# ENHANCING THE NUPRL PROOF DEVELOPMENT SYSTEM AND APPLYING IT TO COMPUTATIONAL ABSTRACT ALGEBRA

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Paul Bernard Jackson

January 1995

ENHANCING THE NUPRL PROOF DEVELOPMENT SYSTEM AND APPLYING IT TO COMPUTATIONAL ABSTRACT ALGEBRA

Paul Bernard Jackson, Ph.D.

Cornell University 1995

This thesis describes substantial enhancements that were made to the software tools in the Nuprl system that are used to interactively guide the production of formal proofs. Over 20,000 lines of code were written for these tools. Also, a corpus of formal mathematics was created that consists of roughly 500 definitions and 1300 theorems. Much of this material is of a foundational nature and supports all current work in Nuprl. This thesis concentrates on describing the half of this corpus that is concerned with abstract algebra and that covers topics central to the mathematics of the computations carried out by computer algebra systems.

The new proof tools include those that solve linear arithmetic problems, those that apply the properties of order relations, those that carry out inductive proof to support recursive definitions, and those that do sophisticated rewriting. The rewrite tools allow rewriting with relations of differing strengths and take care of selecting and applying appropriate congruence lemmas automatically. The rewrite relations can be order relations as well as equivalence relations. If they are order relations, appropriate monotonicity lemmas are selected.

These proof tools were heavily used throughout the work on computational algebra. Many examples are given that illustrate their operation and demonstrate their effectiveness.

The foundation for algebra introduced classes of monoids, groups, rings and modules, and included theories of order relations and permutations. Work on finite sets and multisets illustrates how a quotienting operation hides details of datatypes when reasoning about functional programs. Theories of summation operators were developed that drew indices from integer ranges, lists and multisets, and that summed over all the classes mentioned above. Elementary factorization theory was developed that characterized when cancellation monoids are factorial. An abstract data type for the operations of multivariate polynomial arithmetic was

defined, and the correctness of an implementation of these operations was verified. The implementation is similar to those found in current computer algebra systems.

This work was all done in Nuprl's constructive type theory. The thesis discusses the appropriateness of this foundation, and the extent to which the work relied on it.

# Biographical Sketch

Paul Jackson was born in Berkeley, California on 20th July 1962. He grew up in London, England and from 1973 to 1980 attended William Ellis Grammar School. In his last years there, he became very interested in computers; he implemented his own homebrew system, created hardware and software for a run-length-encoded graphics display, and had the honour of introducing Her Majesty Queen Elizabeth II to John Conway's 'Game of Life'.

From 1981 to 1984 he was an undergraduate in Engineering at Churchill College, Cambridge. He graduated in 1984 with a starred First in Electrical Sciences.

From 1984 to 1986 he designed application-specific integrated circuits with GE in the Research Triangle Park, North Carolina.

He was a graduate student in Physics at Cornell from 1986 to 1988, and in 1988 received an MS in Physics. From 1988 to 1994 he was a graduate student in Computer Science at Cornell.

To Larry and Letizia

# Acknowledgements

I would like to thank the chair of my special committee, Bob Constable, for providing an exciting and stimulating research environment for the past five years. During this time, he provided much support and encouragement. He opened up for me a whole new world of logic and applied logic. I wish to thank Keshav Pingali for agreeing to serve on my committee. I wish to thank Jim Sethna, my minor committee member from Physics, for being supportive of my switch from Physics to Computer Science, and enthusiastic about my venture into formal mathematics.

I am grateful to the National Science Foundation for a Graduate Fellowship for the two years I was in Physics and for my first year in Computer Science. I am also grateful to the National Aeronautics and Space Administration for Fellowship NGT-50786 in their Graduate Student Researcher's Program. I particularly enjoyed and found stimulating the annual meetings that NASA held in Washington for all fellowship holders. During my years as a graduate student, my work was also supported by NSF grant CCR-9244739 and ONR grant N00014-92-J-1764.

I wish to thank the organisers and sponsors of several workshops and summer schools that I have had the pleasure to be invited to. These included:

- Workshops on Formal Methods in Software Engineering, held in 1991 and 1993 in Fort Monroe, Virginia, and Philadelphia, Pennsylvania, and sponsored by the ARO and ONR.

- An International Summer School on Logic and Algebra of Specification, held in 1991 in Marktoberdorf, Germany, sponsored by the NATO Science Committee under their Advanced Study Institutes Programme and by the NSF.

- The QED Workshop, held in 1994 at Argonne, Illinois. I want especially to thank the organizers, Bob Boyer and Rusty Lusk for inviting me and Ralph Wachter at the ONR for funding this workshop. Few definite conclusions were reached, but I found the workshop tremendously stimulating and it provided an excellent opportunity for meeting other researchers in the field.

- The 2nd International Summer School in Logic for Computer Science, held in 1994 in Chambery, France and on the topic of Automated Deduction. This was organized by Michel Parigot at the Université Paris 7, and was part of the Euroconferences programme of the European Community.

I want especially to thank Douglas Howe for the many discussions we have had over the years. His earlier work on the Nuprl system and later work on new theories for Nuprl have both been a constant source of inspiration.

I want to thank David Basin for introducing me to the Nuprl project while we rode the ski-lifts at Greek Peak, and for guiding me in my first proofs in Nuprl. I wish to thank Stuart Allen for the elegant design of the Nuprl V4.1 editor and our programmer Richard Eaton for all the work he has put into the Lisp foundations of Nuprl V4.1.

Chet Murthy provided lively and provocative conversations on topics in and outside of Computer Science. Todd Wilson diligently read rough drafts of much of this thesis, and gave me much helpful feedback. It has also been a pleasure to share an office with him during the last months of writing this thesis. Karl Crary provided many useful comments on a final draft.

