IAPOs may not be reusable on the proof obligations in the data-representation retrenchment DRPOs which involves a change in datatype. However it is expected that by composing the tactics from IAPOs and those from DRPOs, the tactics may be reusable in the proof of the OAPOs which involves both an operation signature change and a datatype change (see Section 5.3.1).

5.5 Tactics from proofs of retrenchment in PVS

The three PVS specifications in Appendix B are parsed successfully which means they do not contain any syntax errors.

5.5.1 Automatic generation of proof obligations as TCCs

On typechecking using the PVS typechecker, the machine specifications `DSub`, `IASub` and all the architecture retrenchment proof obligations specifications `IAPOs`, `DRPOs`, `OAPOs` do not generate any TCCs therefore these specifications are type-correct. However the machine specifications `DRSub`, `OASub` generate TCCs for the machine operations `init`, `S2Def`, `S3Def` and messages concerning the types of the state-variables as shown in Figure B.3 and Figure B.4 respectively. The TCCs for the machine theories `DRSub`, `OASub`, concern the satisfaction of the invariant (`a(u) < MaxReal`) by the after-state (`u`) of the operations. From the definition of the initialization operation, the TCC for the initialization operation `initDef_TCC1` is proved complete by the PVS tactic for TCCs `tcp`. However the TCCs for S2Def and S3Def could not be finished by tactic `tcp` since these operations take the input variable `b` which is not initialized in the machine. Thus it is impossible to prove that the subtraction of an uninitialized variable `b` from an initialized variable gives a result satisfying the invariant. These unfinished TCCs are unraveled in the proof of the architecture retrenchment proof obligations specifications where the *WITHIN* clause chains this information throughout the retrenchment specifications.

5.5.2 Tactics for the retrenchment POs not in Normal Form

Chapter 4 demonstrates that the definition of `grind` conforms to the normal form of proofs given by Theorem 3.6.3 albeit the instantiator argument defaults to the heuristic PVS tactic `(inst?)` which sometimes instantiates with incorrect arguments leading to an unprovable proofstate, or fails to find a suitable instantiation at all. The `RobustGrind` tactic (Definition 4.3.2) can be used to apply `(grind)` as the first step in an interactive proof development, and if `(grind)` fails due to an incorrect automatic instantiation, the original goal is simplified into manageable subgoals for the user to perform the creative proofsteps. The initialization and invariant-preservation proof obligations, can be proved by `(grind)` since these two obligations just require skolemisation of variables, which PVS can perform automatically.

The proofs of RetInitPO, which involves initialization operations init(u0), init(u1), init(u2), init(u3); SubRefinementPO and ConcessionPO which both involve the machine operations S0, S1, S2, S3, both require instantiation of the after-state variables $\tilde{u}, \tilde{o}$ in Theorem 5.4.1. The PVS (grind) tactic is able to find the correct instantiation terms u0!1, u1!1, u2!1 for u0p, u1p, u2p respectively for RetInitPO from the proofstate. However (grind) is not able to find the correct instantiation terms from the proofstate for SubRefinementPO and ConcessionPO—it gives the erroneous instantiation for RetInitPO above, which results in a subgoal that cannot be proved as shown by the failed proof attempts in Appendix B.1. Thus the pattern-matching technique used by the PVS tactic (inst?) is not powerful enough to find the correct instantiation terms to enable the proofs for SubRefinementPO and ConcessionPO to complete.

The constructive definitions of the machine operations S0, S1, S3 are used to actually compute from the skolem variables u0!1, u1!1, u2!1, the values that can be used as instantiation terms in the PVS manual instantiation tactic inst, e.g. manual instantiation by (inst + "S0Def(u0!1)") completes the proof for SubRefinementPO in the input-architecture retrenchment IAPOs (see Appendix B.1). This alleviates the human user from having to construct the instantiation expressions explicitly on the PVS prover command line, which is more laborious especially for nontrivial instantiations involving a significant amount of variables. In addition, the correctness of the definition of the operation itself is checked in that the values the operation computes are actually the values which satisfy the before-state and after-state predicates and the whole retrenchment as a whole [11]. In this way, PVS can be used in the invent-and-verify (posit-and-proof)

[11]Note that the operations are defined constructively, and the before-state and after-state predicates are defined declaratively.

method of specification development, whereby constructive definitions are specified and then proved by demonstrating that they can compute suitable values which satisfy proof obligations such as the retrenchment proof obligations.

The interactive proof in Appendix B.1 demonstrates that the proof of retrenchment proof obligations not in normal form consists of three phases—rewriting, instantiation and completion.

5.5.2.1 Rewriting to *instantiation* phase

Definition 4.3.2, when evoked as `(RobustGrind :defs nil)` in order to avoid overwhelming the user with too much detail, simplifies the `SubRefinementPO` proof obligation into the following two manageable subgoals:

```
RetrenchmentPO1:
inv(S)(u), G(u,v), inv(T)(v), trm(T)(v,j), W(u,v,i,j,A), prd(T)(v,i,v',p)
|- EXISTS (u',o'): (trm(S)(u,i) & prd(S)(u,i,u',o')) & R(v',p',u',o')
RetrenchmentPO2:
inv(S)(u), G(u,v), inv(T)(v), trm(T)(v,j), W(u,v,i,j,A)
|- (trm(S)(u,i)
```

The presence of these two subgoals means that the tactic `(grind)` is unable to find a proof automatically, in particular the correct instantiation terms. `RetrenchmentPO2` requires to prove that if the before-state satisfies the invariants, retrieves relation, concrete termination condition, and WITHIN clause, then the abstract termination condition shall be established. `RetrenchmentPO1` requires to prove that if the antecedent of `RetrenchmentPO2` and the concrete before-after relation are valid, then the after-state of the abstract machine and retrieves relation the shall be established. Thus to avoid making the proof trivial, the antecedent should be true, and this is investigated by proving the initialisation, invariant preservation, and retrenchment initialisation proof obligations first. These proof obligations, as well as `RetrenchmentPO2`, can be proved by `RobustGrind` when the optional `defs` parameter is not used.

5.5.2.2 Instantiation

The proof obligations that require instantiation are `RetInitPO` and `SubRefinementPO` and `ConcessionPO`, where the instantiation is for the elimination of the existential quantifier. `RobustGrind` (Definition 4.3.2 of Chapter 4) is able to find the correct instantiation terms `u0!1, u1!1, u2!1` for `u0p, u1p, u2p` respectively for `RetInitPO` from the proofstate.

Manual instantiation for `SubRefinementPO` and `ConcessionPO` is invoked by the proof-step `(inst` *fnum terms*`)`, and the instantiation terms are the definitions of the initialization, and of the machine operation; with the skolem variables from the skolemisation of the initial state as follows:

`(inst + "S0Def(u0!1)")` for `IAPOs`

`(inst + "S1Def(u1!1)(i1!1)")` for `DRPOs`

`(inst + "S2Def(u2!1)(i2!1)")` for `OAPOs`

5.5.2.3 Completing the proof

After the instantiation step, all quantifiers will have been eliminated—provided none of the definitions in the specification are in terms of quantifiers. Thus all that is left to do is to expand some definitions in the proofstate, and perform propositional simplification using `(grind :if-match nil)`.

The ConcessionPO in the input-architecture retrenchment `IAPOs` does not prove since the Concedes relation is defined as *false*. This demonstrates that the antecedent is true, which is as expected. For example, for the IAPOs in Appendix B, after the rewriting and instantiation with `(inst + "S0Def(u0!1)")`, the proofstate above reduces to the unprovable proofstate:

```
ConcessionPO :
{-1}  (a1(u1!1) = 3)
{-2}  3 >= 1
{-3}  a0(u0!1) = 3
{-4}  b0(u0!1) = 1
{-5}  (b1(i1!1) = 1)
{-6}  a1(u1p!1) = 2
  |-------
```

This gives confidence that the SubrefinementPO, which is provable, is correct—since the antecedent of the SubrefinementPO is the same as the antecedent of the ConcessionPO, therefore the Subrefinement consequent `EXISTS (u1p: U1): prd(S1!1)(u1!1, i1!1, u1p) & G(u1p, u2p!1)` must be true for the proofstate to be provable.

5.5.3 Tactics for retrenchment POs in PNF

The retrenchment proof obligations `RetInitPO, SubrefinementPO, ConcessionPO` can be expressed in prenex normal form according Theorem 5.4.1, e.g. the retrenchment proof obligations for `IAPOs` are converted to prenex normal form as follows:

```
RetInitPO: THEOREM
 FORALL (u0:U0, u1:U1, S0:[U0->U0], S1:[U1->[I1->U1]]):
  FORALL (u1p:U1): EXISTS (u0p:U0): init(u0,u1p) => (init(u0,u0p) & G(u0p,u1p))
```

```
SubRefinementPO: THEOREM
 FORALL (u0:U0, u1,A1:U1, i1:I1, S0:[U0 -> U0], S1:[U1 -> [I1 -> U1]]):
  FORALL (u1p:U1): EXISTS (u0p:U0):  (init(u0) & init(u0,u1)) =>
   ((inv(S0)(u0) & G(u0,u1) & inv(S1)(u1) & trm(S1)(u1,i1) & W(u0,A1,u1,i1))
    => (trm(S0)(u0) & trm(S1)(u1,i1) & (trm(S0)(u0)
       => (prd(S1)(u1,i1,u1p)) => (prd(S0)(u0,u0p) & G(u0p,u1p)))))

ConcessionPO: THEOREM
 FORALL (u0:U0, u1,A1:U1, i1:I1, S0:[U0 -> U0], S1:[U1 -> [I1 -> U1]]):
  FORALL (u1p:U1): EXISTS (u0p:U0): (init(u0) & init(u0,u1)) =>
   ((inv(S0)(u0) & G(u0,u1) & inv(S1)(u1) & trm(S1)(u1,i1) & W(u0,A1,u1,i1))
     => (trm(S0)(u0) & trm(S1)(u1,i1) & (trm(S0)(u0)
        => (prd(S1)(u1,i1,u1p) => (prd(S0)(u0,u0p) & C(u0p,u1p,A1))))))
```

Theorem 3.6.1 can then be applied as the first creative step to skolemise the universal variables, and then use the skolem variables in the machine operations to generate the instantiation terms for the after-state. The LK mechanical rules are then applied on the quantifier-free formula using `(grind)` to finish the proof.

5.5.4 General retrenchment tactic

The proofs for the data-representation retrenchment and output-architecture retrenchment follow the same pattern of proof as that described for the input-architecture retrenchment above, and using the instantiation described in Section 5.5.2.2 above. The input-architecture was found to be a subrefinement, and the data-representation retrenchments and output-architecture retrenchment proved for both the subrefinement and concession

POs, since the Concession is also a subrefinement as alluded to in Section 5.3.2.

A composite tactic of the three phases (Sections 5.5.2.1, 5.5.2.2, 5.5.2.3) that can prove the retrenchment POs in and not in prenex normal form is:

```
(defstep RSubRetTac (fnum &rest terms)
  (then (RobustGrind$) (inst fnum terms) (grind))
 "INPUT = IAPOs or DRPOs or OAPOs
  OUTPUT = TRUE
  PRECONDITIONS = DSub \/ IASub \/ DRSub \/ OASub
  EFFECTS = QED"
 "proving retrenchment proof obligation ...")
```

Where the formal parameters `fnum terms` are replaced by the actual instantiation arguments corresponding to the particular proof obligation being proved as given in Section 5.5.2.2. This tactic was successfully used to automatically prove each of the proof obligations in all the architectural retrenchments for this example.

5.6 Summary

This chapter has demonstrated the shallow embedding of the B-Method in PVS in order to reason about the Retrenchment Method in PVS. Figure 5.2 summarizes the formulation of the Banach-Poppleton vanilla retrenchment as Architectural retrenchment, and Figure B.1 shows the B-Method Architectural retrenchment specifications for an example from [Pop01]. Theorem 5.3.1 demonstrates that Architectural Retrenchment is equivalent to the one-off Banach-Poppleton retrenchment specification of a machine. Theorem 5.3.2 demonstrates that the one-off Banach-Poppleton Operation Retrenchment Proof Obligation can be split into 2 separate retrenchments: (1) the satisfaction of the *RETRIEVES* relation $G(u, v)$ is considered a subrefinement; and (2) the satisfaction of the *CONCEDES* relation $C(u, v, o, p, A)$ is considered a concession.

Figures B.2 and B.3 show the PVS specifications of the B-Method Architectural retrenchment specifications in Figure B.1; and Figures B.5, B.6, B.5 show the PVS specification of the B-Method retrenchment proof obligations. These specifications contain slots for each of the machine attributes as defined in Figure 5.2 and Figure 5.1 respectively and thus can be used as specification templates—the user need only type in the relevant definitions corresponding the particular retrenchment. The PVS `IMPORTING` clause is used for the *RETRENCHES* clause, the state is a specified as a PVS `record`, and operations can be defined declaratively and constructively using Table 5.1. Theorem 5.4.1 demonstrates how the Generalized Substitution Language fragment $[T(v, j, p) \neg [S(u, i, o)] \neg R(v, p, u, o)$ can be converted into a Higher-Order Logic statement (in prenex normal form) to enable the proof of the B-Method Retrenchment proof obligations in PVS.

On typechecking, each of the PVS specifications `DSub, IASub, IAPOs, DRPOs, OAPOs` do not generate any Type Correctness Conditions (TCCs). However `DRSub, OASub` give TCCs which are thrown for the operation definitions and concern the value of the input variable which is not initialized by the operation and thus it is not possible to complete the proof. These unfinished TCCs are accounted for in the proof of the retrenchment proof obligations which involves the *WITHIN* clause wherein the "contract" on values for the variables used in the retrenchment is specified. Thus the *WITHIN* clause is a means of chaining values from one specification to another and acts as a repository wherein the TCC proof obligations generated for a machine can be added to enable a proof of retrenchment to complete. This can be seen as a formal way to apply logic induction.

For retrenchment proof obligations not in prenex normal form (as in the B.5, B.6, B.5) the tactic `RobustGrind` is used to rewrite the proof obligation to an instantiable form; the instantiation is then performed as in Section 5.5.2.2; and the proof is finished by applying the mechanical rules and decision procedures automatically by invoking `(grind)`. For the proof obligations expressed in prenex normal form using Theorem 5.4.1, Theorem 3.6.1 is the first creative proof step, which is then followed by the automatic application of mechanical rules and decision procedures via `(grind)`. The resultant proofs both conform to the normal form given in Theorem 3.6.3. In this work, the other creative rules (cut and induction) are not used.

For the example of retrenching reals by a subset of the reals `FinReal: TYPE = {x:real | x <= 10}`, the robust tactic formulated for the Input Architecture Retrenchment (`RSubRetTac` *fnum terms*) which is defined as `(then (RobustGrind$) (inst fnum terms) (grind))`, was found to work for all the data-representation and output-architecture retrenchments where the actual arguments *fnum terms* are the appropriate instantiation for the retrenchment architecture being proved. This is because the change in datatype is from the set of Reals to a proper subset of reals `FinReal`, which is therefore a refinement. The next chapter deals with retrenchment of reals by floats, where the float data-type and operations are specified according to the IEEE-854 standard for floating-point computations.

Chapter 6

Retrenching reals by floats

"When performing computations in floating-point arithmetic the computed values of intermediate variables as well as the computation of the final values of a calculation are somewhat different from those values computed with the same algorithm in the field of real numbers." [KB01].

This chapter examines the modest change in specification from real computation to floating-point computation.

6.1 Introduction

A recent trend in software engineering is the application of formal methods on the continuous domain. We look at the formal development of specifications involving real numerical computation. Retrenchment, a liberalization of the stepwise refinement method has been proposed to deal with the problems peculiar in this domain.

Reals are used in the 'idealistic world' of hand computation, whereas computers use floats in the 'realistic world' of electronic digital computation, i.e. in computer-aided formal software development, the users requirements are captured using idealistic datatypes, e.g. infinite reals, and the implementation is executed on computer using the finite resources of the computer, e.g. the machine numbers (floats). The main objective of this chapter is to investigate whether the tactics formulated for the retrenchment of infinite reals to finite reals, can successfully prove conjectures if there is a change of datatype and operation definition to floats and floating-point operations respectively.

Following the formal methods application lifecycle articulated in [NASa], we describe the problem in Section 6.2, and some approaches to reliable numerical computation in Section 6.2.2. Section 6.2.1 discusses the characterization of the problem domain in more detail. The next phase is modeling and this is the subject of Section 6.3. The specification, Section 6.3.1, is done in *PVS*, and Section 6.4.1 discusses the proof method employed. Finally, the extraction of tactics from the proofs using out theory of abstraction is discussed in Section 6.5. Section 6.6 discusses the results obtained and the viability of our approach.

6.2 Reliable numerical computation

When performing computations in floating-point arithmetic (i.e. finite precision) the computed values of intermediate variables as well as the computation of the final values of a calculation are somewhat different from those values computed with the same algorithm in the field of real numbers. Additionally, imprecise data ... contaminate the computed results with errors [KB01]. This is known as the Real-to-Floats problem.

This problem highlights the problems between idealized specifications—those involving real numbers—and realizable specifications—those involving floating-point numbers. Catastrophes caused by software due to incorrect numerical computing include the explosion of Ariane 501 [ESA96], and the Intel Pentium floating-point division instruction flaw [Pra95].

6.2.1 Characterization of the problem domain

A floating-point *representation* format is "a data structure specifying the fields that comprise a floating-point numeral, the layout of these fields, and their arithmetic interpretation." [Sun96].

Definition 6.2.1. (`Floating-point number`) [Gol91]: In general, a floating-point number will be represented as $s\ d.dd...d \times b^e$ which represents the number $\pm(d_0 + d_1 \times b^{-1} + \ldots + d_{p-1} \times b^{-(p-1)}) \times b^e$ where $0 \leq d_i < b$. The float datatype can be stored as a tuple $\mathbb{F} = (b, f, \alpha, e_{max}, e_{min})$

Where b is the base or radix; f is the fraction (mantissa or significand) $d.dd...d$; the exponent adjustment α is approximately equal to $(3 \times (e_{max} - e_{min}))/4$ and should also be exactly divisible by 12 [IEE87]; e_{max} and e_{min} are the maximal and minimal exponent allowed by the precision; p is the precision (i.e. number of digits in the fraction); and s is the sign (+ or -) of the float. For example, the IEEE *single representation format* consists of a 23-bit fraction, an 8-bit biased exponent, a 1-bit sign, and the base is 2 on a digital computer.

A floating-point *storage* format specifies how a floating-point representation format is stored in the memory of a computer. The IEEE standard defines representation formats, but the choice of storage formats is left to the implementers, e.g. in the double representation format, in the SPARC architecture, the higher address 32-bit word contains the least significant bits of the fraction, whereas in the Intel and PowerPC architectures, the lower address 32-bit word contains the least significant 32 bits of the fraction [Sun96]. Assembly language software sometimes relies on using storage formats but higher level languages deal with linguistic notions of floating-point data types, e.g. *single, double, double extended* representation or precision [Sun96].

The floating-point representation format is at the sufficient level of abstraction for our purposes as it does not go into details of how the representation is actually stored in the computer memory space. *Normalization* ensures that for the most significand bit of the mantissa/significand (msb), $(0 < msb < b)$, and this ensures the smallest possible exponent (e) is used, e.g. normalization ensures that $msb = 1$ on a digital/binary computer, and the msb bit can be used as an extra fraction bit thereby increasing the precision of the format. However normalization only occurs after the real has already been converted to a float, and so can be ignored without loss of generality.

6.2.1.1 Assumptions about floating-point computation

The following assumptions hold for computer (i.e. floating-point) arithmetic, where the function *value*(f) calculates the corresponding real value of a float representation f.

Assumption 6.2.1. *($(1+\varepsilon)$-property) [KB01]: For an operation $\circ \in \{+, -, \times, /\}$ on reals, and its machine analog $\circledcirc \in \{\oplus, \ominus, \otimes, \oslash\}$ on floats, the relative error between the operation is given by $\mid \frac{(value(f_1) \circ value(f_2)) - (f_1 \circledcirc f_2)}{(value(f_1) \circ value(f_2))} \mid \leq \overline{\varepsilon}$ forall floating-point numbers f_1, f_2 where $\mid (value(f_1) \circ value(f_2)) \mid \in [MinReal, MaxReal]$; and $\overline{\varepsilon} = \varepsilon$ for round to nearest; $\overline{\varepsilon} = 2\varepsilon$ for directed rounded operations; the machine epsilon $\varepsilon = (b/2)(b^{-p})$.*

Thus the relative error between a real computation and its floating-point counterpart can be expressed in terms of the machine epsilon ε which can be determined from the precision used for representing the floats. The $(1+\varepsilon)$-property corresponds to *Theorem* 2 of [Gol91] provided the operation (subtraction or addition) is done with $p + 1$ digits, i.e. with one guard digit which means that the truncation of the operands is done to $p+1$ digits and the result is rounded to p digits. Note that the process of normalization ensures the availability of one guard digit. Theorem 1 [Gol91] says that the relative error can be as large as $b - 1$.

For a normalized float, $MinReal = 1.00...00 \times b^{e_{min}} = b^{e_{min}}$ and $MaxReal = 1.11...11 \times b^{e_{min}}$ where there are p digits in the fraction.

Assumption 6.2.2. *(Underflow) In the underflow range, $U = (-MinReal, MinReal)$, i.e. for floating-point operands f_1, f_2 with $\mid value(f_1) \circ value(f_2) \mid < MinReal$, it holds that*

$\mid (value(f_1) \circ value(f_2)) - value(f_1 \circledcirc f_2) \leq MinReal$

The sign of the exact result $(value(f_1) \circ value(f_2))$ and the sign of the result of the floating-point operation $value(f_1 \circledcirc f_2)$ are always the same.

Assumption 6.2.3. *(Exact result) Let f_1, f_2 be floats. Whenever the result of a real operation with floating-point operands is already representable as a floating-point number, this number must be the result of the corresponding floating-point operation:*

$f_1, f_2 \in \mathbb{F} \Rightarrow value(f_1 \circledcirc f_2) = (value(f_1) \circ value(f_2))$.

Sun's numerical computation guide [Sun96] identifies five exceptions that arise in floating-point computation: overflow, underflow, inexact representation, invalid operation and division by zero. Invalid operation, division by zero, and overflow are *common* exceptions which can seldom be ignored when they occur, and can be trapped by the ieee_handler(3m) software. Underflow and inexact are seen more often, with most operations incurring the inexact exception, and they can usually though not always, be safely ignored.

6.2.2 Specification of programs involving real computation

To formally specify and develop a program, f, which given a quantity x, calculates the value of $f(x)$, the following approaches can be used [BK96]:

1. Specify f using computational real numbers and refine the specification to use computational reals [Bri79, San68]. Computational real [Esc00] numbers are those real

numbers such that an algorithm exists that computes rational approximations of arbitrary precision to them. This makes reasoning about the specification easy but at the expense of a final program that uses enormous amounts of computing resources.

2. Specify f using floating-point numbers and refine the specification to use floating-point numbers [Bro81, Wic89]. There is loss of abstraction and a detailed error analysis of final code is required in order to ensure that results are returned within acceptable error bounds. Also, the use of many of the convenient properties of the reals is lost.

3. Specify f using intervals of reals in the original specification and refine them to intervals bounded by floating-point numbers in the implementation [AA98, BK96, Kre95] This maintains the convenience and abstractness of the reals while retaining the speed of floats. Natural translations of numerical algorithms over the reals to floating point numbers are usually incorrect whereas intervals retain correctness by providing a stringent way of representing and reasoning about associated errors already present in measurements taken as input data. This brings about a slightly different notion of data refinement.

4. Specify f using real numbers and approximate the reals by floating-point numbers [Pop01, PB02, Har96]. Programs produced have good performance but at the expense of making specification and reasoning difficult—natural specifications are likely to be infeasible, and reasoning will need to constantly refer to the detailed properties of floating-point numbers in order to carry out the necessary error analysis.

In the context of stepwise software development, the last two approaches are natural—in particular, the retrenchment method [Pop01] can be used to formalize the problem of approximating reals with floats. The detailed properties of floating-point numbers can be made available through a formal specification of a floating point standard, e.g. the IEEE-854 floating-point standard [IEE87] defines the datatype *float* and the basic floating-point operations *Add, Sub, Mul, Div, Rem, Sqrt.* The float datatype depicts how the computer actually represents numbers, and the floating-point operations depict how the computer actually performs operations on numbers.

The IEEE-854 standard has been specified and proved in PVS [Min95, CM95]. Thus by incorporating the IEEE-854 standard in PVS retrenchment specifications, proofs of correctness for the fourth style of specification above can be performed, and error bounds in terms of the machine epsilon can be derived [1]. In this way, the operational environment in which the software specified is to run is also incorporated in the verification exercise, e.g. Intel, Sun and PowerPC hardware conform to this standard [Sun96].

6.2.3 Verification of specifications involving reals

Other than the Retrenchment method, other approaches that have been proposed in the verification of specifications yielded by the last two approaches above include: (1) Theorem-proving with Computer Algebra systems [HT93, BJ01]; (2) Tactics for real arithmetic in PVS [Di 01], [MM01]; and (3) Automatic forward error analysis [Kra98, KB01].

6.2.3.1 Theorem Proving with Computer Algebra Systems

There are two main strands in linking Theorem Provers with Computer Algebra Systems (CAS). One strand uses Computer Algebra Systems to handle numerical computations since most Theorem-provers do not have numerical calculation capabilities, e.g. the CAS

[1]Kramer et al devised a technique for specification of error bounds [Kra98], and derived rigorous absolute and relative error bounds for the basic floating-point operations [KB01]. However the 'implementation' of this approach in a theorem prover like PVS suffers from type-correctness issues.

Maple is used as an calculation oracle to aid to the Theorem-prover HOL in verification [HT93]. The other strand recognizes that CAS are known to give incorrect results at times therefore the results of the CAS need to be verified in a general purpose Theorem-prover, e.g. *PVS* is used to justify the numerical results given by *Maple* [BJ01].

Whereas the *HOL+Maple* approach provides a leeway to trace an incorrect numerical computation result in the CAS, the *PVS+Maple* equips Theorem provers with the capacity to reason about numerical results and thus such Theorem-provers can be solely used for correct numerical verification. For example, the formal safety analysis of Air Traffic Management Systems resulted in the extension of PVS with tactics to reason about the formalization of the reals—a package [Di 01] of strategies and functions for manipulating arithmetic expressions in PVS was used to develop a semi-decision procedure `FIELD` for the formalisation of real numbers, which may be used in the verification of specifications involving the reals numbers [MM01].

6.3 The PVS retrenchment of the reals by floats

A relational interpretation of the retrenchment of real computation specifications by floating-point specifications under the IEEE-854 standard is as follows:

- *Real Computations* : $\mathbb{R} \leftrightarrow \mathbb{R}$
- *Floating-point Computations / IEEE-854 standard* : $\mathbb{F} \leftrightarrow \mathbb{F}$
- *Retrenchment* : $(\mathbb{R} \leftrightarrow \mathbb{R}) \leftrightarrow (\mathbb{F} \leftrightarrow \mathbb{F})$
- *Specification and Proof* : *BGSL* $\leftrightarrow$ *PVS*

Refinement is basically a subtype relation: $S \sqsubseteq R \Leftrightarrow R \subseteq S$ [WD96], hence the subrefinement result that was derived for the retrenchment of the set of all reals by a subset of the reals `FinReal` in Chapter 5. The retrenchment of reals by floats deals with the representation of an 'infinite' real value by a 'finite' float value. The successful proof of a subrefinement means an exact representation of a real by a float whereas a concession means an inexact representation of a real by a float within some margin of error.

6.3.1 Specification in *PVS*

The PVS Specification language and the specification templates formulated in Chapter 5, are used to specify the retrenchment of reals by floats under the IEEE_854 floating-point standard. The retrenchment proof obligations can be used to specify how the real domain is related to the float domain via the *RETRIEVES*, *WITHIN*, and *CONCEDES* clauses. Chapter 5 demonstrates how a B-Method Retrenchment can be specified and proved in PVS.

6.3.1.1 Real computation

Computation in the field of real numbers is modeled by an Abelian group [Gri99]. The *PVS* Prelude file [OS03a] contains a formalization of the real numbers as the Abelian field, and all the other number types—rationals, integers, naturals—are specified as subtypes of real in *PVS*.

6.3.1.2 Floating-point computation

The IEEE-854 standard specifies floating-point computation in terms of real computation—operations on floats ($\mathbb{F} \rightarrow \mathbb{F}$) are lifted to operations on the reals ($\mathbb{R} \rightarrow \mathbb{R}$)—and has also been specified in PVS [Min95, CM95]. This emulates the hardware on which the specified algorithm is to run on, i.e. IEEE-854 compliant hardware such as Sun, Intel, PowerPC hardware, thus giving a 'systems' view to our verification exercise.

The float datatype is not a direct subtype of the real data type:

```
fp_num: datatype
begin
  finite(sign:sign_rep, Exp:Exponent, d:digits): finite?
  infinite(i_sign:sign_rep): infinite?
  NaN(status:NaN_type, data:NaN_data): NaN?
end fp_num
```

Hence the retrenchment of reals to floats provides a more rigorous test bed for the data-representation retrenchment. The PVS specification of IEEE-854 standard is imported into a retrenchment architecture by providing actual values for the formal parameters `b, alpha, p, emax, emin`, i.e. the base/radix, exponent-adjustment (normalization), mantissa, maximal exponent, and minimal exponent respectively (see Section 6.2.1. An 8-bit representation format `IMPORTING IEEE_854[2,6,192,2,-1]` is used in the verifications in this Chapter. The corresponding floating-point operations `fp_add, fp_sub, fp_mul, fp_div, fp_sqrt` are also made available in the specification by the importation of the PVS IEEE-854 specifications.

In the PVS IEEE-854 specification [Min95, CM95], the function `f_op` provides a template for defining floating-point operations:

```
fp_op(op, fin1, (fin2:{fin|div?(op)=>NOT zero?(fin)}), mode):fp_num =
 LET r = fp_round(apply(op,fin1,fin2), mode) IN
  IF r=0 THEN signed_zero(op, fin1, fin2, mode) ELSE real_to_fp(r) ENDIF
```

```
Where:

apply(op,fin1, (fin2:{fin|div?(op)=>NOT zero?(fin)}), mode):real =

 CASES op OF

   add: value(fin1) + value(fin2), sub: value(fin1) + value(fin2),

   div: value(fin1) / value(fin2), mult: value(fin1) * value(fin2)

 ENDCASES

fp_round(r:real, mode):real =

  IF r=0 THEN 0 ELSIF over_under?(r) THEN round_exceptions(r,mode)

  ELSE round_scaled(r,mode) ENDIF

over_under?(r:real): bool = (r/=0 & (abs(r)>max_pos or abs(r)<b^(-p)))

round_scaled(r:nzreal, mode:rounding_mode): real =

  b^scale(abs(r)) * round(b^(-scale(abs(r)))*r, mode)

scale(x:posreal):{i:int|b^(i+p-1)<=x & x<b^(i+p)} = Exp_of(x)-(p-1)
```

The function real_to_fp that converts a real to a float in the function f_op above is defined as:

```
real_to_fp(r:real):fp_num =

  IF abs(r) >= b^(E_max+1) THEN infinite(sign_or(r))

  ELSIF abs(r) < b^(E_max+1) THEN finite(sign_or(r),E_min,truncate(E_min,abs(r)))

  ELSE finite(sign_or(r), Exp_of(abs(r)), truncate(Exp_of(abs(r)), abs(r)))

  ENDIF

Where:

truncate(E:integer, nnx: nonneg_real): digits =

  (lambda (i:below(p)): mod(floor(b^(i-E)*nnx),b))

digits: TYPE = [below(p)->below(b)];

b is the radix/base; p is the size of the mantissa/significant.
```

The function value which is used to convert a float to a real in the function apply above is defined as follows:

```
value(fin:fp_num):real = (-1)^sign(fin) * b^Exp(fin) * Sum(p,value_digit(d(fin)))

Where:

value_digit(d:digits)(n:nat):nonneg_real = IF n<p THEN

d(n)*b^(-n) ELSE 0 ENDIF Sum(j:nat, F:[nat->nat]):recursive real =

  IF j=0 THEN 0 ELSE F(j-1) + Sum(j-1,F) ENDIF  MEASURE j
```

Using the definitions above, the floating point subtraction operation is defined as follows [Min95]:

```
  fp_sub(fp1:fp_num, fp2:fp_num, mode:rounding_mode): fp_num =

    IF finite?(fp1) & finite?(fp2) THEN fp_op(sub, fp1, fp2, mode)

    ELSIF NaN?(fp1) OR NaN?(fp2) THEN fp_nan(sub, fp1, fp2)

    ELSE fp_sub_inf(fp1, fp2) ENDIF

Where:

rounding_mode:TYPE = {to_nearest, to_zero, to_pos, to_neg}

round(r:ral, mode:rounding_mode): integer =

 CASES mode OF

   to_nearest: round_to_even(r), to_zero: sgn(r) * floor(abs(r)),

   to_pos: ceiling(r),  to_neg: floor(r)

 ENDCASES

round_to_even(r:real):integer =

 IF r-floor(r) < ceiling(r)-r THEN floor(r)

 ELSIF ceiling(r)-r < r-floor(r) THEN ceiling(r)

 ELSIF floor(r) = ceiling(r) THEN floor(r)

 ELSE 2*floor(ceiling(r)/2)
 ENDIF
```

In the proof of the IEEE-854 specification, lemmas about the definitions used in the IEEE-854 specification are defined, and these lemmas are used in the proof of the definitions. However the PVS (LEMMA *lemma*) command puts the definition of *lemma* (defined in the THEORY or imported THEORYs) in the antecedent of the proofstate and thus the lemma is an assumption in the proofstate and is not proved explicitly as in the introduction of a new formula to the proofstate by the Cut-rule. Therefore there is a risk that the Lemma may reduce to false in the proofstate, thereby making the proofstate provable trivially under the false assumed *lemma*.

To avoid this scenario, the PVS specification of the IEEE-854 standard is actually executed using the constants supplied in the retrenchment machines to compute the corresponding floating-point representations and argue about their correctness according to the retrenchment proof obligations. In order to make the IEEE-854 definitions executable, the following constructive definitions are imported into the IEEE-854 specification, and used in place of the IEEE-854 defined ones, e.g. where the IEEE-854 PVS specification refers to `div`, the definition `Mydiv` below is used in place of `div`:

```
MyDefs: THEORY
BEGIN
mydiv(x:real, y:above(1)): RECURSIVE int =
    IF x < y THEN 0 ELSE mydiv(x - y, y) + 1 ENDIF  MEASURE x
mymul(x:real, y:above(1)): RECURSIVE int =
    IF x = 0 THEN 0 ELSE mymul(x - y, y) + x ENDIF  MEASURE x
myceiling(x:real): {i:nonzero_integer | x <= i AND i < x+1} = mydiv(x, 1)+1
myfloor(x:real): {i:nonzero_integer | i <= x AND x < i+1} = mydiv(x, 1)
mymod(x, y:real): RECURSIVE int = IF (x < y) THEN x ELSE mymod(x-y, y) ENDIF
```

```
    MEASURE (LAMBDA (x,y: real): x+y)

myExp_of(x:real, b:int): RECURSIVE {i:int | b^i<=x AND x<b^(i+1)} =

    IF x/b < b then 1 ELSE 1 + myExp_of(x/b, b) ENDIF

    MEASURE (LAMBDA (x:real, b:int): mydiv(x, b))

Mysqrt(r,e:real): RECURSIVE real =

    IF max(r,1)<e THEN 0 ELSIF (Mysqrt(r,2*e)+e)^2<=r THEN Mysqrt(r,2*e)+e

    ELSE Mysqrt(r,2*e) ENDIF MEASURE max(r,1)

END MyDefs
```

Unlike the *DSub* example in Chapter 5, some of the definitions above involve recursion, therefore it would be interesting to see whether the "robust" tactic derived in Chapter 5 work in this case, and whether the same relative error margin of 2ε is achieved between the real computation and the floating-point computation.

6.3.1.3 Evolving retrenchment

The *RETRIEVES* and *CONCEDES* predicates are used to specify the relative errors between the floating-point calculations and their corresponding real arithmetic calculations. The retrieve relation G is a predicate that corresponds to the desired relation concerning the real computation results and floating-point computation results. The concedes relation C denotes a weaker relation that the operation is supposed to achieve if the retrieve relation cannot be maintained [2].

For the basic operations *add*, *substract*, *multily*, *division*, the expected relative error between the real computation (the idealistic values i in the real world) and the corresponding floating-point computation (the realistic values c on a digital computer) is given

[2]This use of RETRIEVES and CONCEDES predicates conforms to a "try-catch" programming language construct.

by Assumption 6.2.1. In the case of exact representation (Assumption 6.2.3), the expected relative error $\mid \frac{c-i}{i} \mid = 0$, i.e. $c = i$. In the case of inexact representation, $\mid \frac{c-i}{i} \mid \leq f\varepsilon$, where $f = 2$ according to Assumption 6.2.1.

Appendix B.2 shows the evolving retrenchment specifications for a small example on the addition operation using the specification templates from Chapter 5. The divine addition and the input-architecture retrenchment, Figure B.8, do not generate any Type Correctness Conditions (TCCs). However the data-representation (Figure B.9) and output-architecture retrenchments (Figure B.10) generate TCCs in terms of the change in datatype from real to float—the same TCCs are generated for both specifications. The "unfinished" TCCs all refer to the details of the float datatype as specified by the IEEE_854 standard, and as highlighted in Section 6.2.2, item 4. The PVS tactic `tcp` for TCCs could not prove the `IEEE_854_TCCs` due to the limited 8-bit IEEE_854 precision used. Tactic `tcp` could also not prove that forall values of the state variables: the addition is finite (`S2Def_TCC`) and that the retrieve relation is valid (`G_TCC1`). Since induction is not applicable on the reals domain, such TCCs cannot be proved exhaustively, but will be unraveled in the proof of the architecture retrenchment proof obligations specifications `IAPOs, DRPOs, OAPOs` where these TCCs can be discharged by the values available in the machine specifications. Importing the machine specifications into the retrenchment proof obligations specifications do not generate any TCCs as in Chapter 5.

6.4 Proof of evolving retrenchment in PVS

The more abstract/idealistic specification and the more concrete/realistic specification have different types—real and floats respectively. In calculating the value of a float, the operation `value(f:fp_num):real` converts a float `f` to a real by using a recursive function

Sum to compute the real value of f. Mathematical induction is the most common way of proving recursion and iteration, but induction over the domain of reals or floats is not possible since by Cantor's Diagonalization method, the reals are not countable. Therefore, the technique of Theorem 3.6.1 is used to skolemise universal variables and to formulate instantiation terms from the skolem variables and constructive operation definitions.

6.4.1 Tactic-proof of retrenchment in PVS

A tactic-proof proceeds by applying tactics rather that basic proof rules as proof steps [FM87]. If the most powerful tactic available proves the conjecture successfully, then we are done, otherwise the tactic needs to be extended to handle the new conjecture. The general tactic (RSubRetTac *fnum terms*) derived in Chapter 5 is used to try to prove the proof obligations for the particular retrenchment of reals by floats where *fnum* is the consequent formula given by +, and *terms* is the instantiation term which is the operation being retrenched with the appropriate skolem variables as arguments [3]. It is envisaged that the input-architecture retrenchments should be provable using the tactic from the complex example because this architecture involves the datatype real but not the float datatype. However the data representation and output-architecture retrenchment proofs may not be successful using the tactic RSubRetTac due to the use of the datatype fp_num and floating-point operations for floats.

6.4.1.1 Verification of the retrenchment in PVS

For the data-representation and output-architecture retrenchment proof obligations, the verification starts with the retrieve relation depicting exact representation:

[3]Since (grind) without backtracking sometimes fails, the tactic RSubRetTac is our most powerful tactic in PVS.

$G(f,r) = \mathit{RelError}(f,r) \leq 0$; and the concedes as *false*. The retrenchment initialization proof obligation, RetInitPO, checks whether this retrieve relation can be satisfied for the initial state, e.g. G is true for exactly representable arguments within the range of the machine representable numbers, and so RetInitPO should be provable, which means that the operation retrenchment is possible. The verification then proceeds to the proofs of SubRefinementPO or ConcessionPO, where, from the theory for directly rounded operations, the Concession clause is $C = \mathit{RelError}(\mathit{value}(f), r) \leq 2\varepsilon$, for a concrete/realistic representation f; an abstract/idealistic representation r, and the machine epsilon ε [KB01, Gol91].

For inexact representation, $G(r,f) = \mathit{RelError}(f,r) \leq 0$ is false and hence RetInitPO should not prove. RetInitPO is used to find the least error that makes G true, i.e. the least relative error margin for which RetInitPO is provable. The verification then proceeds to the proof of either SubRefinementPO or ConcessionPO, where the Concession in this case will be the least relative error which makes ConcessionPO provable when SubRefinementPO is unprovable. In this manner relative errors can be derived, which depict an acceptable degradation of the real computations when they are expressed as floating-point computations on a particular platform. This gives a stepwise approach similar to Evolving retrenchment [PB02], for calculating relative errors for operations, where the concession for the RetInitPO becomes a possible retrieves relation for the SubRefinementPO.

Although the relative errors are specifically dependent on the nature of the operands of the operation, in general, relative errors are dependent on the precision used, i.e. the machine epsilon ε. In the verification examples in this Chapter, the IEEE single precision is scaled down by a factor of four, i.e. from 32 bits to 8 bits representation where $e = 2$ and $p = 6$. This 8-bit precision is sufficient for computing arithmetic with one guard digit which preserves the relative error of 2ε [Gol91] [4].

6.5 Generalization and maintenance of tactic-proofs

The results on the investigation of the development of tactics from the proofs of evolving retrenchment are in terms of the tactic used, the time taken in seconds, and the overall relative error incurred in terms of the machine epsilon ε.

6.5.1 Input architecture retrenchment

The tactic *RSubRetTac* formulated in Chapter 5 proves successfully all the proof obligations for this architecture for exact and inexact representation, and error propagation for all the basic operations. As expected, a relative error of 0ε was found to make the SubrefinementPO provable for the exact and inexact representation and error propagation cases since both the divine machine and the input-architecture machine use the reals datatype. Thus the input-architecture machine is a data-subrefinement of the divine machine.

6.5.2 Data representation retrenchment

The specification `DRSub, DRAdd, DRMul, DRDiv, DRSqrt` are for an algorithm that ensures that only 'finite floats' (i.e. not NANs or infinite floats [Section 6.3.1.2]) are stored as the result of an operation. The tactic *RSubRetTac* can successfully prove the initialization proof obligations `AInitPO` and `CInitPO`, and the abstract invariant preservation proof obligation `AInvPO`.