I would like to thank Miriam Leeser for first suggesting that I look at the question of verifying floating-point hardware. Though this did not become the main topic of this thesis, I feel that all the work here provides a broad and solid foundation for pursuing this and other hardware verification tasks in the future.

I want to thank Dexter Kozen and Rich Zippel for stimulating my interest in mathematics and algebra.

I wish to thank Prakash Panangaden for his advice on career matters.

I want to thank John Beck for introducing me to category theory, and Anil Nerode and Richard Platek for their stimulating lectures in logic and applied logic.

I would like to thank several people for the stimulation and inspiration that they have provided through their work and lectures and through the talks that I have had with them. They include Bob Boyer, J. Moore, N. Shankar, Andrzej Trybulec, Piotr Rudnicki, Mike Gordon, Tom Melham and John Harrison.

I want to mention John Severn, who was a physics teacher at my grammar school and who ran the school electronics club. I had many enjoyable times with this club, and it was through it that I first became interested in computers. Also, John Severn, together with another physics teacher, Mike Detheridge, instilled in me a love for and fascination with physics.

I would like to acknowledge all my friends, with whom I have had many happy times. Those from Ithaca include Michael Slifker, Michelle Friedman, Beth-Ellen Rosenbaum, Al Shapere, Bruce Hendrickson, Steven Rapkin, James Allan, George Musser, Talia Shaffer and Paul Jeffries. Aswin van den Berg introduced me to the delights of swing dancing. I had many enjoyable times playing music with Ed Lyman, he on piano and harpsichord, myself on flute. It also has been a pleasure keeping up with friends from Cambridge and London.

Finally, I wish to acknowledge Elizabeth Ezra, whose love, companionship and empathy have all been simply wonderful.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

## 1.1 Motivation

This thesis describes foundational work in the design of computer systems to assist engineers, mathematicians and scientists in the production and checking of completely-rigorous mathematical arguments.

There is a long history of the successful use of computers to automate mathematical calculations involving numbers and arithmetic. For evidence of this, one only need look at the wealth of current work in scientific computation and numerical analysis.

More recently, there has been very rapid growth in the popularity of computer algebra systems such as Mathematica [Wol91] and Maple [CGG+91]. At their core, these systems have routines for carrying out such symbolic manipulations as factorizing polynomials, for computing integrals and differentials, and for finding symbolic solutions to sets of equations. They also provide languages for rapidly constructing packages that extend the systems' capabilities.

A major problem with many computer algebra systems is that they have few design features to ensure that the symbolic manipulations that they carry out respect the mathematical meaning of the symbols being manipulated. For example, Mathematica has no type system and does not by default do capture-avoiding substitution. The algorithms used are often complicated and it is all too easy for programmers to make mistakes. Further, computer-algebra-system programmers often find it convenient to ignore special cases so that the procedures they write are not even logically sound. A trivial example of this is the common practice of simplifying $1/(1/x)$ to $x$, where $x$ is might be understood to range over real or complex numbers but there is no check on $x$ being non-zero. Such decisions are often made because there is no way of uniformly carrying out such checks, and even if checks were attempted, they might considerably slow down symbolic calculation. Further, users of computer algebra systems might well be unaware of the corners that were cut by the programmers of the packages that they use.

Computer algebra systems such as Maple [CGG+91] and Axiom [JS92] are addressing the issue of the meaning of symbolic expressions by adopting sophisticated type systems. Some too make an attempt to track side conditions of calculations. However, all computer algebra systems lack the absolute notion of rigor found in mathematics.

Rigor in mathematics is established by having precise languages for stating logical propositions and precise rules defining what is a valid inference and what is a valid proof. Unfortunately, computer algebra systems have only vague notions of what a proposition is, and no notions of what a proof is, or what a formally correct inference procedure is.

An until-now fairly separate field of research has been the construction of mechanical *theorem provers*. I use this term here in a loose sense, encompassing not only resolution theorem provers and systems such as NQTHM [BM88a], but also systems perhaps better called *proof checkers* such as Automath [dB80] and Mizar [Rud92], and *proof development systems* such as Nuprl and HOL [GM93]. The HOL group refers to their system as a *theorem proving environment*, which I think is an apt phrase for describing all these kinds of system.

Such systems all support notions of definition, theorem and proof analogous to those that a mathematician employs. They have precise semantics and are carefully engineered to ensure that inferences always comply with this semantics. One technique used to ensure correctness is to insist that all inference procedures eventually justify themselves in terms of a small fixed number of primitive rules. Another is to have scrupulous code walk-throughs and testing regimens. Another, still in its early stages, is to use theorem provers to formally verify core parts of their own designs. This last technique is often called *reflection*.

Because of the foundational nature of theorem provers, their domains of reasoning have usually been both far more restricted and of a different nature than those of computer algebra systems; they have mostly been successfully applied in areas of concrete, discrete mathematics. Currently, the principle motivating application of theorem provers is increasing the reliability of digital hardware and software, particularly when errors can be life-threatening or very costly. The inherent rigor of theorem provers makes them ideal for checking masses of tedious details and identifying subtle errors.

However, it is becoming increasingly desirable to obtain the assurances of safety, reliability and correctness that theorem provers can provide in domains much more traditionally associated with computer algebra systems. Engineers want to check the designs of digital signal processing (DSP) systems and hybrid control systems where there is an intimate mixture of algebraic operations on continuous, quasi-continuous (e.g. floating-point numbers) and discrete quantities.

## 1.2 Aims

One of the major aims of the work described in this thesis has been to broaden the range of domains and the kinds of operations that the Nuprl proof development system can be used to reason about. The approach has been to try to do this in an abstract unified way, rather than repetitively developing many disparate concrete theories.