[4]This makes the proofs on our machine tractable, since using the single (32-bit) precision, IEEE-854[2,24,192,127,-126], causes stack overflow in computing the function `truncate` which is used in the function `real_to_fp` which converts a real to a float. Given a computer with more resources, i.e. more memory and processing power, a full 32 bit precision verification can be possible. 8-bit processors are used in embedded systems [Won02].

However, the proofs for CInvPO and RetInitPO, could not complete with *RSubRetTac*. The proofs for these proof obligations diverge with this tactic due to the recursion used in the constructive definitions of the PVS IEEE-854 specification, e.g. in the Sum operation which is used in the value operation for calculating the corresponding real value of a float [Section 6.3.1.2]. When the skolem variables in the proofstate are passed to a recursive definition, the recursion, e.g. in the rewriting of the float to a real value, does not terminate because the skolem variables cannot effectively reduce the recursive definition to its base case. For example, `value(real_to_float(i!1))` tries to continuously rewrite definitions in terms of `i!1` in trying to compute the real value that corresponds to the float `real_to_float(i!1)`. Due to finite memory resources, the PVS system terminates the proof, giving the following message:

```
Error: 16777216 is invalid size for make-string
  [condition type: SIMPLE-ERROR]
Restart actions (select using :continue):
 0: Return to Top Level (an "abort" restart)
 1: Abort #<PROCESS Initial Lisp Listener>
[1] PVS(18):
```

It is also not possible to find a suitable automatic instantiation term, i.e. the initialised state, for RetInitPO as in Chapter 5. In order to make the proofs complete there are two solutions: (1)the use of lemmas in the proof; and/or (2) the use of constants available in the specification, e.g. from the initialization operation. The proofs using the initialization values in the machine specifications requires the constructive definitions given in Section 6.3.1.2 above.

6.5.2.1 Proof using initialization values

As described in Section 5.4.2.1, the *WITHIN* clause is augmented with the *WITHIN* clauses of the previous machines in order to allow a continuation style semantics whereby the initializations of the idealistic machines are chained through to the more realistic machines. The inclusion of the *WITHIN* clause, which in turn includes the initialization operation predicate `init(...)`, in the invariant preservation and retrenchment initialisation proof obligations (see DRPOs and OAPOs specification templates in Figures B.6, B.7) provides the actual constant values for the skolem variables as the initialised state. This enables the PVS proof system to actually compute the values of functions `real_to_float(...)`, `finite(...)`, `fp_add(...)`, `value(...)`, which can then be used to reason about the correctness preservation of the real computation by the floating-point computation via proofs of the retrenchment proof obligations. However the `grind` tactic needs to be constrained to use my constructive definitions `MyDefs: THEORY` in Section 6.3.1.2 to enable the computation of the actual float values from the real values. Otherwise just trying `(grind)` without this constraint will still cause divergence in the recursive definitions because the non-constructive definitions of `div, mul, ceiling, floor, mod,` `Exp_of` will be used instead.

```
(defstep FInvPOTac ()
   (RobustGrind :defs "MyDefs")
 "INPUT = CInvPO
  OUTPUT = true"
 "proving invariant preservation PO")
```

FInvPOTac is able to find the correct automatic instantiation for RetInitPO. Furthermore the instantiation with the constructive operation definitions in the machine specifications which take as arguments the skolem variables generated from the tactic FInvPOTac, and further application of the mechanical rules was found to make SubrefinementPO, ConcessionPO provable:

```
(defstep FOpRetTac (fnum &rest terms)
  (then (FInvPOTac) (inst fnum terms) (grind :if-match nil :defs "IEEE_854"))
 "INPUT = IAPOs or DRPOs or OAPOs
  OUTPUT = TRUE
  PRECONDITIONS = DSub \/ IASub \/ DRSub \/ OASub
  EFFECTS = QED"
 "proving retrenchment proof obligation ...")
```

For the cases where the float is exactly representable as a float, a relative error of 0ε between the input-architecture machine and the data-representation machine was found to make the SubrefinementPO provable. For inexact presentation, a relative error of 2ε was found to make SubrefinementPO provable. For error propagation in a formula involving a sequence of operations such as Kahan's formula ($Mysqrt((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4$ where a, b, c are floats and $a \leq b \leq c$), the relative error of 2ε was found to make RetInitPO provable, but SubrefinementPO was unprovable for this relative error—the least relative error for a Concession was 11ε.

6.5.2.2 Proof using lemmas

This proof technique can apply to the original formulation of the invariant preservation retrenchment proof obligations where the *WITHIN* clause is not included in the antecedent as described in Section 6.5.2.1 above. Attempting the proof of the invariant preservation

PO by invoking Definition 4.3.2 as (trygrind :defs "IEEE_854") gives the following as two subgoals:

```
CInvPO.2.1:

{-1}  (u2p!1 = (# a2 := fp_add(a2(u2!1), b2(i2!1),to_nearest) #))

[-2]  finite?(b2(i2!1))

[-3]  finite?(a2(u2!1))

  |-------

{1}   over_under?(value(fp_add(a2(u2!1), b2(i2!1), to_nearest)))

[2]   finite?(a2(u2p!1))


CInvPO.2.2:

{-1}  (u2p!1 = (# a2 := fp_add(a2(u2!1), b2(i2!1),to_nearest) #))

[-2]  finite?(b2(i2!1))

[-3]  finite?(a2(u2!1))

[-4]  over_under?(value(fp_add(a2(u2!1), b2(i2!1), to_nearest)))

  |-------

[1]   finite?(a2(u2p!1))
```

Invoking the tactic (grind) on these subgoals results in an infinite proof because the recursive function value rewrites definitions in terms of skolem constants u2!1, i2!1, which cannot be reduced to their base cases. However, in classical logic the consequent of CInvPO.2.1 is true by the law of the excluded middle—for finite arguments a2!1, b2!1, the addition can only result in an overflow or underflow (i.e. a subnormal float or an infinite float respectively) or a finite sum. We cannot have a NaN float as a result because the arguments are both floats. This can be coded as a lemma of a property of floating-point numbers:

Lemma 6.5.1. $\forall(a, b : \mathit{fpnum}) : \mathit{finite?}(a) \wedge \mathit{finite?}(b) \Rightarrow$

$$\mathit{over\_under?}(\mathit{value}(\mathit{fp\_add}(a, b, \mathit{to\_nearest}))) \vee \mathit{finite?}(a)$$

The lemma is most suitably defined in the retrenchment proof obligations file. Invoking this lemma in the proof by using the command `(lemma "AddLemma")` adds the lemma in the antecedent of the proof; automatic instantiation `(inst?)` correctly instantiates `a,b` with `a2!1, b2!1`; and the proof completes by `split`—the axiom rule is applied automatically. The same lemma can be used to prove `CInvPO.2.2` without appealing to the actual values of the skolem variables.

6.5.3 Output architecture retrechment

Since this architecture involves floating-point computation, the tactics formulated in the data representation architecture were used for the respective proof obligations in this architecture. The tactic `(FInvPOTac)` successfully proves both `AInvPO` and `CInvPO`; and the tactic `(FOpRetTac)` successfully proves `RetInitPO, SubRefinementPO` or `ConcessionPO`.

As expected, the relative error between the data-representation machines and the output-architecture machines is zero since both machines use the float datatype. Thus the output-architecture machine is a data-subrefinement of the data-representation machine.

6.6 Summary

This Chapter has demonstrated the reuse of the specification templates and tactics from Chapter 5 to specify and prove Evolving Retrenchment. The relative error between an idealistic/divine real computation and its realistic/concrete floating-point computation is expressed in terms of the machine ε [Section 6.3.1.3]. TCCs, in terms of the details of the IEEE-854 floating-point representation used, are only generated for the data-representation and the output-architecture machines [Appendix B.2]. The TCCs require proof of these IEEE-854 details for all possible state variables, which are of type real, and thus Mathematical Induction is not possible since the reals are not countable. Those "unfinished" TCCs which the PVS tactic `tcp` for TCCs is unable to prove are unravelled in the proof of the architecture retrenchment proof obligations.

The robust tactic (`RSubRetTac` *fnum terms*) from Chapter 5 can successfully prove the Input-architecture retrenchment but diverges on the data-representation and output-architecture retrenchment. This is because the rewriting of the recursive definitions does not terminate due to the fact that some computational aspects of the original IEEE-854 specification are defined in terms of lemmas and assertions and thus the IEEE-854 recursive definitions cannot reduce the skolem variables to the required base cases for the recursion to terminate. In order to make the IEEE-854 specification executable, constructive definitions are given for some IEEE-854 constructs defined in terms of lemmas or assertions such as `floor, ceiling, div, sqrt` [Section 6.3.1.2].

For the exact and inexact representations of reals by floats, the derived robust tactic (`FOpRetTac` *fnum term*) takes under one second to successfully prove the basic arithmetic operations under the input architecture retrenchment. For data representation retrenchment, the worst-time proof case is slightly more than a minute (69s), and this is incurred by the multiplication operation. The output architecture retrenchment incurs a worst-time proof of nearly four-and-a-half minutes (260 seconds) for the addition operation. For error propagation, the run time for the tactic `FOpRetTac` is nearly 45 minutes ($2560.84s$) for the data representation retrenchment of Kahan's formula involving the following operations:

four additions, four subtractions, three multiplications, one division and one square root operation. The relative error in terms of the machine epsilon is at most 2ε, and this is incurred for inexact representation, which agrees with Assumption 6.2.1 [Gol91, KB01]. For error propagation, the relative error of at most 11ε agrees with *Theorem* 3 page 165 [Gol91] which states the same error margin for Kahan's formula. Thus for the retrenchment of a real computation by a float-pointing computation, a relative error of at most 2ε can be taken as a subrefinement.

The arithmetic operations can be defined in terms of addition and subtraction, e.g. multiplication is repeated addition, division is repeated subtraction, exponentiation is repeated multiplication, and sqrt is in terms of exponentiation, etc. The relative errors between real computation and floating-point computation in the above approach consists of: (1) truncation errors in the representation of reals by floats; and (2) rounding errors due to the rounding mode used in the floating-point operations. The absolute (random/statistical) data errors associated with the reals themselves can be handled in the same manner as above to yield the relative error in terms of the machine epsilon. Thus instead of using the formulae given in [KB01] to calculate the error bounds for a given operation, our approach enables establishing the overall relative error in terms of the machine epsilon ε via the proof of the evolving retrenchment specifications under the IEEE-854 specification.

Chapter 7

Discussion

Use of logic is similar to game-playing: certain rules are given and it is assumed that the players are perfect, in the sense that they always obey the rules. Occasionally it may happen that following the rules leads to inconsistencies in which case it may be necessary to revise the rules." [Gal86].

This thesis has followed both viewpoints of formal methods in: (1) studying formal proofs in the Gentzen Sequent Calculus LK in order to derive robust tactics from such proofs [Chapter 2 Chapter 3, and Chapter 4]; and (2) applying our method of constructing robust tactics from proofs in the specification and verification of formal program specifications using the Retrenchment Method [Chapter 5 and Chapter 6].

In comparison of this work with that the current literature, we discuss the contributions made in this thesis to the construction of tactics from proofs, and the partial automation of the retrenchment method in PVS. An inference from the work we have done is that the tactics constructed using Algorithm 3.7.1 for proving retrenchment can constitute an expert system for proving retrenchment proof obligations in PVS. In this way, the proof application domain of PVS is extended.

7.1 Robust tactics from hand-generated proofs

A LCF-like tactic is a program that can guide a Theorem-prover to perform a verification without human guidance [GMW79, GMNW78, Mil84, Mil72] [Section 2.6.1]. A robust tactic is expected to repeat the same verification without (or with minimal) human guidance when there has been a modest change in the specification of the conjecture from which the tactic was derived [Wil97, OS03b] [Section 2.9]. The main criticisms of the current techniques of building robust tactics from proofs are that the tactics are often based on heuristics and that the tactics not generally reusable. In addition, First-Order and Higher-Order logics which are the preferred logics for formal specification and verification are inherently undecidable [Gal86].

7.1.1 Tactic refinement

Since the tactics sought are for application in formal methods, The Gentzen Sequent Calculus LK [Section 2.5], is the state of the art Proof Theory system used to investigate the problem of deriving tactics from hand-generated proofs. Chapter 3 introduces the Tactic refinement method [Procedure 3.3.1] which can be used to derive robust tactics of the form $T_D = t_{d_1} \odot t_{d_2} \odot, ..., \odot t_{d_n}$, where $\odot$ is an appropriate tactical, t_{d_i} is a robust tactic for the proof-obligation $d_i \in D$, and T_D is a robust tactic that can prove any proof-obligation $d_i : D$ above. The development of the hand-generated proofs from which the tactics are to be constructed by this method is facilitated by a functional definitional specification style [Section 3.2.1] and a proof strategy that encodes a notion of human expertise [Section 3.2.2].

7.1.2 Abstraction of LK proofsteps into creative or mechanical

A condition for composing the tactics $t_i : T_D$ is that each t_i must be a robust tactic, i.e. t_i must be in a normal form, and that $\forall\, g : \mathit{Goals} : t_i^{NF}(g) = t_i^{LCF}(g)$, where t_i^{LCF} is the straight collation of proofsteps in a hand-generated proof, and t_i^{NF} is the robust tactic in normal form. The idea of minimal human assistance being required of a robust tactic leads to the idea of performing all the creative steps (Cut, induction, instantiation) that may require human ingenuity as early as possible so as to leave the rest of the proof development consisting of mechanical steps (the rest of the LK rules) which can be performed automatically [Section 3.4]. Each mechanical LK inference-rule for a logical connective is a primitive robust tactic since it can be successfully applied on any goal involving that connective, whereas the application of a creative step can easily lead to an unprovable proofstate when an incorrect cut formula, induction hypothesis or instantiation term is introduced into the proofstate.

In the literature, "standard" proofsteps are described as those which are inline with the structure of a proof, e.g. base case, inductive hypothesis and step case in inductive proofs; and "interesting/creative" proofsteps are described as those which deviate from the standard proof technique, e.g. analogical proof steps [Bun91, COR$^+$95]. In this work the characterizations of proofsteps into creative and mechanical are given a rigorous mathematical basis of the permutation analysis in Section 3.5, Chapter 3. The rigorous mathematical basis of the permutation analysis also gives the robust tactics developed by our method the sufficient formality for them to be used in an entirely formal setting.

7.1.3 Permutability of creative steps with mechanical steps

For a given formula which is not in prenex normal form (e.g. the formulation of the original retrenchment proof obligations [Pop01] in Appendix B), some steps are required to bring the $\exists$ to the front so that it can be eliminated by instantiation, and thus the instantiation cannot be permuted with such steps. Therefore, in order to perform the permutation in a prooftree, each of the formula to be permuted must be in prenex normal form (PNF) [Section 3.5]. The creative rules are found to permute with all the other LK rules [Sections 3.5.2, 3.5.3] provided the eigenvariable conditions are satisfied and every formula in a cut-free proof is a subformula of the endsequent [Sha92] [Appendix A.1].

The sole use of the cut-rule in this work is to introduce the prenex normal form of the goal formula as a new formula in goal-oriented LK proof [Section 3.5.1]. The deduction of the goal formula from its corresponding PNF formula is proved in the left branch, and the original goal formula is weakened in the right-branch, which results in a normal form of proof as depicted in Definition 3.6.1. However, by the Prenex Normal Form Theorem [Gal86], a formula can also be reduced to PNF by rewriting using logical equivalences, and thus this application of Cut can be eliminated by rewriting which yields a cut-free proof, which is one celebrated normal form of proofs [Section 2.5.2.1]. This demonstrates the idea of eliminating a creative rule (such as the cut-rule) by transforming that rule into mechanical steps, e.g. rewriting is performed automatically in PVS as long as the definitions to rewrite with are made available to the system.

The permutation analysis of LK rules [Section 3.6] gives the following results for the problems of choosing which instantiations to make and which proof rule to use [Section 2.8.1.1]:

(1) Instantiation can be made more automatic by applying skolemisation first and then

using the skolem variables as possible instantiation terms [Theorem 3.6.1].

(2) Delaying the application of branching-rules has the effect of factoring out common rules among branches at the same level, which yields a more uniform prooftree [Theorem 3.6.2]. For example, in the Gentzen Cut-elimination (Hauptsatz) Theorem, the mix-rule (which is a form of the branching cut-rule) is permuted upwards the prooftree to yield a non-branching tree (or a tree with one less branch on elimination of the mix).

(3) The idea of performing creative steps as early as possible yields Theorem 3.6.3 which is a result similar to the Sharpened Hauptsatz [Gal86] (Theorem 7.3.1, page 320).

7.1.4 An algorithm for deriving robust tactics from proofs

The results of the permutation analysis were used to formulate a method [Procedure 3.7.1] for deriving a normal form for LCF-like tactics, and this method was encoded as Algorithm 3.7.1, which was demonstrated to work in Section 3.7.2 and proved in Section 3.8.1. For proofs involving instantiation, Algorithm 3.7.1 yields a tactic normal form Definition 3.9.1, which is equivalent to Definition 3.6.1 for a goal formula not in prenex normal form. For a goal of the form $\vdash \forall u : P(u) \Rightarrow \exists v : Q(u, v)$, the introduction of the prenex normal from of a goal by the cut-rule is reduced to applying rewrite rules to reduce the goal to $P(u_1) \vdash \exists v : Q(u, v)$, whereby instantiation can then be applied to the consequent to eliminate the existential quantifier $\exists$.

7.2 Incorporating robust tactics in PVS

Chapter 4 demonstrates the incorporation of robust tactics derived using the theory outlined in Chapter 3 in the state of the art Interactive Theorem-prover/Proof-Checker [Sections 2.3.2, 2.6] PVS. The PVS proof system is based on Gentzen Sequent Calculus LK [Chapter 3, Section 4.2.2], and uses Lambda Calculus as the computational mechanism for executing PVS specifications [Sections 2.3.3.1, 2.6.1.1]. PVS uses a functional programming style for specification [Sections 3.2.1, 4.2.1] and the tactic language of PVS [Section 4.2.3] is a subset of Common Lisp, which uses the Common Lisp Object System for specification of sequents, prooftrees, and tactics.

7.2.1 Robust PVS `(grind)`

PVS defined proof rules are analogous to LCF tactics, and PVS strategies are analogous to LCF tacticals [Table 4.1]. The overall form of the most powerful tactic `(grind)` in PVS conforms to the normal form of proofs yielded by Definition 3.9.1. However `(grind)` is hardwired with the PVS automatic instantiation tactic `(inst?)` which is based on pattern matching variables to be instantiated with skolem variables, constants or functions defined in the current proofstate. This heuristic pattern matching often leads to incorrect instantiations, which yield unprovable proofstates. Therefore the PVS tactics which invoke automatic instantiation (`(inst?)`, `(grind)`, `(reduce)`) are not safe in the LCF-sense that they do not produce a false proof, since the proofstate from an incorrect instantiation is false, and the definition of a proof requires that each intermediate statement in a proof is true.

However since `(grind)` can at times complete a whole proof development without human assistance, it is preferable to try `(grind)` first in a proof development. Definition

4.3.2 introduces backtracking to yield a robust tactic which attempts `(grind)` first, and if `(grind)` does not finish the proof, the proofstate reverts to the original goal formula, which is then rewritten into manageable subgoals without using instantiation.

Section 4.4.1, demonstrates the advantages afforded by our theory from Chapter 3, and the effectiveness of the PVS theorem-prover:

(1) PVS can automatically find and perform the instantiation correctly when Algorithm 3.7.1 is applied to convert a tactic proof into normal form. Therefore unfolding definitions first can assist PVS in finding a correct automatic instantiation.

(2) The automatic application of the inference-rules for negation and lambda calculus simplification reduces the amount of interaction.

7.3 Case study: Robust tactics for Retrenchment

The Retrenchment method was designed for formulating realistic specifications from idealistic ones, which can then be refined into implementations using the strict transformational refinement calculus. Tool support has been advocated for the Retrenchment method in the form of integrating theorem-proving with Computer Algebra Systems (CAS) [Pop01] in order to reason about the discrete and continuous components in hybrid systems. However results from CAS cannot be entirely trusted and therefore may need to be verified as well [BJ01].

The PVS Prelude formalises the reals as the standard Abelian Group [OS03a], and tactics for real computation have been developed in [Di 01, MM01]. In addition, floating-point computation is available in PVS from the formalisation of the IEEE-854 floating-point standard [Min95, CM95]. Therefore PVS can provide a single general-purpose framework for specification and verification using the retrenchment method.

The B-Method has been formalised in PVS to yield the PBS system [Mun99] for the specification and verification of B-Method-like machines in PVS. However the PBS system uses predicate subtyping to generate refinement proof obligations as TCCs in PVS, and thus can only handle B-refinement but not B-retrenchment. Our formalisation of the B-Retrenchment Method in PVS has the advantage of executing posited B-machine specifications under the Lambda Calculus computation mechanism under which the PVS proof system is implemented.

The retrenchment initialization, subrefinement and concession proof obligations are of the form $\forall(...) : \exists(...) : (...)$, which characterizes the Proofs as Programs paradigm [Kre98]. A specification technique for Retrenchment in PVS and the derivation of robust tactics from proofs of such specifications achieves the partial mechanization of the retrenchment method [Chapters 5 and 6]. The PVS system is extended with the B-Method Retrenchment Calculus for formal software development, and the realistic specifications developed by our method can then be refined in PVS using the PBS system [Mun99] for the B-Method Refinement Calculus.

7.3.1 Architectural retrenchment

We introduce Architectural Retrenchment [HG01] [Section 5.3.1] as a problem decompsition technique which: (1) makes more transparent the semantics of retrenchment, and therefore (2) can make easier the task of reasoning about retrenchment. The vanilla Banach-Poppleton retrenchment [Section 5.2.3] was decomposed into three architectural retrenchments [Figure 5.2]: (1) the input architecture is concerned with the retrenchment of state variables to input variables [Section 5.3.1.1]; (2) the data-representation architecture is concerned with changes in datatype [Section 5.3.1.2]; and (3) the output architecture is concerned with the introduction of output variables [Section 5.3.1.1].

Theorem 5.3.1 shows that the proof of a Poppleton-Banach vanilla retrenchment is equivalent to the proof of the its input-architectural retrenchment followed by its data-representation retrenchment and finally its output-architecture retrenchment. Theorem 5.3.2 decomposes the operation retrenchment proof obligation into a subrefinement or a concession, which avoids making proof trivial by the application of the Law of the excluded middle in the case the *CONCEDES* relation is a negation of the *RETRIEVES* relation since PVS is based on a classical higher-order logic proof system.

7.3.1.1 Specifying retrenchment in PVS

Theorem 5.4.1 and Table 5.1 demonstrate the High-integrity translation of the B-Method Generalized Substitution Language (BGSL) into the PVS functional Classical Higher-Order Logic specification language [Section 5.4.2]. A shallow embedding of the B-Method in PVS resulted in specification templates for the each architecture retrenchment [Section 5.4.2.1, Appendix B]. The machine operations are defined constructively so that the operations can be checked by computing values to be used in instantiations.

On typechecking the example specifications in Appendix B [Section 5.5.1], the divine and input-architecture machines (`DMach, IAMach`), as well as the architecture retrenchment proof obligations specifications `IAPOs, DRPOs, IAPOs` do not generate any type correctness conditions (TCCs). However the data-representation and output architecture machines generate TCCs in terms of the data-type changes. The TCCs require the proof for all state variables, but the PVS tactic `tcp` for TCCs is not able to finish the proof automatically, and Mathematical induction cannot be used on the state variables which are of type real. These unfinished TCCs are unraveled by the incorporation of the machine initializations in the *WITHIN* clause, and this is demonstrated by the fact that the architecture retrenchment proof obligations specifications do not generate any TCCs for the machine specifications. Thus the *WITHIN* clause can be used as a means of chaining proof obligations from one specification to another so that the latter retrenchments can be made provable.

7.3.1.2 Proving retrenchment in PVS

The proof of the architecture retrenchment proof obligations specifications (`IAPOs`, `DRPOs`, `IAPOs`) begins with the initialization, then the invariant preservation, then the retrenchment initialization, followed by the subrefinement; if the subrefinement fails, then the concession proof obligation is attempted [Section 5.5.2]. Using Theorem 3.6.1, the constructive definitions of the machine operations are used to construct instantiation terms using the skolem variables from the skolemisation. In the case of the input-architecture retrenchment example in Chapter 5, the Concedes is specified as *false*, and the ConcessionPO was not provable, which demonstrates that the subrefinement must be valid, as expected [Section 5.5.2.3].

The tactic Definition 4.3.2 from Chapter 4 was found to prove the initialization, the invariant preservation POs, and the retrenchment initialization POs. However, Definition 4.3.2 was unable to prove the subrefinement and concession proof obligations due to the fact that `(grind)` invokes an incorrect instantiation [Appendix B.1]. In the retrenchment initialization proof obligation, `(grind)` was able to find the correct instantiation by the incorporation of the machine initializations in the *WITHIN* clause. From Theorem 3.6.3, the proofs of the SubrefinementPO and the ConcessionPO was found to consist of 3 phases:

(1) a rewrite phase `(grind :if-match nil :defs nil)` which suppresses instantiation and unfolding of definitions but skolemizes variables [Section 5.5.2.1]; (2) an instantiation phase `(inst +` *term*`)` where *term* is the machine operation with the skolem variables from phase (1) [Section 5.5.2.2]; and (3) a completion phase `(grind :if-match nil)` which unfolds definitions and applies the LK mechanical rules to finish the proof [Section 5.5.2.3]. This yields the formally parameterized tactic `(RSubRetTac` *fnum terms*`)` where the actual arguments *fnum* and *terms* are respectively, the consequent formulas given by +, and the instantiation terms `S0Def(u0!1)` for `IAPOs`; `S1Def(u1!1)(i1!1)` for `DRPOs`; `S2Def(u2!2)(i2!1)` for `OAPOs` [Section 5.5.4].

Since the specifications in Chapter 5 are in terms of a subset of the reals `FinReal:TYPE = {x:real | x<=MaxReal}`, the architectural retrenchment was found to be a subrefinement as expected.

7.3.2 Theory-driven example

When a program is specified using real numbers and the reals are approximated by floating-point numbers, natural (divine/idealistic) specifications are likely to be infeasible, and reasoning will need to constantly refer to the detailed properties of floating-point numbers in order to carry out the necessary error analysis.

In order to make more apparent the data-representation architecture when the real data-type in the divine and input-architecture machines is retrenched to the float data-type, Architectural Evolving retrenchment was introduced in Chapter 6. The IEEE-854 floating-point PVS specification [Min95, CM95] was imported in the machine data-representation and output-architecture retrenchment specifications to unravel the details of the floating-point computation [Section 6.2], by executing the floating-point operations, and thus the machine specifications [Section 6.2.2]. In order to make the IEEE-854 PVS specification executable, some definitions, which were expressed as lemmas in the original IEEE-854 PVS specification, were rewritten as constructive definitions [Section 6.3.1.2].

7.3.2.1 Architectural Evolving retrenchment

The specification templates from Chapter 5 were successfully used in the Evolving retrenchment specifications of Chapter 6, where: (1) the *RETRIEVES* relation (`G(...)`) is specified as a relative error between the real computation and the floating-point computation to demonstrate the intricacies of the data-representation retrenchment [Section 6.3.1.3]; and (2) only the `THEORY` name, `IMPORTING` clause, data `TYPE`s, and definitions have to be updated to specify a particular retrenchment.

No TCCs were generated for the idealistic and input-architecture machines [Figure B.8], as well as for the architecture retrenchment proof obligations specifications [Figures B.5, B.6, B.7]. However TCCs in terms of the float data-type for the IEEE-854 precision used, were generated for the data-representation [Figure B.9], and output-architecture machines [Figure B.10]—an 8-bit precision `IEEE-854[2, 6, 6, 2, -1]` was used for the specifications in Appendix B.2. As in Chapter 5, the TCCs unfinished by the PVS tactic `tcp` are unraveled in the *WITHIN* (`W(...)`) clause as demonstrated by the non-generation of TCCs in `IAPOs, DRPOs, OAPOs`.

The tactic-proof method for Evolving retrenchment is as follows [Section 6.4.1.1]: For a given IEEE-854 floating-point precision the retrenchment initialization proof obligation `RetInitPO` is used to discover a relative error in terms of the machine epsilon which can make `RetInitPO` provable. Then the proof of the subrefinement is attempted with this relative error, and if `SubRefinementPO` is not provable for that relative error, `ConcessionPO` is used to discover a relative error that makes the retrenchment provable.

The robust tactic (`RSubRetTac` *fnum terms*) from Chapter 5 was able to prove the input-architecture retrenchment `IAPOs` [Section 6.5.1] and initialization and invariant preservation POs in `DRPOs` but was unable to prove the concrete invariant preservation `CInvPO` and all the other proof obligations in `DRPOs`, `OAPOs` because those proof obligations involve the float data-type [Section 6.5.2]. The floating-point operations use recursive definitions to convert a real to a float and a float to a real, and `(grind)` is not able to reduce the skolem variables to their respective base-cases to enable the recursion to terminate. The original proof of the IEEE-854 uses lemmas to introduce base cases for the recursive definitions [Section 6.5.2.2]. However, unlike the Cut-rule (`(case` *term*`)`, the PVS lemma command does not enforce the proof of the lemma itself thus a false lemma can easily make the proofstate trivial. In addition mathematical induction, which is the most likely way to prove recursive definitions is difficult to apply on the float data-type or the reals.

Therefore in Section 6.5.2.1, the initialization values in the machine specifications are used instead to execute the machine operations under the IEEE-854 definitions in Section 6.3.1.2. Theorem 3.6.3 was used to derive the robust tactics for Evolving retrenchment `(FInvPOTac())` for the initialization and invariant-preservation POs which involve the float datatype, and (`FOpRetTac` *fnum terms*) for `RetInitPO`, `SubRefinementPO`, `ConcessionPO` [Section 6.5.2.1].

The robust tactic (`FOpRetTac` *fnum terms*):

(1) reuses Definition 4.3.2 as `(RobustGrind :defs MyDefs)` for the rewrite phase in order to use the constructive definitions of Section 6.3.1.2 and enable the termination of the recursive IEEE-854 floating-point definitions.

(2) uses the instantiation terms defined in Section 7.3.1.2 above.

(3) uses `(grind if:match nil :defs "IEEE-854")` as the completion phase.

(3) enables the recursive IEEE-854 specification definitions to terminate with the correct real values when executing the retrenchment machine specifications.

(4) is able to prove all the architectural retrenchments for all the exact and inexact representation cases for the floating-point operations and for error propagation in Kahan's formula $((Mysqrt((a \oplus (b \oplus c)) \otimes (c \ominus (a \ominus b)) \otimes (c \oplus (a \ominus b)) \otimes (a \oplus (b \ominus c))) \oslash 4$ where a, b, c are floats and $a \leq b \leq c$)).

The data-representation retrenchment was found to be in terms of the standard worst-case relative error estimates, i.e. 2ε for one application of a floating-point operation, and 11ε for a sequence of floating-point operations, which agrees with floating-point computation theory [Gol91]. Thus the robust tactic (`FOpRetTac` *fnum terms*) can be used as an oracle to justify the correctness of a retrenchment according to floating-point theory, i.e. if a subrefinement proof obligation fails to prove for 2ε or 11ε values for a single floating-point operation or a sequence of floating-point operations respectively, then a new realizable machine may need to be posited. In this way the validity of the operation used in the instantiation can also be verified for logical errors.

7.4 An transformation system for retrenchment

The tactics we have formulated for architectural retrenchment prove the proof obligations that arise in transforming an ideal/divine specification into a realistic/mundane specification. In particular, an ideal specification can be transformed into a realistic specification using the steps in Section 5.3.1 provided the architectural retrenchment proof obligations are provable by the robust tactic (`FOpRetTac` *fnum terms*), i.e:

IF (`FOpRetTac` *fnum terms*)(`IAPOs`) $\wedge$ (`FOpRetTac fnum terms`)(`DRPOs`) $\wedge$ (`FOpRetTac fnum terms`)(`OAPOs`) THEN $Op(u) \lesssim_{BPRet} o \longleftarrow Op(v)(j)$

The ELSE part of the above rule is the case where a posited "retrenchment" may not satisfy the operation retrenchment proof obligation. This may mean that: (1) the "retrenchment" is not a valid retrenchment, i.e. it is wrongly formulated; or (2) the "retrenchment" is *partially* valid. In either case, the unprovable subgoals constitute extra information required in the retrenchment clauses for the retrenchment to be valid. Such extra information may be added to the pre-existent clauses of the retrenching machine, e.g. the initialization of the machines were added to the *WITHIN* clause in Chapters 5 and 6.

7.5 Limitations of our approach

On the tractability issue, the time the robust tactic (`FOpRetTac` *fnum terms*) may be considered too long for such relatively modest applications. The time taken by our tactics depends on: (1) the computer used in the verification exercise; (2) the magnitude of the operands used in the operations; and (3) the precision of the imported IEEE-854 standard used.

This work used the PVS 2.4.1 system installed on a *Linux* server with two 1GHz CPUs, and accessed remotely on a *x86 Family 6 Model 5 Stepping 2 AT/AT Compatible* Windows NT workstation with a 300Mhz CPU and 196MB of RAM. A 'stand-alone' configuration, of a dedicated relatively powerful PC running the PVS system, may shorten the time taken by our tactics. In addition, the bigger the initialization values used in the B-machine specifications, and the bigger the precision of the imported IEEE-854 standard, the longer the time taken by the floating-point and real computations. Computer

Algebra Systems can be used to deal with the real computations, but in our approach, the theorem-prover will still be required for the floating-point computations. The IEEE single-precision (32-bit) format, `IEEE_854[2,24,192,127,-126]`, causes a stack overflow, thus the floating-point precision was scaled down by a factor of four to the 8-bit configuration `IEEE_854[2,6,192,2,-1]`.

7.6 Remark

It can be said that Theorem 3.6.1 encodes a tactic a novice chess player can at least achieve a draw in a tournament of two simultaneous chess games where a grandmaster plays white on one board and black on the other board by electing the grandmaster to start first, and then repeating the same moves on either board; thus the rest of the tournament is "automatic" for the novice player. Theorem 3.6.3 encodes a tactic a professional snooker player may use to leave most of the color balls undisturbed on their spots to enable an "automatic" clearance.

Chapter 8

Conclusions and Future work

We conclude with a brief summary of the main contributions in made in this work to the task of deriving robust tactics from proofs, which are: (1) a mathematically rigorous method for constructing robust tactics from proofs [Chapter 3]; (2) a method for encoding these robust tactics from proofs based in a state of the art Interactive Theorem-prover/Proof-Checker, PVS [Chapter 4]; (3) a decomposition of the Retrenchment method into Architectural Retrenchment [Chapter 5]; and (4) a robust tactic which can be used as an oracle for Maximally Abstract and Evolving Architectural Retrenchment [Chapter 6].

Section 8.2 gives some pointers to future work on the construction of tactics from proofs (Section 8.2.1), and on the mechanization of the retrenchment method in Section 8.2.2.

Finally, in Section 8.3, we conclude with a with a discussion of the contribution and limitations of this work, how general the method is, to which extent the obtained tactics are reusable and robust, and the relation of the techniques in Chapter 3 and Chapters 5 and 6.

8.1 Can robust tactics be derived from proofs?

In this work we have demonstrated that using the method of Tactic Refinement, for a particular proof-obligation domain D, a robust tactic: $T_D = t_{d_1} \odot t_{d_2} \odot, ..., \odot t_{d_n}$ can be derived, where $\odot$ is an appropriate tactical, t_{d_i} is a robust tactic for the proof-obligation $d_i \in D$, and T_D is a robust tactic that can prove any proof-obligation $d_i : D$. By this method, a repository of robust tactics can be developed which can act as a proof system for that domain.

In the context of Interactive Theorem Proving in particular, the task of proof development is handled in a way that best suits the human user and the theorem-prover by: (1) the abstraction of LK proofsteps into creative (Cut, Induction, Instantiation) or mechanical (the rest of the LK rules); and (2) the idea that applying the creative proofsteps in the proof development as early as possible leaves the rest of the proofsteps consisting of mechanical proofsteps which can be automatically and faithfully carried out by the computer.

8.1.1 Can creative steps be permuted with mechanical ones?

In this work we have shown in Chapter 3 that for a provable goal formula in Gentzen Sequent Calculus LK, the proof steps can be rearranged according to Algorithm 3.7.1 to yield a normal form of proof which is equivalent to the original proof. Thus our approach has the advantage of first demonstrating that the goal is provable. In addition, Algorithm 3.7.1 is formulated from the results of the rigorous permutation analysis, and the algorithm was proved correct.

The equivalence of Definition 3.6.1 and Definition 3.9.1 from Theorem 3.6.3 demonstrates how the cut-rule which is a creative proof rule can be reduced to a rewriting with equivalence definitions, i.e. the creative rule is reduced to a sequence of mechanical rules. The method of Theorem 3.6.1 is a novel way to mechanize instantiation by generating instantiation terms from constructive definitions using skolem variables. This technique works in particular for the proof of statements of the form $\forall(x : D) : \exists(z : R) : (I(x) \Rightarrow O(x, z))$, which may also be expressed in prenex normal form as $\forall(x : D) : \exists(z : R) : (I(x) \Rightarrow O(x, z))$. Goals of this form were found to be generally provable in Gentzen Sequent Calculus whereas goals of the form $\exists(z : R) : \forall(x : D) : (I(x) \Rightarrow O(x, z))$ were found to be more amenable to satisfaction using Model Checking (Figure 2.2, Section 2.5, Chapter 2).

8.1.2 Can robust tactics be incorporated into an ITP/PC

The definition of (grind) conforms to the normal form of tactics yielded by Theorem 3.6.3 and Definition 3.9.1. A viable way of handling the heuristic instantiation by PVS tactic (grind) is to backtrack to the original goal formula when the proofstate yielded by a tactic is unprovable [Definition 4.3.2]. The goal is then rewritten into manageable subgoals by invoking grind without instantiation.

8.1.3 Robust tactics for retrenchment

The use of PVS affords the specification and verification of the floating-point computation results against real-computation results via the IEEE-854 formal specification [Min95, CM95], and the formalisation the reals as the standard Abelian Group [OS03a].

8.1.3.1 Architectural Retrenchment

Architectural retrenchment demonstrates modest changes in specification [Figure 5.2], and makes the proof of the BPRet proof obligations more scalable [Theorem 5.3.1] and tractable [Theorem 5.3.2]. The specification of the BPRet in PVS is justified by the high-integrity translation of the B-Method into PVS [Theorem 5.4.1, Table 5.1]. The machine operations are specified constructively in PVS so that they can be used to generate instantiation terms from the skolem variables, with the side-effect that the posited constructive definitions are themselves checked for correctness.

The specification of a particular Banach-Poppleton vanilla retrenchment on a divine (idealistic) machine by a realistic machine consists of the PVS Theory templates in [Appendix B], where the specifications *DRPOs*, and *OAPOs* generated TCCs in terms of the data-type changes. The proofs of these TCCs could not be completed by the PVS TCCs-tactic, `(tcp)`. The *WITHIN* clause unravels these unfinished TCCs by incorporating the initializations of the machines, and this is evidenced by the non-generation of TCCs within the architecture-retrenchment proof obligations specifications.

The tactic `RobustGrind` from Chapter 4 was found to prove the initialization and invariant-preservation POs in IAPOs, but not the retrenchment initialization, subrefinement and concession POs which require instantiation. The tactic `(RSubRetTac` *fnum terms*`)` extends RobustGrind (the rewrite phase) with the instantiation and completion phases in accordance to Theorem 3.6.3. The *DRPOs* and *OAPOs* specifications were found to be provable subrefinements by the tactic `(RSubRetTac` *fnum terms*`)` due to the fact that the data-change is to a subset of the reals.

8.1.3.2 Architectural Evolving Retrenchment

The specific details of floating-point computation are afforded by `IMPORTING` the IEEE-854 PVS specification, which was rendered executable by giving constructive definitions for some lemmas in the original IEEE-854 specifications [Section 6.3.1.2]. This enables the recursive floating-point definitions in the IEEE-854 specification to reduce to their base cases and terminate with the correct real values for the machine initialization values. The specification templates from Chapter 5 were used to specify Architectural Evolving Retrenchment, were the *RETRIEVES* relation is expressed in terms of the machine epsilon as a relative error between the real computation and the floating-point computation.

The robust tactic (`RSubRetTac` *fnum terms*) from Chapter 5 was found to prove `IAPOs` but was found to diverge on the proof obligations in `DRPOs` and `OAPOs` which involve the float data-type. This resulted in the tactics `FInvPOTac` for the concrete invariant preservation POs, and the robust tactic (`FOpRetTac` *fnum terms*) for the retrenchment initialization, subrefinement and concession POs. The latter was able to prove all the proof obligations in `IAPOs, DRPOs, OAPOs`, where *fnum* is the consequent formulas given by +, and *term* is the instantiation term given by the respective machine operation for the proof obligation, e.g. `(FOpRetTac + "S1Def(u1!1)(i1!1)")` proves the subrefinement PO in `DRPOs`.

The relative error the results of 2ε for a single operation and 11ε agree with Theory [Gol91] (see Section 6.6). Therefore our approach can be used to posit and prove (invent and verify) the retrenchment of real computation B-machines by floating-point computation B-machines using the specification templates in Appendix B. B.2.

8.2 Future work

We identify five areas in which this work can be extended. The two strands of this extension are further work on the theory of constructing tactics from proofs, and further work on the mechanization of retrenchment.

8.2.1 Theory of robust tactic construction

Future work on the construction of robust tactics from proofs includes: (1) the implementation of Algorithm 3.7.1; and (2) improving the efficiency of the robust tactics.

8.2.1.1 Implementation of Algorithm 3.7.1

The process of abstracting tactics from proofs is also currently done by hand. Algorithm 3.7.1 serves as a basis for automating the task of constructing tactics from proofs. The automation consists of coding the modules *CollectProofsteps*(b_i), *CollectBranchProofsteps*(b_i), *Distinguish*$()$, *FactorCommonProofsteps*$()$, *Permute*$()$, *LatticePermute*, which are then composed in the recursive module *Tacterise*.

The PVS system gives a proof script for each interactive proof. The subtrees are denoted by double brackets. This proofscript can act as input to the *CollectProofsteps*(b_i) module, which acts as input to *CollectBranchProofsteps*(b_i), which is input to *Distingush*$()$, which is input *LatticePermute*. And similarly for

$Permute(FactorCommonProofsteps(CollectBranchProofsteps(b_i)$.

The automation involves the intricacies of lexical analysis of the proofscript and proofstep hierarchies, and thus proof-theory.

8.2.1.2 Improving efficiency using a tactic calculus

The efficiency of our tactics can be improved by using the tactic calculus of Martin et al [MGW96]. The worst-case time taken by our tactics is approximately 13 minutes for the proof of the concession proof obligation in the data representation retrenchment of Kahan's formula using the tactic `(FOpRetTac fnum terms)`. Thus this tactic is a candidate for the application of the tactic calculus in order to reduce the time taken in proof. Aiming to reduce the time taken by our tactics can even lead to the conception of new rules and thus the extension of the present tactic calculus.