I was also keen to provide software tools that could simplify and speed interactive proof development by automating away tedious detail and offer very general and controllable modes of reasoning.

Another major aim has been to explore techniques for formal specification and implementation of abstract data types. Abstract data types (and the related notion of classes in object-oriented programming) provide fundamental mechanisms for the control of complexity in large software systems by encouraging modularization and code reuse.

I chose to look at in particular the specification and implementation of a set of operations for multivariate polynomial arithmetic. This topic was chosen because of the rich algebraic structures involved, and it was clear that it would provide an excellent opportunity for demonstrating the software tools and formal mathematical theories I had developed earlier. This topic is also a very relevant to the program of formally verify core parts of theorem-proving and computer algebra systems; the implementation I chose to verify is similar to that which is commonly used in current computer algebra systems.

## 1.3 Overview of Nuprl System

I took as my starting-point the Nuprl proof-development system [C$^+$86]. The core of Nuprl is a program called the *refiner* which has knowledge of a set of primitive inference rules and which is ultimately responsible for constructing every proof. Therefore, the correctness of proofs depends solely on the refiner.

The inference rules provide very small steps in proofs. Larger steps are made by invoking programs called *tactics* which choose and sequence the primitive rules. Some tactic invocations result in only one or two primitive rules being applied, others can result in $10^4$ or more!

A key feature of tactics is that they are modular; their behaviours are easy to predict and it is straightforward to combine existing tactics into new larger tactics. Indeed the user is encouraged to do this as he or she becomes more competent with the system. Nuprl's tactic collection form a tool-kit of mathematical techniques for carrying out proof. The tactic software-engineering paradigm was pioneered in the LCF project and since has been adopted in several other theorem provers in use today such as HOL and Isabelle.

When using Nuprl, each proof is maintained as tree shaped data-structure. Nodes of the tree indicate states of the proof, and tactic text annotates each node, showing how immediate children nodes were generated. Tactics therefore document proofs and make proofs readable. Several examples of such proof trees are given in this thesis. By way of contrast, other tactic-based provers by default don't maintain proof trees; during proof they only maintain the unproven leaves of a proof and after a proof is completed they only retain the script of tactic text. Such scripts are often cryptic and the explanatory potential of tactics is lost.

Nuprl's primary application was intended to be the verification and synthesis of correct programs. Previous experience at Cornell with the PL-CV project had suggested that even when just reasoning about programs, it would be very advantageous to choose a logic that could be used as a foundation for all of mathematics.

The most well-known foundational theories for mathematics are set theories; in particular Zermelo-Fraenkel set theory. However, it can be rather difficult and clumsy to reason about computations in set theory. Nuprl adopted a type-theoretic language close in spirit to that of DeBruijn's Automath system [dB80] and strongly influenced by work of Scott [Sco70] and Martin-Löf [ML82]. A major difference between type theory and set theory is that in type theory, the notion of *function* is considered to be primitive and that type theories provide as primitive, at the very least, ways of constructing primitive recursive functions. It seems plausible therefore, that type theories should be better for reasoning about computation. Martin-Löf's type theory was particularly interesting because it was proposed as being an alternative foundation for mathematics, more specifically for *constructive* mathematics.

In constructive mathematics, one distinguishes between different kinds of proof techniques that are normally considered to be equivalent. For example, in order to prove a proposition $\exists x.\, P_x$, it is usually sufficient to prove the impossibility of the proposition $\forall x.\, \neg P_x$. A constructivist would deny this. She would say that a valid proof of $\exists x.\, P_x$ must come up with a specific $a$ such that $P_a$ is true. Further, a constructivist would only accept a proof of a proposition $\forall x\, \exists y.\, P_{xy}$ if a uniform method could be exhibited for constructing a $y$ such that $P_{xy}$, given some arbitrary $x$. A *uniform method* is frequently taken to mean some recursive function or computer program.

Interest in constructive mathematics has revived recently for a couple of reasons. Firstly, constructive mathematics provides a way of viewing the language of logical propositions as a *specification* language for programs. An ongoing thrust of work in computer science has been to develop program specification languages and formalisms for systematically deriving programs from specifications. For constructive mathematics to provide such a methodology, techniques are needed for systematically extracting programs from constructive proofs. Early work in this field includes that of Bishop [Bis70] and Constable [Con71]. What distinguished Martin-Löf's '82 type theory was that the method it suggested for program synthesis was exceptionally simple: a direct correspondence was set up between the constructs of mathematical logic, and the constructs of a functional programming language. Specifically, every proposition was considered to be isomorphic to a type expression, and the proof of a proposition would suggest precisely how to construct an inhabitant of the type, which would be a term in a functional programming language. The term that inhabits the type corresponding to a proposition is often referred to as the *computational content* of the proposition.

Secondly, designers of computer algebra systems and researchers in fields such as computational geometry really care about whether constructions described in algebra text books are effective or not. There has been a revival of much algebra done in the last century when more attention was paid to the constructive nature of mathematics.

From the start, I was concerned about the strengths and weaknesses of working in Nuprl's constructive type theory. I have tried to address at various points in this thesis where the type theory was a help and where a hindrance. Importantly, I should emphasize to the reader that though all the work described in this thesis was does within the constructive type theory of Nuprl, most of the issues discussed are relevant to any design of any theorem-proving environment with algebraic capabilities, no matter what the foundational logic.

## 1.4 Contributions

### 1.4.1 Proof Development Tools

When I joined the Nuprl group, a significant amount of work had been done in V3 of the Nuprl system and the groundwork was being done for V4 (see Section 1.5.1 for more information). I started out by rewriting many of the V3 tactics for the then fledgling V4 system, and over the course of my PhD, I made many significant changes and extensions.