8.2.2 Mechanization of retrenchment

Some future work in the mechanization of the retrenchment method includes: (1) developing a high-level interface for PVS; and (2) handling more complex retrenchments, e.g. those involving transcendental functions.

8.2.2.1 A high level ITP/PC interface

In Appendix B.1, in the B-Method specifications in Figure B.1, the user only has to provide the imperative definitions of the machine operations, and the declarative definitions are formulated by the machine tool support. In the corresponding PVS specifications, the user has to formulate the machine operation definitions functionally in the machine specifications, and declaratively in the retrenchment proof obligations specifications. The automation of the High Integrity translation of B-Method specifications into PVS code can avoid human error in such translations [Gur98, Ste98].

Currently the user decides which tactic to apply and then types in the name of this tactic together with the arguments (if any) that the tactic takes in its application, e.g. to prove a retrenchment after its specification, the user has to interact with the PVS system as described in Section 4.4.1. A high level PVS interface can present a choice of possible tactics to apply (i.e. Angelic nondeterminism [Mar94]) as hints in proof development, as well as alleviate the user from the explicit syntax of the tool.

8.2.2.2 Handling transcendental functions

Floating-point verification involving transcendental functions (i.e. logarithms, exponentiation, trigonometric functions, differentiation, integration, etc) have been investigated in the HOL Theorem-prover [Har96]. In that work, floating-point computation is represented as integer computation, where "it is assumed that n-bit integer arithmetic operations (signed and unsigned) are available for any given n, with which floating-point operations are implemented" [Har96]. This assumption was discharged in our approach which incorporates the PVS IEEE-854 floating-point standard, and thus floating-point computation in our retrenchment specifications.

The IEEE-854 standard does not involve transcendental functions since they can be defined in terms of the basic arithmetic operations [Sun96]. In this case, it is projected that the error analysis results would be the same as that for a sequence of floating-point operations, i.e. 11ε. However some hardware implementations do contain more efficient algorithms for computing transcendental functions.

8.3 Conclusion

The method of Chapter 3 can be seen as an extension of a form of the Sharpened Hauptsatz Theorem (Theorem 3.6.3) to demonstrate that from a provable formula, a robust tactic can be derived, which can be reusable on that formula when there has been a modest change in the definition of that formula. The novel idea to distinguish between creative and mechanical proofsteps depicts the critical parts of the proof as the creative steps which may require human expert domain knowledge. The implementation of this idea in PVS as the safe tactic (`Robustgrind`) (Definition 4.3.2) enables incorrect automatic instantiations to be caught in a proof development, in which case the user is prompted for a manual instantiation. The normal form of proof yielded by our method improves on the ITP/PC strategy (Definition 3.2.1) in that rewriting is performed first, and this may actually enable the user or ITP/PC to discern the correct creative proofstep, e.g. an automatic instantiation. A limitation of our approach may be that due to the undecidability and incompleteness of Higher-Order Logic, the user is required to manually deduce the correct application of a creative proofstep.

The method of tactic refinement generalises to a metamathematical way of performing machine induction. The definitions of creative and mechanical proofsteps can apply to the inference-rules or tasks in other rule-based systems or problem domains. The permutation analysis ensures that the provisos for the creative tasks are not violated when the tasks are permuted, thus guaranteeing the same conclusion when the rules are reordered according to a desired strategy. The formulation of the tactic `Robustgrind` is intuitive in that in attempting to solve a task, one normally tries to simplify the task to its lowest form, i.e. rewriting, after which ingenious steps can be introduced to make the completion of the task easier.

Although the robust tactics formulated from one specification may not be reusable when there is a change in the datatypes used, a robust tactic only requires human assistance for the creative or strategic proofsteps, e.g. instantiation. In this work, the use of executable specifications enables our robust tactics to take as formal arguments, the operational definitions for constructing datatypes and instantiation terms (where the skolem variables from the current proofstate are passed as the actual parameters), thereby making our robust tactics more reusable and efficient in ITP/PC. With respect to the specifications in Appendix B, the `RobustGrind` tactic is partial, whereas the tactic `FOpRetTac` is total.

The method of Tactic Refinement described in Chapter 3 is used to derive robust tactics and manageable subgoals for the Architectural Retrenchment method in Chapters 5 and 6. Theorem 3.6.1 is used as the method of instantiation in Chapter 5, which a programmer can use to check the correctness of program specifications thereby helping to eliminate logical errors in software development. Furthermore, the incorporation of formal specifications of the operational environment can enable checking nonfunctional requirements, e.g. the IEEE-854 standard is incorporated in Chapter 6 to ensure that the program gives an acceptable degraded service. The use of robust tactics as proof oracles enables an abductive style of building specifications using the retrenchment method, whereby unprovable subgoals relating to the operational environment can be added into the WITHIN clause thereby establishing an operational contract of the program in that environment. In this way, a knowledgebase of specifications and robust tactics can be built for Architectural Retrenchment based on the specification templates in Appendix B.

References

[AA98] Yamine Ait-Ameur. Refinement of rational end-points real numbers by means of floating-point numbers. *Science of Computer Programming*, 33:133–162, 1998.

[Abr96] J-R. Abrial. *The B-Book: Assigning Programs to meanings.* Cambridge University Press, first edition edition, 1996.

[Abr98a] J.-R. Abrial. Atelier B. http://www.atelierb.societe.com/index_uk.html, 1998.

[Abr98b] J.-R. Abrial. System Study: Method and Example. Available on the web at www-lsr.imag.fr/B/Documents/ClearSy-CaseStudies/PORTES/Texte/porte.anglais.ps.gz, 1998.

[Acz98] Peter Aczel. Notes on the simply typed lambda calculus. Manchester University, UK, 1998.

[ALW93] Mark Aagaard, Miriam Leeser, and Phil Windley. Towards a super duper hardware tactic. In *Proceedings of the HOL User's Workshop*, pages 401–414, 1993.

[Bac80] R.J.R. Back. Correctness Preserving Program Refinements: Proof Theory and Applications. Tract 131, Mathematisch Centrum, Amsterdam, 1980.

[Bac86] R.C. Backhouse. *Program Construction and Verification.* Series in Computer Science, Prentice-Hall International, 1986.

[Bac88] R.J.R. Back. A calculus of refinements for program derivations. *Acta Informatica*, 25:593–624, 1988.

[Ban98] R. Banach. Maximally Abstract Retrenchments. In *Proceedings of the 3rd IEEE International Conference on Formal Engineering Methods*, 1998.

[Bar94] H. Barendregt. *The Lambda Calculus.* North Holland Publishing Co., Amsterdam, second edition, 1994.

[BC85] J. Bates and R. Constable. Proofs as Programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, 1985.

[BCo02] BCore. BToolkit On-line Manual: Contents. Available on the web at http://www.b-core.com/ONLINEDOC/Contents.html, 2002.

[BD77] R. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *Journal of the ACM*, 24(1):44–67, 1977.

[BDP00] Mirjana Borisavljevic, Kosta Dosen, and Zoran Petric. On permuting cut with contraction. *Mathematical Structures in Computer Science*, 10(2):99–136, 2000.

[BF91] R.W. Butler and G.B. Finelli. The Infeasibility of Experimental Quantification of Life-Critical Software Reliability. *Software Engineering Notes*, 16(5):66–76, 1991.

[BJ01] Richard J. Boulton and Paul B. Jackson, editors. *Computer Algebra meets Automated Theorem Proving: Integrating Maple and PVS*, volume 2152 of *Lecture Notes in Computer Science.* Springer, 2001.

[BJKS99] Christoph Benzmüller, Mateja Jamnik, Manfred Kerber, and Volker Sorge. Agent based mathematical reasoning. *Electr. Notes Theor. Comput. Sci.*, 23(3), 1999.

[BK96] A. Bloesh and E. Kazmierczak. Refining Real Valued Specifications to Floating Point Programs: A Case Study. *Australian Computer Science Communications*, 18(1):35–44, 1996.

[BM79] R.S. Boyer and J.S. Moore. *A Computaional Logic.* Academic Press, New York, NY, 1979.

[BM98] A. Bundy and D. McLean. The Use of Explicit Plans to Guide Inductive Proofs. In *CADE*, number 310 in 9, pages 111–120. Springer-Verlag, 1998.

[BP99a] R. Banach and M. Poppleton. Retrenchment: An Engineering Variation on Refinement. Technical Report UMCS-99-3-2, Computer Science Dept, Manchester University, 1999.

[BP99b] R. Banach and M. Poppleton. Sharp Retrenchment, Modulated Refinement and Simulation. *Formal Aspects of Computing*, 11:498–540, 1999.

[Bri79] Douglas Bridges. Constructive Functional Analysis. In *Research Notes in Mathematics*, volume 28. Pitman Publishing, London, 1979.

[Bro81] W.S. Brown. A Simple but Realistic Model of Floating Point Computation. *ACM Transactions on Mathematical Software*, 7(4):445–480, 1981.

[Bro87] F.P. Brookes Jr. No silver bullet – Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[BS92] J.P. Bowen and V. Stavridou. Safety–critical systems, formal methods and standards. Technical Report OUCL PRG paper, Oxford University Computing Laboratory, December 1992.

[Bun91] Alan Bundy. A science of reasoning. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 178–198, 1991.

[Bun96] Alan Bundy. Proof planning. In Brian Drabble, editor, *Proceedings of the Third International Conference on Artificial Intelligence Planning Systems, Edinburgh, Scotland, May 29-31, 1996*. AAAI, 1996.

[But98] M. Butler. Towards Tool Support for Formal Refinement. Available at http://www.ecs.soton.ac.uk/publications/rj/1997-1998/DSSE/paper05.pdf, 1998.

[BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice-Hall, New York, 1988.

[CAB$^+$86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W.Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scot F. Smith. *Implementing Mathematics in the NuPRL Proof Development System.* Prentice Hall, 1986.

[CH88] T. Conquant and G. Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CM95] V.A. Carreno and P.S. Miner. Specification of the IEEE-854 Floating point Standard in HOL and PVS, 1995.

[Coo00] D. J. Cooper. *Basic Lisp Techniques.* Franz Inc, 2000.

[COR$^+$95] J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *WIFT '95: Workshop on Industrial-Strength Formal Specification Techniques, Boca raton, Florida, 1995*, 1995.

[CSJ99] A. Cavalanti, A. Sampaio, and J.Woodcock. An inconsistency in procedures, parameters, and substitution in the refinement calculus. *Science of Computer Programming*, 33:87–96, 1999.

[DB82] D.D. Douglas and V.R. Basili. A Comparative Analysis of Functional Correctness. *Computing Surveys*, 14(2):229–244, June 1982.

[Di 01] B.L. Di Vito. A PVS Prover Package for Common Lisp Manipulations, Version 0.9. http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS2-library/pvslib.html, November 2001.

[Dij75] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[DLP79] R.A. De Millo, R.L. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5), March 1979.

[Dol95] Axel Dold. Representing, Verifying and Applying Software Development Steps using the PVS System. In V.S. Alagar and Maurice Nivat, editors, *Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, AMAST'95, Montreal*, volume 936 of *Lecture Notes in Computer Science*, pages 431–435. Springer-Verlag, 1995.

[ESA96] ESA/CNES. Ariane 501 Presentation on Inquiry Board report. Technical report, European Space Agency, 8-10 rue Mario Nikis, 75738 Paris Cedex 15, France, June 1996. Available on the web at http://www.sp.ph.ic.ac.uk/ balogh/ariane5.html.

[Esc00] M. Escardó. Introduction to exact real computation. Notes for a tutorial at ISSAC 2000, August 2000.

[Fel93] Amy Felty. Implementing tactics and tacticals in a higher-order logic programming language. *Journal of Automated Reasoning*, 11(1):41–81, 1993.

[FF98] Dirk Fuchs and Marc Fuchs. Cooperation between top-down and bottom-up theorem provers. In Jacques Calmet and Jan A. Plaza, editors, *AISC, Artificial Intelligence and Symbolic Computation, International Conference AISC'98, Plattsburgh, New York, USA, September 16-18, 1998, Proceedings*, volume 1476 of *Lecture Notes in Computer Science*. Springer, 1998.

[FH94] Amy Felty and Douglas Howe. Tactic theorem proving with refinement-tree proofs and metavariables. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 605–619, Nancy, France, 1994. Springer-Verlag LNAI 596.

[FM87] Amy Felty and Dale Miller. Proof explanation and revision. Technical Report MS-CIS-88-17, LINC LAB 104, Dept. of Computer and Information Science, University of Pennsylvania, March 1987.

[FR86] R. Forsyth and R. Rada. *Machine Learning: applications in expert systems and information retrieval.* Ellis Horwood Series in Artificial Intelligence. Ellis Horwood Limited, Chichester, 1986.

[Fra88] Franz Inc. *Common Lisp, The Reference.* Addison-Wesley Publishing Company, Inc., 1988.

[Fuc95] M. Fuchs. Experiments in the Heuristic Use of Past Proof Experience. Technical Report SEKI-Report SR-95-10, University of Kaiserslautern, November 1995.

[Gal86] Jean H Gallier. *Logic for Computer Science: Foundations of Automatic Theorem Proving.* New York: Wiley, 1986. Also available at http://www.cis.upenn.edu/ jean/gbooks/logic.html.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types.* Cambridge University Press, 1989.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic.* Cambridge University Press, 1993.

[GMNW78] M. Gordon, R. Milner, M. Newey, and C.P. Wadsworth. A metalanguage for interactive proof in lcf. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130. ACM Press, New York, NY, USA, 1978.

[GMW79] M.J. Gordon, R. Milner, and C.P. Wadsworth. Edinburgh LCF: A Mechanised Logic of Computation. *Lecture Notes in Computer Science*, 78, 1979.

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):1–48, March 1991.

[Gor88] M.J.C. Gordon. Mechanizing Programming Logics in HOL. Available on the web at http://www.cam.sri.com/tr/crc034/paper.ps.Z, 1988.

[Gre69] C. Green. An application of Theorem Proving to Problem Solving. In *1st International Joint Conference on Artificial Intelligence*, pages 219–239. Morgan Kaufman, 1969.

[Gri81] David Gries. *The Science of Programming.* SV, 1981.

[Gri99] Ralph P. Grimaldi. *Discrete and combinatorial mathematics.* Addison-Wesley, fourth edition, 1999.

[Gri00] Wolfgang Grieskamp. A Computation Model for Z Based on Concurrent Constraint Resolution. In *ZB2000: Formal Specification and Development in Z and B*, pages 414–432, 2000.

[GS93] David Gries and Fred B. Schneider. *A Logical Approach To Discrete Math.* SV, 1993.

[Gur98] T. Gurukumba. From GDL to CSP: Towards the full formal verification of solid state interlockings. Master's thesis, Oxford University Computing Laboratory, September 1998.

[Ham89] R. Hamlet. *Testing for trustworthiness*, pages 97–104. Ablex Publishing Company, 1989.

[Har96] J.R. Harrison. *Theorem Proving with Real Numbers.* PhD thesis, University of Cambridge, 1996.

[HB95] M. G. Hinchey and J. P. Bowen. *Applications of Formal Methods.* Prentice Hall, 1995.

[HG01] J.G. Hall and T. Gurukumba. Decomposing the *DSub* retrenchment. Technical Report Research Report, Computing Department, The Open University, June 2001.

[HJMT95] Jon G. Hall, Jeremy L. Jacob, John A. McDermid, and Ian Toyn. Towards a Z Method: the two button press case study. *IEE Colloquium on "Practical Application of Formal Methods"*, 1995.

[Hoa69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[Hop93] A.A. Hopgood. *Knowledge-Based Systems for Engineers and Scientists.* CRC Press, 1993.

[HT93] John Harrison and Laurent Thery. Reasoning About the Reals: The Marriage of HOL and Maple. In *Logic Programming and Automated Reasoning*, pages 351–353, 1993.

[IEE87] IEEE. *Standard for binary floating point arithmetic.* The Institute of Electrical and Electronic Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA, ANSI/IEEE Standard 854-1987 edition, 1987.

[JKP02] M. Jamnik, M. Kerber, and M. Pollet. Automatic learning in proof planning. Technical Report CSRP-02-3, University of Birmingham, School of Computer Science, March 2002.

[Jon86] C. B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, Eaglewood Cliffs, NJ., 1986.

[KB01] W. Kramer and A. Bantle. Forward error analysis for floating point algorithms. *Reliable Computing*, 7:321–340, 2001.

[Ker98] M. Kerber. Proof planning: A practical approach to mechanized reasoning in mathematics. In Wolfgang Bibel and Peter H. Schmidt, editors, *Automated Deduction: A Basis for Applications. Volume III, Applications.* Kluwer Academic Publishers, Dordrecht, 1998.

[Kle64] Stephen Cole Kleene. *Introduction to Metamathematics.* North-Holland Publishing Co. —Amsterdam, 1964.

[Kne97] R. Kneuper. Limits of Formal Methods. *Formal Aspects of Computing*, 9:379–394, 1997.

[Kra98] W. Kramer. Constructive error analysis. *Journal of Universal Computer Science*, 4(2):147–163, 1998.

[Kre95] V. Kreinovich. Data Processing Beyond Traditional Statistics: Applications of Interval Computations. A Brief Introduction. In V. Kreinovich, editor, *Supplement to the International Workshop on Applications of Interval Computations*, International Journal of Reliable Computing, pages 13–21, 1995.

[Kre98] C. Kreitz. Program synthesis. In *Automated Deduction—A Basis for Applications*, chapter III.2.5, pages 105–134. Kluwer, 1998.

[KSK93] R. Kumar, K. Schneider, and T. Kropf. Structuring and Automating Hardware Proofs in a Higher-Order Theorem Proving Environment. *Formal Methods in System Design*, 2(2):165–223, 1993.

[KW94] Thomas Kolbe and Christoph Walther. Reusing proofs. In *European Conference on Artificial Intelligence*, pages 80–84, 1994.

[Lei94] Daniel Leivant. Higher order logic. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2: Deduction Methodologies*, pages 229–321. Clarendon Press, Oxford, 1994.

[Lev86] N.G. Leveson. Software Safety: Why, What, How. *Computing Surveys*, 18(2), June 1986.

[LH01] Tatjana Lutovac and James Harland. Proof manipulations for logic programming proof systems. Available at citeseer.nj.nec.com/lutovac01proof.html, 2001.

[LS93] B. Littlewood and L. Strigini. Validation of ultra-high dependebility of software-based systems. *CACM*, 1993.

[Mar84] P. Martin. Intuitionistic Type Theory. Bibliopolis, 1984.

[Mar94] A. Martin. *Machine-Assisted Theorem-Proving for Software Engineering.* PhD thesis, Pembroke College, University of Oxford, 1994.

[MC98] Erica Melis and Jaime G. Carbonell. An argument for derivational analogy. In *Advances in Analogy and Research*, 1998.

[Mey97] Bertrand Meyer. *Oject-Oriented Software Construction.* Prentice Hall PTR, Upper Saddle River, New Jersey, 07458, Second edition, 1997.

[MGW96] A.P. Martin, P.H.B. Gardiner, and J.C.P. Woodcock. A Tactic Calculus. *Formal Aspects of Computing*, 8(E):244–285, 1996.

[Mil72] Robin Milner. Implementation and applications of Scott's logic for Computable Functions. In *Proceedings of the ACM on Proving Assertions about Programs*, pages 1–6. ACM Press, New York, NY, USA, 1972.

[Mil84] R. Milner. The Use of Machines to Assist in Rigorous Proof. *Royal Society of London Philosophical Transactions Series A*, 312:411–421, October 1984.

[Min95] P.S. Miner. Defining the IEEE-854 Floating-Point Standard in PVS. Technical Report Technical Memorandum 110167, NASA Langely Research Center, NASA Langely Research Center Hampton, VA 23681-001, June 1995.

[MM01] C. Munoz and M. Mayero. Real Automation on the Field. Available on the web at http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS2-library/pvslib.html, November 2001.

[Mor88] C.C. Morgan. The Specification Statement. *ACM Transactions on Programming Languages and Systems*, 10(3):403–409, 1988.

[Mor94] C.C. Morgan. *Programming from Specifications.* Series in Computer Science. Prentice-Hall International, second edition edition, 1994.

[Mor97] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9(3):287–306, 1997.

[Mun99] Cesar Munoz. PBS: Support for the B-Method in PVS. Technical report, Computer Science Lab, SRI International, February 1999.

[MW75] Z. Manna and R. Waldinger. Knowledge and Reasoning in Program Synthesis. *Artificial Intelligence*, 6(2):175–208, 1975.

[MW80] Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[MW99] Erica Melis and Jon Whittle. Analogy in inductive theorem proving. *Journal of Automated Reasoning*, 22(2):117–147, 1999.

[NASa] NASA. Formal methods specification and analysis guidebook for the verification of software and computer systems. volume ii: A practioner's companion. http://eis.jpl.nasa.gov/quality/Formal_Methods/.

[NASb] NASA. Formal methods specification and analysis guidebook for the verification of software and computer systems. volume i: Planning and technology insertion. http://eis.jpl.nasa.gov/quality/Formal_Methods/.

[NPS90] B. Nordstrom, K. Petersson, and J. Smith. *Programming in Marting-Lof's Type Theory. An Introduction.* Clarendon Press, 1990.

[ORSvH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 20(2), February 1995.

[OS97] Sam Owre and Natarajan Shankar. The Formal Semantics of PVS. Technical report, SRI International, Computer Science Laboratory, August 1997. URL:http://www.csl.sri.com/sri-csl-fm.html.