#### 1.4.1.1 Rewrite Package

The most significant extension I made to the tactics was with a package to support rewriting. Usual treatments of rewriting [DJ90] assume that the equivalence relations being rewritten with respect to are global congruence relations, so that the substitutions suggested by rewrite rules are always valid. However, there are common instances where one wants to rewrite using relations that are not always respected.

The rewrite package I designed verifies every application of a rewrite rule, checking automatically that all relevant congruence properties are obeyed. The package handles also rewrite rules where the rewrite relation is an order relation. In these cases it checks the appropriate monotonicity properties. Examples are given throughout the thesis of both monotone and congruence reasoning. The ability to rewrite with order relations has also been exploited in recent work in Nuprl on real analysis and I expect it could be very useful when developing theories of program refinement calculi and process algebras.

The package was designed around the notion of *conversion* [Pau83b]. Conversions provide a modular language for composing rewrite rules into rewrite strategies.

#### 1.4.1.2 Main Automatic Procedures

**Relational Reasoner** I devised a simple tactic that automatically solved goals that depended on basic properties of order and equivalence relations, such as symmetry, antisymmetry, reflexivity, irreflexivity, transitivity and linearity.

**Arithmetic Reasoner** I implemented a new inference rule for solving linear inequalities over the integers, based on Bledsoe's *sup-inf* algorithm [Ble75]. The chief enhancement I made was to take full advantage of the linear arithmetic properties of non-linear arithmetic functions and non-arithmetic functions that have integer values.

**Type-Checker** All type checking in the Nuprl system is done by proof. Enhancements I made to the type checking tactics included adding much better capabilities for reasoning about type inclusion (necessary because terms in Nuprl's type theory frequently have multiple types) and enabling the type checking of definitions with binding structure. Such definitions were new to V4.

#### 1.4.1.3 Other Features

**Recursive Definitions and Induction Tactics** I implemented a methodology for simply defining general recursive functions using the $\mathsf{Y}$ combinator. The potential for doing this had been previously recognized by Howe and Allen, but had not been exploited to any extent in Nuprl V3. Much of the work here was in developing well-founded-induction tactics for use in proofs of totality of recursive functions. These tactics had to work on the edge of Nuprl's type theory.

**Universe Polymorphism** Constructive type theories including Nuprl's have a cumulative hierarchy of type universes. It is common when universes are mentioned to have some kind of scheme for implicitly quantifying over the levels of these universes. Such a feature was added in Nuprl V4.1. I had to do a significant amount of work to Nuprl's matching code to ensure that these levels would be appropriately instantiated whenever lemmas were applied.

### 1.4.2 Formal Algebra

#### 1.4.2.1 Approach in Nuprl's Type Theory

Nuprl's type theory provides a more restrictive environment for doing mathematics in than say Zermelo-Fraenkel set theory. For instance, when considering how to represent ideals of rings, I was faced with several alternatives, none of which was that satisfactory. On the other hand, I was able to develop the basic theory of common algebraic classes such as monoids, groups, rings and modules in a style similar to that adopted in the computer algebra system Axiom [JS92, DT92, DGT92], and had success with interpreting free constructions computationally.

I took approaches towards inheritance and subtyping of classes which were very simple, but which functioned adequately and highlighted features that one would want to include in a more sophisticated approach.

#### 1.4.2.2 Permutations

I gave a constructive development of the group of permutations on an arbitrary type and then specialized this development to permutations on a finite set. For example, I proved that every permutation is a product of pairwise interchanges. I applied this theory of permutations to developing a theory of the permutation relation on lists. I compared this approach to one based on a recursive definition of the permutation relation and to one based on a *count* function which computed the multiplicity of elements in lists. The description of this work can be found in Chapter 7.

#### 1.4.2.3 Finite Multisets and Finite Sets

I developed a theory of (finite) multisets which included definitions and characterizations of basic multiset operations and predicates. Finite sets were defined as a subtype of these finite multisets and many of the finite multiset operations were given alternative characterizations on sets.

The novelty of this development was chiefly in the use of Nuprl's quotient type [$C^+$86]. The quotient type allows hiding of internal structure of types; multisets were defined from lists by hiding the order of elements in lists. The quotienting operation on types does not group elements of a type into equivalence classes; instead it merely changes the equality relation associated with a type. The significance of this is that quotient types can then be used to give an abstract view of computable functions.

#### 1.4.2.4 Summations

I developed theories of summation over monoids, rings, modules and algebras. I experimented with several summation operators that took indices indexing the expressions being summed over from integer ranges, lists and multisets. Theorems were proven about the way sums can be rearranged and eliminated and how they interact with other operations.

#### 1.4.2.5 Factorization Theory

Factorization is an important basic topic in computer algebra, as well as in mathematics in general. Factorization theory is commonly studied over integral domains, though the basics can be formulated over abelian monoids with cancellation: the non-zero elements of an integral domain under multiplication always form such a monoid.

I developed the elementary theory of factorizations in cancellation monoids, and proved a theorem characterizing when a cancellation monoid is a unique-factorization (or factorial) monoid. The fundamental theorem of arithmetic was shown to be a special case of this theorem.

The case study illustrated the importance of combinatorics and discrete mathematics in algebra. Much of the work in developing these lemmas was in first creating the theory of permutations mentioned previously, and characterizing the notion of 'essential uniqueness'.

### 1.4.3 Polynomial Arithmetic

Abstract algebra and the theory of abstract data types (ADT's) in programming languages have strong similarities that have pointed out many times in the ADT literature [Wir90]; in each case one first defines some class of objects, each object with a domain and certain operations over that domains that have certain abstract properties. Then one studies the characteristics of the objects, drawing on just what one knows from the class definition.

When working in constructive type theory, the same kinds of class definition can be used in either case. In particular, every definition of a class, familiar in abstract algebra, can be viewed as an ADT specification.

#### 1.4.3.1 Specification

I created algebraic classes for monomials and polynomials based on the characterization found in Lang [Lan84] or Bourbaki [Bou74] of the algebra of polynomials as being a free monoid algebra over the ring of coefficients and the free abelian monoid of indeterminates.

An interesting characteristic of this development was the treatment of freeness properties. These freeness properties were viewed as the specifications for functions that instantiated the indeterminates in monomials and polynomials.

#### 1.4.3.2 Implementation

I based the implementation on the standard sparse representation of monomials and polynomials used in most computer algebra systems [DST93, Zip93a]. This representation involves using association lists (a-lists) of indeterminates and exponents to represent monomials, and a-lists of monomials and coefficients to represent polynomials. The keys (indices) of these a-lists were drawn from linear orders and the keys in an a-list were always maintained in order. This ordering requirement resulted in the maintenance of monomials in lexicographic order.

#### 1.4.3.3 Verication

The verification into two stages:

1. I characterized a-lists as defining functions of finite support; that is, functions that return some default value on all but a finite number of arguments from their domains. All the operations on monomials and polynomials were characterized in terms of these functions of finite support. All inductive proofs were localized to this part of the verification.

2. I verified all the algebraic properties of the monomial and polynomial operations solely by referring to these function-of-finite-support characterizations, never by referring to the recursive definitions. These verifications involved a significant amount of algebraic manipulations, but never one induction. The algebraic manipulations were all conceptually straightforward, but many, especially those that involved monotonicity reasoning, would be very difficult to implement in present-day computer algebra systems.

Since a-lists were common to both the monomial and polynomial constructions, I performed much of the initial verification of their properties while considering them to be an implementation of a group class and a monoid copower class. I showed that any implementation of the monoid copower class can simply be specialized to obtain an implementation of the free abelian monoid class, and I generalized the construction of a monoid copower to obtain a free monoid algebra.

## 1.5 Previous and Related Work

### 1.5.1 In Nuprl

The Nuprl project [CB83, $C^{+}86$] grew out of the earlier PL-CV [CJE82] and LambdaPrl [Bat79] projects in program verification and synthesis at Cornell, and the LCF project [GMW79] at Edinburgh. From LCF, Nuprl borrowed the idea of a tactic-driven refiner and the ML language for the tactics. Nuprl's original arithmetic decision procedure '`arith`' came from the PL-CV2 system, as did ideas for the user interface. In the early 1980's the project was under the principle guidance of R. Constable and J. Bates, and in the later 1980's, D. Howe and S. Allen took over from J. Bates. The Nuprl V4.1 implementation was started around 1990 with S. Allen doing much of the design work for a new editor, R. Eaton implementing the new editor and myself developing the tactics and libraries. All together over the years, over a dozen people have been involved in various ways with the Nuprl project.

The Nuprl type-theory is based on one of Martin-Löf's [ML82]. Nuprl's type-theory is discussed in more detail in Chapter 2. Theories developed in Nuprl include the fundamental theorem of arithmetic [How87], metatheory [Kno87, How88a, ACHA90, CH90], category theory [AP90], Ramsey's theorem [Bas89], Higman's lemma [Mur90], hardware verification [BD89, Jac91, Lee92] and software verification [AL92, How88b]. The previous tactics used in Nuprl were developed in chief by Howe [How88a]. In Chapter 3, I compare my tactics with those of Howe as well as with some rewriting tactics that Basin developed [Bas89].

Others have experimented with constructing algebraic classes in Nuprl: Altucher and Panangaden produced a definition of a class of categories [AP90], and Basin and Constable discuss abstract data-type definitions for multisets and for propositional logic [BC93].

Several major changes were made in moving from Nuprl V3 to Nuprl V4.1, aside from my work on the tactics and theories. They included:

- the addition of the abstraction (definition) facility,
- moving to a completely new display-form selection and formatting facility,
- the addition of a rule interpreter, so that most rules are not hard coded, but represented by objects in Nuprl's library,
- the introduction of rules for universe-level polymorphism,
- the addition of a reflection mechanism [ACHA90],
- the creation of a World-Wide-Web server for Nuprl so that theories can be interactively browsed from across the Internet.

### 1.5.2 Theorem Proving

DeBruijn's Automath project was an early and very influential investigation into techniques for mechanically proof-checking mathematics [dB80]. Van Jutting formalized all of a foundational text on elementary analysis — Landau's "Grundlagen" — in Automath [Jut77].

Recently, more mathematics has been formalized in the MIZAR system than any other. MIZAR [Rud92] has been developed over the last 20 years by a team under the leadership of A. Trybulec at the Bialystok branch of the University of Warsaw. It is based on classical first-order predicate logic, extended with second-order schema, and Tarski-Grothendieck set theory. Roughly speaking, this set theory is like Zermelo-Fraenkel set theory, extended with uncountably many inaccessible cardinals. All work done in Mizar is grouped into *articles*. Currently, over 300 articles have been written in the MIZAR language by over 60 authors. These articles contain over 6000 theorems in total. Articles are published in a *Journal of Formalized Mathematics* [Mat94] which is largely-automatically type-set from information in the MIZAR database. The subjects of the articles have been mostly in the fields of analysis, topology and algebra (including some universal algebra and category theory).