[OS03a] S. Owre and N. Shankar. The PVS Prelude Library. Technical Report SRI-CSL-03-01, Computer Science Lab, SRI International, March 2003.

[OS03b] Sam Owre and N. Shankar. Writing PVS proof strategies. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics (STRATA 2003)*, number CP-2003-212448 in NASA Conference Publication, pages 1–15, Hampton, VA, September 2003. NASA Langley Research Center.

[Pau94] L.C. Paulson. Isabelle: A generic Theorem Prover. *Lecture Notes in Computer Science*, (828), 1994.

[Pau99] Lawrence C. Paulson. Logic and proof, 1999. Lecture Notes, Computer Laboratory, University of Cambridge.

[PB02] M. Poppleton and R. Banach. Controlling Control Systems: An Application of Evolving Retrenchment. In *Proc. ZB-02, LNCS*, volume 2272, 2002.

[Pop01] M.R. Poppleton. *Formal Methods for Continuous Systems.* PhD thesis, Department of Computer Science, University of Manchester, 2001.

[Pra95] V. R. Pratt. Anatomy of the Pentium bug. In P. D. Mosses and M. Nielsen and M. I. Schwatzbach, editor, *Proceedings of the 5th International*

Joint Conference on the theory and practice of software development (TAP-SOFT'95), volume 915, pages 97–107. Springer-Verlag, 1995.

[RC90] S. Reeves and M. Clarke. *Logic for Computer Science.* Addison-Wesley, 1990.

[Rob63] J.A. Robinson. Theorem-Proving on the Computer. *Journal of the ACM*, 10(2):163–174, April 1963.

[Rob65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall Series in Computer Science, 1998. FDR is available on the web at http://www.formal.demon.co.uk/FDR2.html.

[Rou99] Y. Rouzard. Interpreting the B-Method in Refinement Calculus. In J. Wing, J. Woodcock, and J. Davies, editor, *Proc. FM'99: World Congress on Formal Methods, Toulouse, France*, volume 1708. Springer-Verlag, September 1999.

[Rus94] J. Rushby. Critical System Properties: Survey and Taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[Rus99] J. Rushby. Integrating Formal Verification: using Model Checking with Automated Abstraction, Invariant Generation and Theorem Proving. In D. Dams, R. Gerth, S. Leue, and M. Massink, editors, *Theoretical and Practical Aspects of SPIN Model Checking: 5th and 6th International SPIN Workshops*, volume 1680, pages 1–11. Springer-Verlag, July 1999. Invited paper presented at 5th SPIN workshop, Trento, Italy, 5th July 1999.

[RvH93] J. Rushby and F. von Henke. Formal Verification of Algorithms for Critical Systems. *IEEE Transactions on Software Engineering*, 19(1):13–23, January 1993.

[San68] N.A. Sanin. Constructive Real Numbers and Constructive Function Spaces. In *Translations of Mathematical Monographs*, volume 21. American Mathemetical Society, Providence, Rhode Island, 1968.

[Sch00] Johann Schumann. Automated theorem proving in high-quality design. In Steffen Hölldobler, editor, *Intellectics and Computational Logic*, volume 19 of *Applied Logic Series.* Kluwer, 2000.

[Sco72] D.S. Scott. *Formal Semantics of Programming Languages*, chapter Lattice theory, data tyoes and semantics, pages 66–106. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Sha92] Natarajan Shankar. Proof search in the intuitionistic sequent calculus. In D. Kapur, editor, *Proceedings 11th Intl. Conf. on Automated Deduction, CADE'92, Saratoga Springs, CA, USA, 15–18 June 1992*, volume 607, pages 522–536. Springer-Verlag, Berlin, 1992.

[Sha01] N. Shankar. Strategies.lisp. Made available via the PVS Users email list, pvs-owner@csl.sri.com, nov 2001.

[Sim96] A. C. Simpson. *Safety through security.* PhD thesis, Oxford University Computing Laboratory, 1996.

[Smi85] D. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27(1):43–96, 1985.

[Smi99] G. Smith. Stepwise development from ideal specifications. Technical Report 99-35, Software Verification Research Centre, School of Information and Technology, University of Queensland, Brisbane, Australia, November 1999.

[SORSC98a] N. Shankar, S. Owre, J.M. Rushby, and M.W.J. Stringer-Calvert. *PVS Language Reference*, 1998. Available at http://www.csl.sri.com/pvs.html.

[SORSC98b] N. Shankar, S. Owre, J.M. Rushby, and M.W.J. Stringer-Calvert. *PVS Prover Guide*, 1998. Available at http://www.csl.sri.com/pvs.html.

[SORSC98c] N. Shankar, S. Owre, J.M. Rushby, and M.W.J. Stringer-Calvert. *PVS System Guide*, 1998. Available at http://www.csl.sri.com/pvs.html.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, 1986.

[Ste84] G. L. Steel. *Common Lisp, The Language.* Digital Press, 1984.

[Ste98] S. Stepney. *High Integrity Compilation, a case study.* Prentice Hall International (UK) Ltd, web edition edition, October 1998.

[Str98] D.W.J. Stringer-Calvert. *Mechanical Verification of Compliler Correctness.* PhD thesis, Department of Computer Science, University of York, 1998.

[Sun96] Sun Microsystems. Numerical Computation Guide. Available on the web at http://docs.sun.com/db/doc/806-7996, December 1996.

[Tar92] M. Tarver. An Algorithm for Inducing Tactics from Proofs. Technical report, Artificial Intelligence Division, University of Leeds, 1992. Available by ftp on agora.scs.leeds.ac.uk/scs/doc.

[Vie93] R. Vienneau. A review of formal methods. Technical report, Kaman Science Corporation, 1993.

[vW94] J. von Wright. The lattice of data refinement. *Acta Informatica*, 31:105–135, 1994.

[Wad93] P. Wadler. A taste of linear logic. *Lecture Notes in Computer Science*, 711, 1993. Invited lecture delivered at Mathematical Foundations of Computer Science, Gdansk, August-September.

[WD96] J.C.P. Woodcock and J. Davies. *Using Z Specification, Refinement and Proof.* Series in Computer Science. Prentice-Hall International, 1996.

[Wic89] B.A. Wichmann. Towards a Formal Specification of Floating Point. *Computer Journal*, 32(5):4321–435, 1989.

[Wil97] Matthew Wilding. Robust Computer System Proofs in PVS. In C. Michael Holloway and Kelly J. Hayhurst, editors, *LFM97: Fourth NASA Langley Formal Methods Workshop.* NASA Conference Publication, 1997.

[Wir71] N. Wirth. Program evelopment by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, April 1971.

[Won02] W. Wong. 8-Bitters Flourish In A 32-Bit World. Electronic Design, June 2002.

[Wor96] J.B. Wordsworth. *Software Engineering with B.* Addison-Wesley, 1996.

[YBH97] W.D. Yu, A. Barshefsky, and S.T. Huang. An Empirical Study of Software Faults Preventable at a Personal Level in a Very Large Software Development Environment. *Bell Labs Technical Journal*, Summer 1997:221–232, 1997.

Appendix A

The PVS system

$$\frac{}{\Sigma, a \vdash_{\Gamma} a, \Lambda}\,(Ax)\ \texttt{auto} \qquad \frac{}{\Sigma, \bot \vdash_{\Gamma} \Lambda}\,(\bot\vdash)\ \texttt{auto} \qquad \frac{}{\Sigma \vdash_{\Gamma} \top, \Lambda}\,(\vdash\top)\ \texttt{auto}$$

$$\frac{\Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\, W\vdash\ \texttt{(delete)}\ \texttt{(hide)} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, A}\,\vdash W\ \texttt{(delete)}\ \texttt{(hide)}$$

$$\frac{A, A, \Gamma \vdash \Delta}{A, \Gamma \vdash \Delta}\,(C\vdash)\ \texttt{(copy)}\ \texttt{(reveal)} \qquad \frac{\Gamma \vdash \Delta, A, A}{\Gamma \vdash \Delta, A}\,\vdash C\ \texttt{(copy)}\ \texttt{(reveal)}$$

$$\frac{\Gamma_1, B, A, \Gamma \vdash \Delta}{\Gamma_1, A, B, \Gamma_2 \vdash \Delta}\,(E\vdash) \qquad \frac{\Gamma \vdash \Delta_1, B, A, \Delta_2}{\Gamma \vdash \Delta_1, A, B, \Delta_2}\,\vdash E$$

$$\frac{\Gamma \vdash \Delta, A}{\neg A, \Gamma \vdash \Delta}\,(\neg\vdash)\ \texttt{auto} \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, \neg A}\,(\vdash\neg)\ \texttt{auto}$$

$$\frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \vee B, \Gamma \vdash \Delta}\,(\vee\vdash)\ \texttt{(split)} \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,(\vdash\vee)\ \texttt{(flatten)}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, (A \wedge B) \vdash \Delta}\,(\wedge\vdash)\ \texttt{(flatten)} \qquad \frac{\Gamma \vdash A, \Delta \quad \Gamma \vdash B, \Delta}{\Gamma \vdash (A \wedge B), \Delta}\,(\vdash\wedge)\ \texttt{(split)}$$

$$\frac{B, \Gamma \vdash \Delta \qquad \Gamma \vdash A, \Delta}{\Gamma, A \Rightarrow B \vdash \Delta}\,(\Rightarrow\vdash)\ \texttt{(split)} \qquad \frac{\Gamma, A \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}\,(\vdash\Rightarrow)\ \texttt{(flatten)}$$

$$\frac{\Sigma, a, b \vdash_{\Gamma, a} \Lambda \quad \Sigma, c \vdash_{\Gamma, \neg a} a, \Delta}{\Sigma, \mathtt{IF}(a, b, c) \vdash_{\Gamma} \Lambda}\,(\Leftrightarrow\vdash)\ \texttt{(smash)}\ EGV2 \qquad \frac{\Sigma, a \vdash_{\Gamma, a} b, \Lambda \quad \Sigma \vdash_{\Gamma, \neg a} a, c, \Lambda}{\Sigma \vdash_{\Gamma} \mathtt{IF}(a, b, c), \Lambda}\,(\vdash\Leftrightarrow)\ \texttt{(smash)}\ \texttt{(bash)}$$

$$\frac{\Gamma, A[t/x] \vdash \Delta}{\Gamma, \forall x : A \vdash \Delta}\,(\forall\vdash)\ \texttt{(inst?)} \qquad \frac{\Gamma \vdash A[y/x], \Delta}{\Gamma \vdash \forall x : A, \Delta}\,(\vdash\forall)\ \texttt{(skolem!)}\ EVC1$$

$$\frac{\Gamma, A[y/x] \vdash \Delta}{\Gamma, \exists x : A \vdash \Delta}\,(\exists\vdash)\ \texttt{(skolem)}\ EVC1 \qquad \frac{\Gamma \vdash A[t/x], \Delta}{\Gamma \vdash \exists x : A, \Delta}\,(\vdash\exists)\ \texttt{(inst?)}$$

$$(\forall(x : T) : a) = (\lambda(x : T) : a) \qquad (\exists(x : T) : a) = \neg(\forall(x : T) : \neg a)$$

$$\frac{A, \Gamma \vdash \Delta \quad \Gamma \vdash \Delta, A}{\Gamma \vdash \Delta}\,(Cut)\ \texttt{(case)} \qquad \frac{\vdash P(n_0) \quad (\forall n \mid n_0 \leq n : (\forall i \mid n_0 \leq i \leq n : P(i)) \vdash P(n+1))}{(\vdash \forall n \mid n_0 \leq n : P(n))}\,\texttt{(induct "n")}$$

Rules labeled (auto) are automatically invoked by PVS. The Exchange rule is entirely omitted in PVS [SORSC98b]. *EVC*1 is the eigenvariable condition that x is not free in the assumptions nor in p. *EVC*2 is the eigenvariable condition that Δ is Λ with all instances of a eliminated. Mathematical induction is over the set $\{n_0, n_0 + 1, n_0 + 2, \ldots\}$ of integers [GS93].

A.1 The Permutation cases for LK

Original Prooftree	Permuted Prooftree
$\dfrac{\dfrac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, \exists x : A}\vdash \exists}{\Gamma \vdash \Delta, \exists x : A, B}\vdash W$	$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, A[t/x], B}\vdash W}{\Gamma \vdash \Delta, B, A[t/x]}\vdash E}{\Gamma \vdash \Delta, B, \exists x : A}\vdash \exists}{\Gamma \vdash \Delta, \exists x : A, B}\vdash E$
$\dfrac{\dfrac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, \exists x : A}\vdash \exists}{B, \Gamma \vdash \Delta, \exists x : A}W \vdash$	$\dfrac{\dfrac{\Gamma \vdash \Delta, A[t/x]}{B, \Gamma \vdash \Delta, A[t/x]}\vdash W}{B, \Gamma \vdash \Delta, \exists x : A}\vdash \exists$
$\dfrac{\dfrac{\Gamma \vdash \Delta, B, A[t/x]}{\Gamma \vdash \Delta, B, \exists x : A}\vdash \exists}{\Gamma \vdash \Delta, \exists x : A, B}\vdash E$	$\dfrac{\dfrac{\Gamma \vdash \Delta, B, A[t/x]}{\Gamma \vdash \Delta, A[t/x], B}\vdash E}{\Gamma \vdash \Delta, B, \exists x : A}\vdash \exists$
$\dfrac{\dfrac{\Gamma, B \vdash \Delta, A[t/x]}{\Gamma, B \vdash \Delta, \exists x : A}\vdash \exists}{B, \Gamma \vdash \Delta, \exists x : A}E \vdash$	$\dfrac{\dfrac{\Gamma, B \vdash \Delta, A[t/x]}{B, \Gamma \vdash \Delta, A[t/x]}E \vdash}{B, \Gamma \vdash \Delta, \exists x : A}\vdash \exists$
$\dfrac{\dfrac{\Gamma \vdash \Delta, A[t/x], B, B}{\Gamma \vdash \Delta, \exists x : A, B, B}\vdash \exists}{\Gamma \vdash \Delta, \exists x : A, B}\vdash C$	$\dfrac{\dfrac{\Gamma \vdash \Delta, A[t/x], B, B}{\Gamma \vdash \Delta, A[t/x], B}\vdash C}{\Gamma \vdash \Delta, \exists x : A, B}\vdash \exists$
$\dfrac{\dfrac{B, B, \Gamma \vdash \Delta, A[t/x]}{B, B, \Gamma, \vdash \Delta, \exists x : A}\vdash \exists}{B, \Gamma \vdash \Delta, \exists x : A}C \vdash$	$\dfrac{\dfrac{B, B, \Gamma \vdash \Delta, A[t/x]}{B, \Gamma \vdash \Delta, A[t/x]}C \vdash}{B, \Gamma \vdash \Delta, \exists x : A}\vdash \exists$

Table A.1: Permutation of Instantiation with Structural Rules

Original Prooftree	Permuted Prooftree
$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B}{\Gamma\vdash\Delta,\exists x:A,B}\vdash\exists \qquad \Gamma\vdash\Delta,\exists x:A(x),C}{\Gamma\vdash\Delta,\exists x:A(x),B\wedge C}\vdash\wedge$ Where the same instantiation can be performed in the right branch or a different instantiation can be used. Using the same instantiation makes the proof uniform. Therefore performing branching rules as early as possible can result in common inference-rules among the prooftree branches.	$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma\vdash A[t/x],\Delta,B}{\Gamma\vdash A[t/x],\Delta,B,\exists x:A}\vdash W}{\Gamma\vdash A[t/x],\Delta,\exists x:A,B}\vdash E \qquad \dfrac{\dfrac{\Gamma\vdash\Delta,\exists x:A,C}{\Gamma\vdash A[t/x],\Delta,C,\exists x:A}\vdash W}{\Gamma\vdash A[t/x],\Delta,\exists x:A,C}\vdash E}{\Gamma\vdash A[t/x],\Delta,\exists x:A,B\wedge C}\vdash\wedge}{\Gamma\vdash\Delta,\exists x:A,B\wedge C,A[t/x]}\vdash E}{\Gamma\vdash\Delta,\exists x:A,B\wedge C,\exists x:A}\vdash\exists}{\Gamma\vdash\Delta,\exists x:A,B\wedge C}(\vdash E;\vdash C;\vdash E)$ Where, according to Gentzen's original formulation, the Exchange-rule is used to bring formula in scope for manipulation by the inference-rules; Contraction is advocated before eliminating a quantifier in order to retain the quantified formula in the premise as in Gentzen System G and this makes Gentzen System LK complete; and weakening produces the same leaves as in the original prooftree. This use of structural rules to obtain the same leaves as in the Original Prooftree increases the complexity of the Permuted Prooftree.
$\dfrac{\dfrac{B,C,\Gamma\vdash\Delta,A[t/x]}{B,C,\Gamma\vdash\Delta,\exists x:A,}\vdash\exists}{B\wedge C,\Gamma\vdash\Delta,\exists x:A}\wedge\vdash$	$\dfrac{\dfrac{B,C,\Gamma\vdash\Delta,A[t/x]}{B\wedge C,\Gamma\vdash\Delta,A[t/x]}\wedge\vdash}{B\wedge C,\Gamma\vdash\Delta,\exists x:A}\vdash\exists$
$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B,C}{\Gamma\vdash\Delta,\exists x:A,B,C}\vdash\exists}{\Gamma\vdash\Delta,\exists x:A,B\vee C}\vdash\vee$	$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B,C}{\Gamma\vdash\Delta,A[t/x],B\vee C}\vdash\vee}{\Gamma\vdash\Delta,\exists x:A,B\vee C}\vdash\exists$
$\dfrac{\dfrac{B,\Gamma\vdash\Delta,A[t/x]}{B,\Gamma\vdash\Delta,\exists x:A}\vdash\exists \qquad \Gamma\vdash\Delta,\exists x:A(x)}{B\vee C,\Gamma\vdash\Delta,\exists x:A,}\vee\vdash$ The $\exists$ in the right-branch may also need to be eliminated for the proof to proceed, and the same instantiation term in the left branch may be used. Hence performing a branching rule before a none-branching one has the effect of duplicating proofsteps among prooftree branches.	$\dfrac{\dfrac{B,\Gamma\vdash\Delta,A[y/x] \qquad C,\Gamma\vdash\Delta,A[t/x]}{B\vee C,\Gamma\vdash\Delta,A[t/x]}\vee\vdash}{B\vee C,\Gamma\vdash\Delta,\exists x:A}\vdash\exists$ Ignoring structural rules gives a Permuted Prooftree of the same complexity as the Original Prooftree. The Permuted Prooftree has the effect of factoring out the common instantiation terms among the prooftree branches (Theorem 3.6.2). However, structural rules can be used to return the same leaves as in the Original Prooftree if desired.
$\dfrac{\dfrac{B,\Gamma\vdash\Delta,A[t/x]}{B,\Gamma\vdash\Delta,\exists x:A}\vdash\exists}{\Gamma\vdash\Delta,\exists x:A,\neg B}\vdash\neg$	$\dfrac{\dfrac{B,\Gamma\vdash\Delta,A[t/x]}{\Gamma\vdash\Delta,A[t/x],\neg B}\vdash\neg}{\Gamma\vdash\Delta,\exists x:A,\neg B}\vdash\exists$

Table A.2: Permutation of Instantiation with Logical Rules

Original Prooftree	Permuted Prooftree
$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B}{\Gamma\vdash\Delta,\exists x:A,B}\vdash\exists}{\neg B,\Gamma\vdash\Delta,\exists x:A}\neg\vdash$	$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B}{\neg B,\Gamma\vdash\Delta,A[t/x]}\neg\vdash}{\neg B,\Gamma\vdash\Delta,\exists x:A}\vdash\exists$
$\dfrac{\dfrac{B,\Gamma\vdash\Delta,A[t/x],C}{B,\Gamma\vdash\Delta,\exists x:A,C}\vdash\exists}{\Gamma\vdash\Delta,\exists x:A,B\Rightarrow C}\vdash\Rightarrow$	$\dfrac{\dfrac{B,\Gamma\vdash\Delta,A[t/x],C}{\Gamma\vdash\Delta,A[t/x],B\Rightarrow C}\vdash\Rightarrow}{\Gamma\vdash\Delta,\exists x:A,B\Rightarrow C}\vdash\exists$
$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B}{\Gamma\vdash\Delta,\exists x:A,B}\vdash\exists \quad \dfrac{C,\Gamma\vdash\Delta,A[t/x]}{C,\Gamma\vdash\Delta,\exists x:A}\vdash\exists}{B\Rightarrow C,\Gamma\vdash\Delta,\exists x:A,}\Rightarrow\vdash$ The same instantiation can be applied in both subtrees thus yielding a uniform proof in terms of the instantiation terms used in the proof.	$\dfrac{\dfrac{\Gamma\vdash\Delta,A[y/x],B \quad C,\Gamma\vdash\Delta,A[t/x]}{B\Rightarrow C,\Gamma\vdash\Delta,A[t/x]}\Rightarrow\vdash}{B\Rightarrow C,\Gamma\vdash\Delta,\exists x:A}\vdash\exists$ Applying the instantiation before the branching $\Rightarrow\vdash$ has the effect of factoring out the common instantiation terms among the proof branches in the Original Prooftree (Theorem 3.6.2).
$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B[z/w]}{\Gamma\vdash\Delta,\exists x:A,B[z/w]}\vdash\exists}{\Gamma\vdash\Delta,\exists x:A,\forall w:B}\vdash\forall$ The skolem variable z can be used as the instantiation term t pending type-correctness conditions (Theorem 3.6.1).	$\dfrac{\dfrac{\Gamma\vdash\Delta,A[t/x],B[z/w]}{A[y/x],\Gamma\vdash\Delta,A[t/x],\forall w:B}\vdash\forall}{\Gamma\vdash\Delta,\exists x:A,\forall w:B}\vdash\exists$ The skolem variable z should not occur in the conclusion and therefore z and t must be distinct.
$\dfrac{\dfrac{B[t/w],A[t/x]\Gamma\vdash\Delta}{B[t/w],\Gamma\vdash\Delta,\exists x:A}\vdash\exists}{\forall w:B,\Gamma\vdash\Delta,\exists x:A}\forall\vdash$ The same instantiation term used in the $\forall\vdash$ rule can be used for the $\vdash\exists$ rule pending type correctness conditions.	$\dfrac{\dfrac{B[t/w]\Gamma\vdash\Delta,A[t/x]}{\forall w:B,\Gamma\vdash\Delta,A[t/x]}\forall\vdash}{\forall w:B,\Gamma\vdash\Delta,\exists x:A}\vdash\exists$ The same instantiation term used in the $\forall\vdash$ rule can be used for the $\vdash\exists$ rule pending type correctness conditions.
$\dfrac{\dfrac{L,\Gamma\vdash\Delta,A[t/x] \quad \Gamma\vdash\Delta,A[t/x],L}{\Gamma\vdash\Delta,A[t/x]}\,Cut}{\Gamma\vdash\Delta,\exists x:A(x)}\vdash\exists$ Where $P_1=\Gamma\vdash\Delta,A[t/x]$. Gentzen Hauptsatz Theorem permutes the Cut above other inference-rules as in the Original Prooftree above, which has the effect of factoring-out common rules among the cut branches in the adjacent Permuted Prooftree (Theorem 3.6.2).	$\dfrac{\dfrac{L,\Gamma\vdash\Delta,A[t/x]}{L,\Gamma\vdash\Delta,\exists x:A(x)}\vdash\exists \quad \dfrac{\Gamma\vdash\Delta,A[t/x],L}{\Gamma\vdash\Delta,\exists x:A(x),L}\vdash\exists}{\Gamma\vdash\Delta,\exists x:A(x)}\,Cut$ Where $P_2=\Gamma\vdash\Delta,\exists x:A(x),L$. The permutation of Cut down the prooftree enables the creative introduction of a new formula (the Cut formula) as early as possible in the proofstate. In addition, it may be possible to eliminate the cut, e.g. the cut is used to introduce the PNF of the goal formula in Section 3.5.1 but goal formula can also be rewritten into PNF by definitional equivalences.