I think there are several keys to MIZAR's success. Firstly, it started with a set theoretic framework which is known to be theoretically adequate for all of mathematics, including category theory. Secondly, a rich type-theory was layered on top of the set theory. The type theory allows for the definition of subtypes and parameterized types, and has a *structure* facility for the definition of algebraic classes. The system copes automatically with set subtyping relationships between elements of classes that have different underlying signatures. Section 5.4 explains my terminology here. Unlike in Nuprl, all type-checking is done automatically, before proof. Thirdly, much effort has been put into the organisation of articles in the MIZAR database to ease and speed cross-referencing between articles. Typically an article draws on the definitions and theorems from many previous articles. The level of automation is surprisingly low when compared to that in most other theorem-proving environments. This point underscores the significance of the other design decisions in constructing MIZAR.

It is instructive to note that MIZAR is able to reap many of the benefits associated with using a type theory, without having to use a type theory as the foundation of their logic.

In terms of applying theorem-provers to hardware and software verification, most success has been with the NQTHM system of Boyer and Moore [BM79, BM88a]. Accomplishments include the checking the RSA public-key encryption algorithm [BM84] and the verification of microprocessor designs [HB92]. NQTHM has also been used to formalize Gödel's incompleteness theorem [Sha86]. The generation of proofs in NQTHM is highly automated. The user commonly only guides proofs by perhaps giving a few high level hints and suggesting useful lemmas. NQTHM automatically guesses how to do inductions and how to prove the subgoals of inductions. NQTHM also has a linear-arithmetic decision procedure tightly integrated in with the the prover program. NQTHM's logic is significantly weaker than Nuprl's: it is quantifier-free and includes a theory of recursive functions over Lisp-like S-expressions. Its strength is roughly that of Primitive Recursive Arithmetic (PRA). This logic is too weak for abstract algebra: there is no way to define algebraic classes of objects and reason with them in ways common in algebra, though 'functional instantiation' extensions do allow some basic algebraic reasoning.

The HOL system [GM93] is a tactic-based interactive theorem prover with a classical logic similar on Church's simple theory of types [Chu40] but with the addition of a type-polymorphism scheme similar to that found in the ML functional programming language. This theory is slightly weaker than ZF set theory. HOL has mostly been used in domains related to hardware and software verification, though its foundational theories are quite general purpose and some success has been had with more abstract mathematics. Harrison developed some real analysis covering topics including limits of series, differentiability and properties of transcendental functions [Har92]. Harrison and Thèry demonstrated using the Maple computer algebra system [CGG+91] to factor and integrate expressions for HOL [HT93]. Maple's operations were verified in HOL by carrying out the much simpler inverse operations of expanding out factors and differentiation. E. Gunter[Gun89] developed a basic theory of groups, proved the group isomorphism theorems and showed that the integers mod $n$ form a group. A. Gordon demonstrated how to prove the binomial theorem over arbitrary rings [Gro91].

Formulating abstract algebra in HOL is awkward because the type theory provides so few features in comparison with set theory. Decisions have to be made about how to represent the most basic notions of algebra in the type theory: notions such as set, cartesian product and function space. It is simplest to use a HOL type to represent a set, but also rather limiting. For instance, there is no general way of defining a subtype relation that asserts that one type is a subtype of another. Still, Harrison was able to use this approach in his analysis work when he defined the notion of a topology . Another approach to representing a set is to use a type together with a unary predicate on that type. Gunter took this approach, representing the carrier set $\sigma$ of a group by a type $\tau$ and a predicate $p$. She then defined the function space $\sigma \rightarrow \sigma \rightarrow \sigma$ of the operator of a group as the HOL type $\tau \rightarrow \tau \rightarrow \tau$ together with the predicate $\lambda f{:}\tau \rightarrow \tau \rightarrow \tau.\ \forall x, y{:}\tau.\ (p\ x \wedge p\ y) \Longrightarrow p\ (f\ x\ y)$. The awkwardness here is that all instances of groups now must have operators defined for arguments over the whole of type $\tau$ rather than just over $\sigma$. Further, the equality relation provided by the type theory for functions like this group operator is stronger than desired: two functions $f$ and $g$, thought of as being members of the function space $\sigma \rightarrow \sigma$, might agree on all arguments in $\sigma$, yet might not be considered equal by the HOL type-theory's equality relation because they disagree on an argument in $\tau$ but outside of $\sigma$.

Another major limitation of HOL's current type theory is that any quantification over types is always universal and always on the outside of any formula. Embeddings of set theory without this restriction are being explored by R. Jones at ICL in the UK. In this work, HOL's type theory takes on more the role of a metalogic. Recently, M. Gordon has also been investigating techniques for merging HOL with set theory [Gor94].

Other theorem provers that use a constructive type theory and encode logic using the propositions-as-types correspondence include Alf [ACN90], LEGO [Pol90], and Coq [DFH+91]. Alf uses a more recent type theory of Martin-Löf's. Both LEGO and Coq use the Calculus of Constructions (CoC) [CH85], extending it with a kind of inductive definitions [Luo89, PPM89]. Bailey developed a concrete theory of polynomials in one variable in the LEGO system and has proven the correctness of Euclid's algorithm over these polynomials [Bai93]. Aczel and Barthe are currently investigating doing Galois theory in LEGO [Acz93, Bar93].

The most significant differences between the CoC based type theories and Nuprl's, from the point of view of formalizing algebra, are in the treatment of equality and in that the CoC-based theories don't have a *set* type constructor. Equality between elements of types is always essentially a $\beta\eta$ equality: two terms are equal just when they evaluate to the same normal form and their subterms are equal. There is no notion of quotienting a type. Without a construct corresponding to Nuprl's set type, the only way of of forming something close to a subtype is by using a dependent product type. Type-checking is decidable in these systems, but doing abstract algebra is rather involved. The approaches that have been looked at involve using a somewhat cumbersome encoding for sets called *setoids*. I discuss these in Chapter 5.

Theorem provers such as Larch [GH93], IMPS [FGT92b], PVS [ORS92], and 2OBJ [GSHH91] have notions of *theories* or *modules* which allow the collection together of type definitions, operator definitions and specification of predicates that the operators must satisfy. Further, these modules can be parameterized by one another and can inherit structure from one another. As with module-like structures in programming languages, these modules help structure these systems' libraries of definitions and theorems.

Although these superficially resemble the algebraic class definitions I have made in Nuprl, they are defined at a level of abstraction above the type system and cannot be included in formulae in the same way that types are. For example, I think it would be hard if not impossible in these systems to define a class of groups using one of these modules and then show that a ring structure can be imposed on the set of homomorphisms between the groups.

Interestingly, the PVS system also has dependent data types similar to Nuprl's dependent product and dependent function types, so the class definitions could be set up as described in this thesis. However, without some version of quotient types it would be difficult to verify *constructive* implementations of the classes where the equality on the representation type must be weakened in order to hide irrelevant detail.

One other theorem-proving system worth mentioning is McAllester's Ontic system [McA89]. This system is based on Zermelo-Fraenkel set-theory. A milestone reached in it is the Stone representation theorem. Chen looked at adapting the inference algorithms that McAllester devised to the Nuprl environment [Che92].

### 1.5.3 Computer Algebra

Clarke and Zhao have added theorem-proving capabilities to Mathematica [Wol91] to create their Analytica system [CZ92]. They have impressive results in proving equivalences of sums of series, but their work has been hindered by the lack of rigor inherent in the Mathematica environment.

In several computer algebra systems, much effort has been put into allowing computations over a wide variety of types; for example Axiom [JS92, DT92, DGT92] which evolved from the ScratchPad system at IBM. There are strong similarities between Axiom's approach to constructivity and the approach adopted in this thesis.

My interest in computer algebra has been stimulated by discussions with Zippel who has designed his own system Weyl [Zip93b]. An ongoing project at Cornell is to set up links between Nuprl and Weyl.

### 1.5.4 Constructive Mathematics

Excellent introductions to constructivism in mathematics have been given by for example, Troelstra and van Dalen [TvD88] and Dummett [Dum77]. Throughout the history of mathematics there has been some sensitivity to constructivity in algebra. For example, Edwards [Edw89] wrote in summarizing Kronecker's views: "Kronecker believed that a mathematical concept was not well defined until you had shown how, in each specific instance, to decide [algorithmically] whether the definition was fulfilled or not." However, after the turn of the century when significant new results were proven non-constructively (for example, Hilbert's Basis Theorem), and non-constructive set-theoretic foundations were established for mathematics, constructivist sympathies were rejected by many mathematicians. There were definitely exceptions, especially among logicians; for example, there was Brouwer who founded a school of 'Intuitionistic' mathematics that was dogmatically constructive. Brouwer's work was revived by Heyting [Hey66] and Bishop [Bis67, BB85] who tried to show how to systematically hide constructive details so that constructive mathematics more resembled classical mathematics. Today, a few mathematicians are exploring constructive algebra in this light [MRR88, BB85]. Other investigations have been carried out where the computations are made more explicit [FS55, MN79].

The strongest revival of interest in constructivity in algebra has undoubtably come from those concerned with the theory and design of computer algebra systems. One of the most significant results here has been the discovery and subsequent refinement by Buchburger of the Gröbner Basis algorithm for finding an elegant normal form for the generating set of an ideal in polynomial rings over a field, given an arbitrary initial finite set of generators of the ideal. This algorithm in some form is used in many computer algebra systems today for solving systems of polynomial equations. This algorithm and others are surveyed in many of the new texts that have come out recently on the mathematics of computer algebra [BWi93, CLO92, DST93, Mis93, Zip93a].

## 1.6 Layout of Thesis

The layout of the thesis is as follows:

- Chapter 2 gives background information on Nuprl's type theory and the present state of the Nuprl V4.1 system. This chapter ends with a summary of the theories I have developed in V4.1.
- Chapter 3 surveys the current tactic collection, highlighting the new contributions I have made, but also trying to give a general overview of the tactics.
- Chapter 4 covers in depth the rewrite package that I set up for Nuprl V4.1.
- Chapter 5 discusses the alternatives approaches that I considered to making definitions in Nuprl's type theory for algebraic classes, and indicates why I made the choices that I did.
- Chapter 6 gives introduces the information on the basic algebraic classes that I set up. It also covers my treatment of order relations on these classes, the theory of summations, and tactics developed to support reasoning over these algebraic classes. The work described in later chapters depends heavily on this work.
- Chapter 7 describes the development of the theory of permutation functions and relations.
- Chapter 8 covers work on unique factorization in cancellation monoids, ending with an example of how the fundamental theorem of arithmetic is a special case of the last theorem proven.
- Chapter 9 describes my implementation of finite sets and multisets and operations over them, illustrates the use of Nuprl's quotient type and provides a warm-up case study in ADT specification and implementation.
- Chapter 10 gives an extended case study in the specification and implementation of an ADT for multivariate polynomials.
- Chapter 11 summarizes the contributions of the thesis, outlines directions for future work, and discusses both the appropriateness of Nuprl's type theory and the dependence of the work described on it.

# Chapter 2

# Background on Nuprl

## 2.1 Type Theory

The most common formal system studied in logic as a foundation for mathematics is first-order predicate calculus and some set theory, most commonly Zermelo-Fraenkel set theory [Sho67]. Nuprl uses instead a *type theory* which takes the place of both predicate calculus and set theory.

### 2.1.1 What is Type Theory?

Type theory is an active research area in mathematics, logic and computer science and a diverse range of theories are collected under this name. Here, I look at a few of the characteristics of type theories, concentrating on those relevant to Nuprl's.