Table A.3: Permutation of Instantiation with Logical Rules

Appendix B

Architectural retrenchment in PVS

B.1 Backtracking on a failed proof attempt

```
SubRefinementPO :

  |-------
{1}   FORALL (S0: [U0 -> U0], S1: [U1 -> [I1 -> U1]], S2: [U2 -> [I2 -> U2]],
              u0: U0, A1, u1: U1, A2, u2: U2, i1: I1, i2: I2, A2: U2):
        ((   inv(S1)(u1) & G(u1, u2) & inv(S2)(u2) & trm(S2)(u2, i2) & W(u0, A1, u1, A2, u2, i1, i2))
          => (trm(S1)(u1, i1) & trm(S2)(u2, i2) & (trm(S1)(u1, i1) => (FORALL (u2p: U2): prd(S2)(u2, i2, u2p)
               => (EXISTS (u1p: U1): prd(S1)(u1, i1, u1p) & G(u1p, u2p))))))

Rule? ((try (grind) (fail) (skip)) (skip) (grind :defs nil :if-match nil))
        .
        .
        .
Trying repeated skolemization, instantiation, and if-lifting,
this simplifies to:
SubRefinementPO :

{-1}  a1(u1!1) = a2(A2!1)
{-2}  a2(A2!1) <= 10
{-3}  (b2(i2!1) <= 10)
{-4}  (a2(u2!1) = a2(A2!1))
{-5}  a0(u0!1) = a2(A2!1)
{-6}  b0(u0!1) = b2(i2!1)
{-7}  a1(A1!1) = a2(A2!1)
{-8}  (b1(i1!1) = b2(i2!1))
{-9}  a2(A2!1) >= b2(i2!1)
{-10} 1 = b2(i2!1)
{-11} 3 = a2(A2!1)
{-12} a2(u2p!1) = a2(A2!1) - b2(i2!1)
  |-------


Attempted proof of SubRefinementPO failed.

Attempted proof of SubRefinementPO failed.
SubRefinementPO :

  |-------
[1]   FORALL (S0: [U0 -> U0], S1: [U1 -> [I1 -> U1]], S2: [U2 -> [I2 -> U2]],
              u0: U0, A1, u1: U1, A2, u2: U2, i1: I1, i2: I2, A2: U2):
        ((   inv(S1)(u1) & G(u1, u2) & inv(S2)(u2) & trm(S2)(u2, i2) & W(u0, A1, u1, A2, u2, i1, i2))
          => (trm(S1)(u1, i1) & trm(S2)(u2, i2) & (trm(S1)(u1, i1) => (FORALL (u2p: U2): prd(S2)(u2, i2, u2p)
               => (EXISTS (u1p: U1): prd(S1)(u1, i1, u1p) & G(u1p, u2p))))))

Trying repeated skolemization, instantiation, and if-lifting,
this yields  2 subgoals:

SubRefinementPO.1 :

{-1}  inv(S1!1)(u1!1)
```

```
{-2}  G(u1!1, u2!1)
{-3}  inv(S2!1)(u2!1)
{-4}  trm(S2!1)(u2!1, i2!1)
{-5}  W(u0!1, A1!1, u1!1, A2!1, u2!1, i1!1, i2!1)
{-6}  prd(S2!1)(u2!1, i2!1, u2p!1)
  |-------
{1}   EXISTS (u1p: U1): prd(S1!1)(u1!1, i1!1, u1p) & G(u1p, u2p!1)

Rule? (inst + "S1Def(u1!1)(i1!1)")
Instantiating the top quantifier in + with the terms: S1Def(u1!1)(i1!1), this yields  2 subgoals:

SubRefinementPO.1.1 :

[-1]  inv(S1!1)(u1!1)
[-2]  G(u1!1, u2!1)
[-3]  inv(S2!1)(u2!1)
[-4]  trm(S2!1)(u2!1, i2!1)
[-5]  W(u0!1, A1!1, u1!1, A2!1, u2!1, i1!1, i2!1)
[-6]  prd(S2!1)(u2!1, i2!1, u2p!1)
  |-------
{1}   prd(S1!1)(u1!1, i1!1, S1Def(u1!1)(i1!1)) & G(S1Def(u1!1)(i1!1), u2p!1)

Rule? (grind)
        ...
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of SubRefinementPO.1.1.

SubRefinementPO.1.2 (TCC):

[-1]  inv(S1!1)(u1!1)
[-2]  G(u1!1, u2!1)
[-3]  inv(S2!1)(u2!1)
[-4]  trm(S2!1)(u2!1, i2!1)
[-5]  W(u0!1, A1!1, u1!1, A2!1, u2!1, i1!1, i2!1)
[-6]  prd(S2!1)(u2!1, i2!1, u2p!1)
  |-------
{1}   real?(b1(i1!1)) & (a1(u1!1) >= b1(i1!1))

Rule? (grind)
        ...
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of SubRefinementPO.1.2.

This completes the proof of SubRefinementPO.1.

SubRefinementPO.2 :

{-1}  inv(S1!1)(u1!1)
{-2}  G(u1!1, u2!1)
{-3}  inv(S2!1)(u2!1)
{-4}  trm(S2!1)(u2!1, i2!1)
{-5}  W(u0!1, A1!1, u1!1, A2!1, u2!1, i1!1, i2!1)
  |-------
{1}   trm(S1!1)(u1!1, i1!1)

Rule? (grind)
        ...
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of SubRefinementPO.2.

Q.E.D.
Run time  = 1.41 secs.
Real time = 559.76 secs.
NIL
PVS(40):
```

```
MACHINE          $DSub$

VARIABLES        $a, b$
INVARIANT        $a \in \mathbb{R}$
                 $\wedge\ b \in \mathbb{R}$

INIT             $X(u)$
OPERATIONS
  $S =$
  BEGIN
    $a \geq b \mid a := a - b$
  END
END
```

```
MACHINE          $MSub$
RETRENCHES       $DSub$
VARIABLES        $aa$
INVARIANT        $aa \in \mathbb{FR}$
RETRIEVES        $(aa < Ov \Rightarrow a = aa)$
                 $\wedge\ (aa = Ov \Rightarrow a \geq Ov)$
INIT             $Y(v)$
OPERATIONS
  $resp \longleftarrow T(bb) =$
  BEGIN
    $bb \in \mathbb{FR} \mid$
      $aa < Ov \wedge aa \geq bb \Rightarrow$
        $aa := aa - bb \parallel resp := ok$
      $[\,]$
      $aa = Ov \vee aa < bb \Rightarrow$
        $aa := aa \wedge resp := fail$
  LVAR $AA$
  WITHIN $a \geq b \wedge AA = aa \wedge (bb < Ov \Rightarrow b = bb)$
  CONCEDES $(resp = ok \Rightarrow a = aa)$
      $\wedge\ (resp = fail \Rightarrow aa = AA)$
  END
END
```

```
MACHINE          $DSub$

VARIABLES        $a, b$
INVARIANT        $a \in \mathbb{R}$
                 $\wedge\ b \in \mathbb{R}$

INIT             $X(u)$
OPERATIONS
  $S =$
  BEGIN
    $a \geq b \mid a := a - b$
  END
END
```

```
MACHINE          $IASub$
RETRENCHES       $DSub$
VARIABLES        $aa$
INVARIANT        $aa \in \mathbb{R}$

RETRIEVES        $a = aa$
INIT             $Y(v)$
OPERATIONS
  $T(bb) =$
  BEGIN
    $bb \in \mathbb{R} \wedge aa \geq bb \mid$
      $aa := aa - bb$
  LVAR $AA$
  WITHIN $b = bb$
  CONCEDES $false$
  END
END
```

```
MACHINE          $DRSub$
RETRENCHES       $IASub$
VARIABLES        $aaa$
INVARIANT        $aaa \in \mathbb{FR}$
RETRIEVES        $(aaa < Ov \Rightarrow aa = aaa)$
                 $\wedge\ (aaa = Ov \Rightarrow aa \geq Ov)$
INIT             $Z(u)$
OPERATIONS
  $U(bbb) =$
  BEGIN
    $bbb \in \mathbb{FR} \mid$
    $aaa < Ov \wedge aaa \geq bbb \Rightarrow$
      $aaa := aaa - bbb$
    $[\,]$
    $aaa = Ov \vee aaa < bbb \Rightarrow$
      $aaa := aaa$
  LVAR $AAA$
  WITHIN $aa \geq bb \wedge AAA = aaa$
      $\wedge\ bbb < Ov \Rightarrow bb = bbb$
  CONCEDES $(aaa < Ov \Rightarrow aa = aaa)$
      $\wedge\ (aaa = Ov \Rightarrow AAA = aaa)$
  END
END
```

```
MACHINE          $OASub$
RETRENCHES       $DRSub$
VARIABLES        $aaaa$
INVARIANT        $aaaa \in \mathbb{FR}$
RETRIEVES        $aaa = aaaa$

INIT             $\alpha(v)$
OPERATIONS
  $resp \longleftarrow V(bbbb) =$
  BEGIN
    $bbbb \in \mathbb{FR} \mid$
    $aaaa < OF \wedge aaaa \geq bbbb \Rightarrow$
      $aaaa := aaaa - bbbb \parallel resp = ok$
    $[\,]$
    $aaaa = Ov \vee aaaa < bbbb \Rightarrow$
      $aaaa := aaaa \parallel resp := fail$
  LVAR $AAAA$
  WITHIN $aaa \geq bbb \wedge AAAA = aaaa$
      $\wedge\ (bbbb < Ov \Rightarrow bbb = bbbb)$
  CONCEDES $(aaa < Ov \Rightarrow aaa = aaaa \wedge resp = ok)$
      $\wedge\ (aaa = Ov \Rightarrow aaaa = AAAA \wedge resp = fail)$
  END
END
```

Figure B.1: Architecture Retrenchment in B

```
DSub: THEORY
BEGIN
  U0:TYPE = [# a0: real, b0:real #]
  S0: VAR [U0 -> U0]

  initDef(u0:U0):U0 = LET u0 = (# a0:=3, b0:=1 #) IN u0

  init(u0p:U0):bool = (a0(u0p)=3 & b0(u0p)=1)

  S0Def(u0:U0|(a0(u0)>=b0(u0))):U0 = LET u0=initDef(u0) IN u0 WITH [a0:=a0(u0)-b0(u0), b0:=b0(u0)]

  inv(S0)(u0:U0):bool = real?(a0(u0)) & real?(b0(u0))

  trm(S0)(u0:U0):bool = (a0(u0)>=b0(u0))

  prd(S0)(u0:U0,u0p:U0):bool =  ((a0(u0)>=b0(u0)) => (a0(u0p) = a0(u0)-b0(u0) & b0(u0p)=b0(u0)))
 END DSub

----------------------------------------------------------------------------------------------------------
No TCCs generated.


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

IASub: THEORY
BEGIN
  IMPORTING DSub
  U1:TYPE = [# a1:real #]
  I1:TYPE = [# b1:real #]
  A:TYPE = [# a1:real #]
  S1: VAR [U1 -> [I1 -> U1]]

  initDef(u0:U0, u1:U1):U1 = LET u1 = (# a1:=3 #) IN u1

  init(u0:U0, u1p:U1): bool = a1(u1p)=a0(u0)

  S1Def(u1:U1)(i1:I1|real?(b1(i1)) & (a1(u1)>=b1(i1))): U1 = LET u1=(# a1:=3 #) IN u1 WITH [a1:=a1(u1)-b1(i1)]

  inv(S1)(u1:U1): bool = real?(a1(u1))

  trm(S1)(u1:U1,i1:I1): bool = (real?(b1(i1)) & a1(u1)>=b1(i1))

  prd(S1)(u1:U1, i1:I1, u1p:U1): bool =
    (real?(b1(i1)) & a1(u1)>=b1(i1)) => a1(u1p)=a1(u1)-b1(i1)

  G(u0:U0, u1:U1): bool = (a1(u1) = a0(u0))

  W(u0:U0, A1:A, u1:U1, i1:I1): bool = init(u0) & init(u0,u1) & (b1(i1) = b0(u0))

  C(up0:U0, up1:U1, A1:A): bool = false
END IASub

----------------------------------------------------------------------------------------------------------
No TCCs generated.
```

Figure B.2: DSub and IASub in PVS

```
DRSub: THEORY
BEGIN
  IMPORTING IASub
  MaxReal:posreal=10
  FinReal:TYPE = {x:real| x<=MaxReal}
  U2:TYPE = [# a2:FinReal #]
  I2:TYPE = [# b2:FinReal #]
  S2:VAR [U2 -> [I2 -> U2]]
  AA:TYPE = [# a2:FinReal #]

  initDef(u0:U0, u2:U2):U2 = LET u2 = (# a2:=3 #) IN u2

  init(u0:U0, u2p:U2): bool = (a2(u2p)=3)

  S2Def(u2:U2)(i2:I2|real?(b2(i2))): U2 = LET u2= (# a2:=3 #) IN
    IF (a2(u2)<10 & a2(u2)>=b2(i2)) THEN u2 WITH [a2:=a2(u2)-b2(i2)] ELSE u2 WITH [a2:=a2(u2)] ENDIF

  inv(S2)(u2:U2): bool = a2(u2) <= 10

  trm(S2)(u2:U2, i2:I2): bool =
     (b2(i2)<=10) & ((a2(u2)<=10 & a2(u2)>=b2(i2)) => true) & ((a2(u2)=10 OR (a2(u2)<b2(i2))) => true)

  prd(S2)(u2:U2, i2:I2, u2p:U2): bool = (b2(i2)<=10) =>
    (((a2(u2)<10 & a2(u2)>=b2(i2)) & a2(u2p)=a2(u2)-b2(i2)) OR
    ((a2(u2)=10 OR  a2(u2)<b2(i2)) & a2(u2p)=a2(u2)))

  G(u1:U1, u2:U2): bool = (a2(u2)<10 => a1(u1)=a2(u2)) & (a2(u2)=10 => a1(u1)>=10)

  W(u0:U0, A1:A, u1:U1, A2:AA, u2:U2, i1:I1, i2:I2): bool=  init(u0,u2) & W(u0,u1,A1,i1) &
    a1(u1)>=b1(i1) & (b2(i2)<=10 => b1(i1)=b2(i2)) & a2(u2)=a2(A2)

  C(u1p:U1, u2p:U2, A2:AA): bool =
   (a2(u2p)<10 => a1(u1p)=a2(u2p)) & (a2(u2p)=10 => a2(u2p)=a2(A2))

END DRSub

-----------------------------------------------------------------------------------------------------------
% Subtype TCC generated (at line 11, column 29) for LET u2 = (# a2 := 3 #) IN u2  %expected type  U2
  % proved - complete
initDef_TCC1: OBLIGATION  FORALL (u21: U2): LET u2: [# a2: real #] = (# a2 := 3 #) IN u2`a2 <= 10;

% Subtype TCC generated (at line 15, column 42) for LET u2 = (# a2 := 3 #) IN
    %   IF (a2(u2) < 10 & a2(u2) >= b2(i2)) THEN u2 WITH [a2 := a2(u2) - b2(i2)]
    %   ELSE u2 WITH [a2 := a2(u2)] ENDIF %expected type  U2
  % unfinished
S2Def_TCC1: OBLIGATION  FORALL (i2: I2 | real?(i2`b2), u21: U2): LET u2: [# a2: real #] = (# a2 := 3 #) IN
      IF (u2`a2 < 10 & u2`a2 >= i2`b2) THEN u2 WITH [a2 := u2`a2 - i2`b2]
      ELSE u2 WITH [a2 := u2`a2] ENDIF `a2 <= 10;

-----------------------------------------------------------------------------------------------------------
Messages for theory DRSub4:

LET/WHERE variable u2 ... is given type  [# a2: real #] from its value expression.

LET/WHERE variable u2 ... is given type  [# a2: real #] from its value expression.
```

Figure B.3: DRSub in PVS

```
OASub: THEORY
BEGIN
  IMPORTING DRSub
  Response:TYPE = {ok,fail}
  U3:TYPE = [# a3:FinReal #]
  I3:TYPE = [# b3:FinReal #]
  O3:TYPE = [# resp:Response #]
  S3:VAR [U3 -> [I3 -> [U3,O3]]]
  AAA:TYPE = [# a3:FinReal #]

  initDef(u0:U0, u3:U3):U3 = (# a3:=3 #)

  init(u0:U0, u3:U3): bool = (a3(u3)=3)

  S3Def(u3:U3)(i3:I3 | real?(b3(i3))): [U3,O3] = LET u3=(# a3:=3 #) IN
    IF (a3(u3)<10 & a3(u3)>=b3(i3)) THEN (u3 WITH [a3:=a3(u3)-b3(i3)], (#resp:=ok#))
    ELSE (u3 WITH [a3:=a3(u3)], (#resp:=fail#)) ENDIF

  inv(S3)(u3:U3): bool = (a3(u3) <= 10)

  trm(S3)(u3:U3, i3:I3): bool =
   b3(i3)<=10 & ((a3(u3)<10 & a3(u3)>=b3(i3)) => TRUE) &
   ((a3(u3)>=10 OR a3(u3)<=b3(i3)) => TRUE)

  prd(S3)(u3:U3, i3:I3, u3p:U3, o3:O3): bool =
   b3(i3)<=10 =>
   (((a3(u3)<10 & a3(u3)>=b3(i3)) & (a3(u3p)=a3(u3)-b3(i3) & resp(o3)=ok))
   OR ((a3(u3)=10 OR a3(u3)<b3(i3)) & (a3(u3p)=a3(u3) & resp(o3)=fail)))

  G(u2:U2, u3:U3): bool =
    (a3(u3)<=10 => a2(u2)=a3(u3))

  W(u0:U0, A1:A, u1:U1, A2:AA, u2:U2, A3:AAA, u3:U3, i1:I1, i2:I2, i3:I3): bool = init(u0,u3) &
    W(u0,A1,u1,A2,u2,i1,i2) & (b3(i3)<=10 => b2(i2)=b3(i3)) & a3(A3)=a3(A3)

  C(u2p:U2, u3p:U3, o3:O3, A3:AAA): bool =
    (resp(o3)=ok => a2(u2p)=a3(u3p)) & (resp(o3)=fail => a3(u3p)=a3(A3))
END OASub

----------------------------------------------------------------------------------------------------------
% Subtype TCC generated (at line 15, column 49) for LET u3 = (# a3 := 3 #) IN
    %  IF (a3(u3) < 10 & a3(u3) >= b3(i3)) THEN (u3 WITH [a3 := a3(u3) - b3(i3)], (# resp := ok #))
    %  ELSE (u3 WITH [a3 := a3(u3)], (# resp := fail #)) ENDIF
    % expected type  [U3, O3]
  % unfinished
S3Def_TCC1: OBLIGATION  FORALL (i3: I3 | real?(i3`b3), u31: U3): LET u3: [# a3: real #] = (# a3 := 3 #) IN
      IF (u3`a3 < 10 & u3`a3 >= i3`b3) THEN (u3 WITH [a3 := u3`a3 - i3`b3], (# resp := ok #))
      ELSE (u3 WITH [a3 := u3`a3], (# resp := fail #)) ENDIF`1`a3 <= 10;

----------------------------------------------------------------------------------------------------------
Messages for theory OASub4:

LET/WHERE variable u3 at line 15, col 53 is given type
  [# a3: real #] from its value expression.
```

Figure B.4: OASub in PVS

```
IAPOs: THEORY
BEGIN
  IMPORTING IASub

  AInitPO: THEOREM
   FORALL (u0:U0, S0:[U0 -> U0]): init(u0) => inv(S0)(u0)

  CInitPO: THEOREM
   FORALL (u0:U0,u1:U1, S1:[U1->[I1->U1]]):  init(u0) & init(u0,u1) => inv(S1)(u1)

  AInvPO: THEOREM
   FORALL (u0:U0, S0:[U0 -> U0]): init(u0) =>
    (inv(S0)(u0) & trm(S0)(u0) => (trm(S0)(u0) &
     (FORALL (u0p:U0): (trm(S0)(u0) & prd(S0)(u0,u0p)) => inv(S0)(u0p))))

  CInvPO: THEOREM
   FORALL (u0:U0, A1,u1:U1, i1:I1, S1:[U1 -> [I1 -> U1]]):
    ((inv(S1)(u1) & trm(S1)(u1,i1)) => (trm(S1)(u1,i1) & (FORALL (u1p:U1):
     (trm(S1)(u1,i1) & prd(S1)(u1,i1,u1p)) => inv(S1)(u1p))))

  RetInitPO: THEOREM
    FORALL (u0:U0, A1,u1:U1, i1:I1, S0:[U0->U0], S1:[U1->[I1->U1]]): FORALL (u1p:U1):
     ( init(u0,u1p) & W(u0,A1,u1,i1)) => (EXISTS (u0p:U0): init(u0p) & G(u0p,u1p))

  SubRefinementPO: THEOREM
    FORALL (u0:U0, u1,A1:U1, i1:I1, S0:[U0 -> U0], S1:[U1 -> [I1 -> U1]]):
    ((inv(S0)(u0) & G(u0,u1) & inv(S1)(u1) &
     trm(S1)(u1,i1) & W(u0,A1,u1,i1))
     => (trm(S0)(u0) & trm(S1)(u1,i1) & (trm(S0)(u0)
         => (FORALL (u1p:U1): (trm(S1)(u1,i1) & prd(S1)(u1,i1,u1p))
             => (EXISTS (u0p:U0): (trm(S0)(u0) & prd(S0)(u0,u0p)) & G(u0p,u1p))))))

  ConcessionPO: THEOREM
     FORALL (u0:U0, u1,A1:U1, i1:I1, S0:[U0 -> U0], S1:[U1 -> [I1 -> U1]]):
     ((inv(S0)(u0) & G(u0,u1) & inv(S1)(u1) &
      trm(S1)(u1,i1) & W(u0,A1,u1,i1))
      => (trm(S0)(u0) & trm(S1)(u1,i1) & (trm(S0)(u0)
          => (FORALL (u1p:U1):(trm(S1)(u1,i1) & prd(S1)(u1,i1,u1p))
             => (EXISTS (u0p:U0):(trm(S0)(u0) & prd(S0)(u0,u0p)) & C(u0p,u1p,A1))))))
END IAPOs

--------------------------------------------------------------------------------------
No TCCs are generated.
```

Figure B.5: Input architecture retrenchment proof obligations in PVS

```
DRPOs: THEORY
BEGIN
  IMPORTING DRSub

  AInitPO: THEOREM
    FORALL (u0:U0,u1:U1, S1:[U1 -> [I1 -> U1]]):
     (init(u0) & init(u0,u1)) => inv(S1)(u1)

  CInitPO: THEOREM
    FORALL (u0:U0,u2:U2, S2:[U2 -> [I2 -> U2]]):
     (init(u0) & init(u0,u2)) => inv(S2)(u2)

  AInvPO: THEOREM
   FORALL (u0:U0,u1:U1, i1:I1, S1:[U1 -> [I1 -> U1]]):
    inv(S1)(u1) & trm(S1)(u1,i1) => (trm(S1)(u1,i1) &
      (FORALL (u1p:U1):(trm(S1)(u1,i1) & prd(S1)(u1,i1,u1p)) => inv(S1)(u1p)))

  CInvPO: THEOREM
   FORALL (u0:U0, u1:U1, u2:U2, i1:I1, i2:I2, A1:U1, A2:fp_num, S2:[U2 -> [I2 -> U2]]):
    ((inv(S2)(u2) & trm(S2)(u2,i2) & W(u0,A1,u1,A2,u2,i1,i2) ) => (trm(S2)(u2,i2)
    & (FORALL (u2p:U2): (trm(S2)(u2,i2) & prd(S2)(u2,i2,u2p)) => inv(S2)(u2p))))

  RetInitPO: THEOREM
    FORALL (u0:U0, A1,u1:U1, A2:fp_num, u2:U2, i1:I1, i2:I2, S1:[U1 -> [I1 -> U1]],
     S2:[U2 -> [I2 -> U2]]): FORALL (u2p:U2): (init(u0,u2p) & W(u0,A1,u1,A2,u2,i1,i2))
       => (EXISTS (u1p:U1): init(u0,u1p) & G(u1p,u2p))

  SubRefinementPO: THEOREM
    FORALL (S0:[U0 -> U0], S1:[U1 -> [I1 -> U1]], S2:[U2 -> [I2 -> U2]],
            u0:U0, A1,u1:U1, A2,u2:U2, i1:I1, i2:I2, A2:fp_num):
    ((inv(S1)(u1) & G(u1,u2) & inv(S2)(u2) & trm(S2)(u2,i2) & W(u0,A1,u1,A2,u2,i1,i2))
   => (trm(S1)(u1,i1) & trm(S2)(u2,i2) & (trm(S1)(u1,i1) => (FORALL (u2p:U2):
       prd(S2)(u2,i2,u2p) => (EXISTS (u1p:U1): prd(S1)(u1,i1,u1p) & G(u1p,u2p))))))

  ConcessionPO: THEOREM
    FORALL (S0:[U0 -> U0], S1:[U1 -> [I1 -> U1]], S2:[U2 -> [I2 -> U2]],
            u0:U0, A1,u1:U1, A2,u2:U2, i1:I1, i2:I2, A2:fp_num):
    ((inv(S1)(u1) & G(u1,u2) & inv(S2)(u2) & trm(S2)(u2,i2) &  W(u0, A1,u1,A2,u2,i1,i2))
   => (trm(S1)(u1,i1) & trm(S2)(u2,i2) & (trm(S1)(u1,i1) => (FORALL (u2p:U2):
       prd(S2)(u2,i2,u2p) => (EXISTS (u1p:U1):prd(S1)(u1,i1,u1p) & C(u1p,u2p,A2))))))
END DRPOs

-------------------------------------------------------------------------------------------

No TCCs are generated.
```

Figure B.6: Data representation retrenchment proof obligations in PVS

```
OAPOs: THEORY
BEGIN
  IMPORTING OASub

  AInitPO: THEOREM
    FORALL (u0:U0,u2:U2, S2:[U2 -> [I2 -> U2]]):
     (init(u0) & init(u0,u2)) => inv(S2)(u2)

  CInitPO: THEOREM
    FORALL (u0:U0,u3:U3, S3:[U3 -> [I3 -> [U3,O3]]]):
     (init(u0) & init(u0,u3)) => inv(S3)(u3)

  AInvPO: THEOREM
  FORALL (u0:U0, A1,u1:U1, A2:fp_num, u2:U2, i1:I1,i2:I2, S2:[U2 -> [I2 -> U2]]):
   ((inv(S2)(u2) & trm(S2)(u2,i2) & W(u0,A1,u1,A2,u2,i1,i2)) => (trm(S2)(u2,i2) &
    (FORALL (u2p:U2): (trm(S2)(u2,i2) & prd(S2)(u2,i2,u2p)) => inv(S2)(u2p))))

  CInvPO: THEOREM
  FORALL (u0:U0, A1,u1:U1, A2:fp_num, u2:U2, A3:fp_num, u3:U3, i1:I1, i2:I2,i3:I3,
          S3:[U3 -> [I3 -> [U3,O3]]]): %(init(u0) & init(u0,u3)) =>
    ((inv(S3)(u3) & trm(S3)(u3,i3) & W(u0,A1,u1,A2,u2,A3,u3,i1,i2,i3)) =>
     (trm(S3)(u3,i3) & (FORALL (u3p:U3, o3:O3):
      (trm(S3)(u3,i3) & prd(S3)(u3,i3,u3p,o3)) => inv(S3)(u3p))))

  RetInitPO: THEOREM
   FORALL (u0:U0, u1:U1, u2:U2, u3:U3, i1:I1, i2:I2, i3:I3, A1:U1, A2:fp_num, A3:fp_num,
          S2:[U2->[I2->U2]], S3:[U3->[I3->[U3,O3]]]):
    FORALL (u3p:U3): (init(u0,u3p) & W(u0,A1,u1,A2,u2,A3,u3,i1,i2,i3)) =>
     (EXISTS (u2p:U2): init(u0,u2p) & G(u2p,u3p))

  SubRefinementPO: THEOREM
   FORALL (u0:U0, u1,A1:U1, u2:U2, A2:fp_num, u3:U3, A3:fp_num, i1:I1, i2:I2, i3:I3,
    S0:[U0->U0], S1:[U1->[I1->U1]], S2:[U2->[I2->U2]], S3:[U3->[I3->[U3,O3]]]):
   ((inv(S2)(u2) & G(u2,u3) & inv(S3)(u3) & trm(S3)(u3,i3) &
    W(u0,A1,u1,A2,u2,A3,u3,i1,i2,i3)) => (trm(S2)(u2,i2) & trm(S3)(u3,i3) &
    (trm(S2)(u2,i2) => (FORALL (u3p:U3,o3:O3):prd(S3)(u3,i3,u3p,o3)
       => (EXISTS (u2p:U2):prd(S2)(u2,i2,u2p) & G(u2p,u3p))))))

  ConcessionPO: THEOREM
   FORALL (u0:U0, u1,A1:U1, u2:U2, A2:fp_num, u3:U3, A3:fp_num, i1:I1, i2:I2, i3:I3,
    S0:[U0->U0], S1:[U1->[I1->U1]], S2:[U2->[I2->U2]], S3:[U3->[I3->[U3,O3]]]):
   ((inv(S2)(u2) & G(u2,u3) & inv(S3)(u3) & trm(S3)(u3,i3) &
     W(u0,A1,u1,A2,u2,A3,u3,i1,i2,i3)) => (trm(S2)(u2,i2) & trm(S3)(u3,i3) &
     (trm(S2)(u2,i2) => (FORALL (u3p:U3,o3:O3):prd(S3)(u3,i3,u3p,o3)
        => (EXISTS (u2p:U2): prd(S2)(u2,i2,u2p) & C(u2p,u3p,o3,A3))))))
END OAPOs

-------------------------------------------------------------------------------------

No TCCs are generated
```

Figure B.7: Output-architecture retrenchment proof obligations in PVS

B.2 Evolving Retrenchment in PVS

```
DAddExact: THEORY
BEGIN
  U0:TYPE = [# a0:real, b0:real #]
  S0:VAR [U0->U0]

  initDef(u0:U0):U0 = (# a0:=1/4, b0:=1/4 #)

  init(u0:U0):bool = (a0(u0)=1/4 & b0(u0)=1/4)

  S0Def(u0:U0):U0 = (# a0:=a0(u0)+b0(u0), b0:=b0(u0) #)

  inv(S0)(u0:U0):bool = real?(a0(u0)) & real?(b0(u0))

  trm(S0)(u0:U0):bool = TRUE

  prd(S0)(u0:U0,u0p:U0):bool = (a0(u0p)=a0(u0)+b0(u0) & b0(u0p)=b0(u0))
END DAddExact


============================================================
IAAddExact: THEORY %Uses the idea of maximally abstract retrenchments
BEGIN
  IMPORTING DAddExact
  U1:TYPE = [# a1:real #]
  I1:TYPE = [# b1:real #]
  S1:VAR [U1 -> [I1 -> U1]]
  u1: U1

  initDef(u0:U0,u1:U1):U1 = (# a1:=a0(u0) #)

  init(u0:U0,u1:U1): bool = (a1(u1)=a0(u0))

  S1Def(u1:U1)(i1:I1): U1 = (# a1:=a1(u1)+b1(i1) #)

  inv(S1)(u1:U1): bool = real?(a1(u1))

  trm(S1)(u1:U1, i1:I1): bool = real?(b1(i1))

  prd(S1)(u1:U1, i1:I1, u1p:U1): bool = real?(b1(i1)) => a1(u1p)=a1(u1)+b1(i1)

  G(u0:U0, u1:U1): bool = (a1(u1) = a0(u0))

  W(u0:U0, A1,u1:U1, i1:I1): bool = b1(i1)=b0(u0) & init(u0) & init(u0,u1)

  C(up0:U0, up1:U1, A1:U1): bool = false
END IAAddExact

-----------------------------------------------------------------------------
No TCCs generated
```

Figure B.8: DAddExact and IAAddExact in PVS

```
DRAddExact: THEORY
BEGIN
  IMPORTING IEEE_854[2, 6, 6, 2, -1], IAAddExact
  U2:TYPE = [# a2:fp_num #]
  I2:TYPE = [# b2:fp_num #]
  S2:VAR [U2 -> [I2 -> U2]]
  u2: U2
  i2: I2

  initDef(u0:U0,u2:U2):U2 = (# a2:=real_to_fp(a0(u0)) #)

  init(u0:U0,u2:U2): bool = (a2(u2) = real_to_fp(a0(u0)))

  S2Def(u2:U2)(i2:I2): U2 = IF (NOT over_under?(value(fp_add(a2(u2), b2(i2), to_nearest)))) THEN
    (# a2:=fp_add(a2(u2), b2(i2), to_nearest) #) ELSE (# a2:=real_to_fp(1/32) #) ENDIF

  inv(S2)(u2:U2): bool = finite?(a2(u2))

  trm(S2)(u2:U2, i2:I2): bool = finite?(b2(i2))

  prd(S2)(u2:U2, i2:I2, u2p:U2): bool = finite?(b2(i2)) =>
    ((NOT over_under?(value(fp_add(a2(u2), b2(i2), to_nearest)))) &
     (u2p = (#a2:=fp_add(a2(u2), b2(i2), to_nearest)#)))
    OR ((over_under?(value(fp_add(a2(u2), b2(i2), to_nearest)))) & (u2p = (#a2:=real_to_fp(1/32)#)))

  G(u1:U1, u2:U2): bool = (NOT over_under?(a1(u1))) => (RelError(value(a2(u2)), a1(u1)) <= 0)

  W(u0:U0, A1:U1, u1:U1, A2:fp_num ,u2:U2, i1:I1, i2:I2) :bool = init(u0,u2) &
       W(u0,A1,u1,i1) & ((NOT over_under?(b1(i1))) => (b2(i2)=real_to_fp(b1(i1))))

  C(u1p:U1, u2p:U2, A2:fp_num): bool = false
END DRAddExact

----------------------------------------------------------------------------------------------

% Assuming TCC generated (at line 3, column 12) for IEEE_854[2, 6, 6, 2, -1]
  %generated from assumption IEEE_854.Exponent_range  %unfinished
IMP_IEEE_854_TCC1: OBLIGATION (2 - -1) / 6 > 5;

% Assuming TCC generated (at line 3, column 12) for IEEE_854[2, 6, 6, 2, -1]
  %generated from assumption IEEE_854.Significand_size % unfinished
IMP_IEEE_854_TCC2: OBLIGATION 2 ^ (6 - 1) >= 10 ^ 5;

% Assuming TCC generated (at line 3, column 12) for IEEE_854[2, 6, 6, 2, -1]
  %generated from assumption IEEE_854.Exponent_Adjustment % unfinished
IMP_IEEE_854_TCC3: OBLIGATION  abs(6 - (3 * (2 - -1) / 4)) <= 6 & integer?(6 / 12);

% Subtype TCC generated (at line 15, column 30) for fp_add(a2(u2), b2(i2), to_nearest):
  %expected type  (finite?[2, 6, 2, -1])  %unfinished
S2Def_TCC1: OBLIGATION
 FORALL (i2: I2, u2: U2): finite?[2, 6, 2, -1](fp_add[2, 6, 6, 2, -1](u2`a2, i2`b2, to_nearest));

% Subtype TCC generated (at line 32, column 49) for  a2(u2): expected type  (finite?[2, 6, 2, -1])
  % unfinished
G_TCC2: OBLIGATION
 FORALL (u1: U1, u2: U2): (NOT over_under?[2, 6, 6, 2, -1](u1`a1)) IMPLIES finite?[2, 6, 2, -1](u2`a2);

% Subtype TCC generated (at line 35, column 22) for  b1(i1): expected type real  %proved - incomplete
W_TCC1: OBLIGATION
 FORALL (A1, i1: I1, u0: U0, u1: U1, u2: U2):  W(u0, A1, u1, i1) AND init(u0, u2) IMPLIES real_pred(i1`b1);
```

Figure B.9: DRAddExact in PVS

```
OAAddExact: THEORY
BEGIN
  IMPORTING IEEE_854[2, 6, 6, 2, -1], DRAddExact %8-bit architecture
  Response:TYPE = {ok, fail}
  U3:TYPE = [# a3:fp_num #]
  I3:TYPE = [# b3:fp_num #]
  O3:TYPE = [# resp:Response #]
  S3:VAR [U3 -> [I3 -> [U3, O3]]]
  u3: U3
  i3: I3

  initDef(u0:U0,u3:U3):U3 = (# a3:=real_to_fp(a0(u0)) #)

  init(u0:U0,u3:U3): bool = (a3(u3) = real_to_fp(a0(u0)))

  S3Def(u3:U3)(i3:I3): [U3, O3] = IF (NOT over_under?(value(fp_add(a3(u3), b3(i3), to_nearest))))
    THEN ((# a3:=fp_add(a3(u3), b3(i3), to_nearest) #), (# resp:=ok #))
    ELSE ((# a3:=real_to_fp(MachEps(2,5)) #), (# resp:=fail #)) ENDIF

  inv(S3)(u3:U3): bool = finite?(a3(u3))

  trm(S3)(u3:U3, i3:I3): bool = finite?(b3(i3))

  prd(S3)(u3:U3, i3:I3, u3p:U3, o3:O3): bool = finite?(b3(i3)) =>
    ((NOT over_under?(value(fp_add(a3(u3), b3(i3), to_nearest)))) &
     (u3p = (#a3:=fp_add(a3(u3), b3(i3), to_nearest)#) & resp(o3)=ok))
    OR ((over_under?(value(fp_add(a3(u3), b3(i3), to_nearest)))) &
        (u3p = (#a3:=real_to_fp(MachEps(2,5))#) & resp(o3)=fail))

  G(u2:U2, u3:U3): bool = (NOT over_under?(value(a2(u2)))) =>
    (RelError(value(a3(u3)),value(a2(u2))) <= 0)

  W(u0:U0, A1,u1:U1, A2:fp_num, u2:U2, A3:fp_num, u3:U3, i1:I1, i2:I2, i3:I3): bool=
   init(u0,u3) & W(u0,A1,u1,A2,u2,i1,i2) & finite?(b2(i2)) &
   ((NOT over_under?(value(b2(i2)))) => (b3(i3)=b2(i2)))

  C(u2p:U2, u3p:U3, o3:O3, A3:fp_num): bool = false
    %RelError(value(a3(u3p)), value(a2(u2p)))<=2*MachEps(2,5)
END OAAddExact

---------------------------------------------------------------------------------

% Assuming TCC generated (at line 3, column 12) for IEEE_854[2, 6, 6, 2, -1]:
  %generated from assumption IEEE_854.Exponent_range  %unfinished
IMP_IEEE_854_TCC1: OBLIGATION (2 - -1) / 6 > 5;

% Assuming TCC generated (at line 3, column 12) for IEEE_854[2, 6, 6, 2, -1]:
  %generated from assumption IEEE_854.Significand_size  %unfinished
IMP_IEEE_854_TCC2: OBLIGATION 2 ^ (6 - 1) >= 10 ^ 5;

% Assuming TCC generated (at line 3, column 12) for IEEE_854[2, 6, 6, 2, -1]:
  %generated from assumption IEEE_854.Exponent_Adjustment  %unfinished
IMP_IEEE_854_TCC3: OBLIGATION  abs(6 - (3 * (2 - -1) / 4)) <= 6 & integer?(6 / 12);

% Subtype TCC generated (at line 17, column 30) for fp_add(a3(u3), b3(i3), to_nearest):
  %expected type  (finite?[2, 6, 2, -1]  % unfinished
S3Def_TCC1: OBLIGATION
 FORALL (i3: I3, u3: U3): finite?[2, 6, 2, -1](fp_add[2, 6, 6, 2, -1](u3`a3, i3`b3, to_nearest));

% Subtype TCC generated (at line 34, column 27) for  a2(u2): expected type (finite?[2, 6, 2, -1])
  % unfinished
G_TCC1: OBLIGATION FORALL (u2: U2): finite?[2, 6, 2, -1](u2`a2);

% Subtype TCC generated (at line 35, column 20) for  a3(u3): expected type (finite?[2, 6, 2, -1])
  % unfinished
G_TCC2: OBLIGATION  FORALL (u2: U2, u3: U3): (NOT over_under?[2, 6, 6, 2, -1](value[2, 6, 2, -1](u2`a2)))
                                            IMPLIES finite?[2, 6, 2, -1](u3`a3);
```

Figure B.10: OAAddExact in PVS