In set theory, one often thinks of there existing a platonic universe of sets, and set notation provides a way of naming many of the principal ones. In type theory, one starts out assuming the existence of specific base sets or *types* like the booleans and the integers. There are then standard ways for producing richer types, for example, using the operations of cartesian product and function space formation. Type theories provide primitive operations for creating elements higher up this hierarchy from elements lower down. For example, a pairing operation creates elements of cartesian products and lambda abstraction creates elements of function spaces.

Type theories also provide primitive operations for taking apart elements and define notions of evaluation on elements. For example, the $\pi_1$ function selects the first element of a pair $\langle a, b\rangle$ so that the element $\pi_1(\langle a, b\rangle)$ evaluates to $a$.

Type theories are of much interest in computer science because often at least a subset of the elements of types can be regarded as programs and data in a functional programming language. The type theories themselves then provide a formal language for reasoning about these programs. Many type theories are abstractions of the type systems that have been used in programming languages from Algol68 onwards.

Theorem-prover designers have found type theories appealing because they intrinsically impose much more structure on the world than set theory, and narrow the gap between the theory foundations and statements about objects of interest. Often too, it is convenient that a major subject matter for theorem provers is program verification.

The study of type theories is usually taken to have started with Russell's 'theory of types' [Rus08, WR27] where a rather complicated system of types are introduced in order to avoid certain 'vicious circle' paradoxes. Church introduced a 'simple theory of types' [Chu40] that was adapted for use in the HOL theorem-proving system [GM93]. A recently-developed family of type theories is that of *constructive* type theories [Gir71, CH88]. These exploit a notion that has come to be known as the 'propositions-as-types' correspondence [CF58, Sco70, Con71] where every logical proposition corresponds to a type, and a proof of a proposition involves finding an element of the type corresponding to the proposition. Since elements of types are often programs, a phrase commonly associated with the 'propositions as types' approach is 'proofs as programs' [BC85]. These type theories are *constructive* because they yield a constructive or intuitionistic logic, and because they give a recipe for automatically building functions that effect the constructions that theorems in constructive logic and mathematics talk about.

Nuprl's type theory [C$^+$86, All87a, All87b] is most closely related to a type theory proposed by Martin-Löf in 1979 as a foundation for constructive mathematics [ML82]. The main differences are:

- Martin-Löf's four kinds of judgement are reduced to one. Roughly speaking, a *judgement* is a kind of sequent. The semantics of Nuprl's judgement is different from any of Martin-Löf's in that a judgement in Nuprl assumes rather than requires the well-formedness of hypotheses. Nuprl's treatment of equality in its judgement is more complicated than that in Martin-Löf's. These changes enabled particular kinds of induction rule to be defined. Without them, the approach we have been using in Nuprl of first introducing general recursive functions and then proving them total over some domain by induction would not have been possible.

- In Nuprl, equality of types is explicitly intensional (i.e. structural). Martin-Löfclaimed in his paper that his equality on types was extensional (types are equal if they have the same members), though he never gave rules to make an intensional interpretation of his type theory inconsistent. This change enabled the reduction in kinds of judgement.

- Nuprl's type theory has several extra types including the *set* type [Con85a], the *quotient* type [Con85a], *recursive* types [CM85], and *partial function* types [CS87].

Allen has given a semantics for Nuprl's type theory without the recursive or partial types [All87a, All87b]. This semantics takes the form of a second-order positive inductive definition that is both classical-set-theoretically valid and acceptable to most constructivist mathematicians. The definition is of a relation from which a type membership relation and a typed equality relation are derived. The definition essentially if of a term model in that terms do not denote anything other than themselves. Mendler [Men88] gave a semantics for Nuprl's recursive types and Smith [Smi89] gave a semantics for the partial function types. Howe [How91a] has given a set-theoretic model in which terms denote sets, and has shown by this model that it is consistent to extend Nuprl's type theory with oracle functions so that the logic created by the propositions-as-types correspondence is classical.

I give below an informal account of Nuprl's type theory.

### 2.1.2 Basic Types

In Nuprl's type theory, the word *term* encompasses the constructs of its functional programming language, types and propositions.

The programming language terms include the untyped lambda calculus, and constructors and destructors associated with each of the types listed below.

A lazy evaluation relation is defined on terms. Any term evaluates to at most one canonical term, and canonical terms always evaluate to themselves.

The basic type constructors of Nuprl's type theory that are relevant for this thesis include:

- The *integers* $\mathbb{Z}$. Primitive operations include binary $+, -, \times, \div$, *rem* (remainder), and unary $-$. Defined subsets of the integers include the non-negative integers $\mathbb{N}$, and the positive integers $\mathbb{N}^+$ (occasionally written as $\mathbb{Z}^+$).

- A *dependent-function* type constructor $\rightarrow$. If $A$ is a type and $B_x$ is a family of types, indexed by $x \in A$, then $x{:}A \rightarrow B_x$ is the type of functions $f$, such that $f(a) \in B_a$ for all $a \in A$. If $B_x$ is the same for all $x \in A$, I write the type as simply $A \rightarrow B$. I assume that $\rightarrow$ associates to the right.

  Since all functions constructible in Nuprl's type theory are computable, each type $A \rightarrow B$ is considered as containing only the computable functions from $A$ to $B$ rather than all set theoretic functions.

  Every canonical element of a dependent-function type is a lambda term $\lambda x.t$.

  Dependent-function types are sometimes known as $\Pi$ types, in which case a notation commonly used is $\Pi\ x{:}A.\ B_x$. Elsewhere these types are sometimes called *cartesian-product* or *dependent-product* types. These names are not used here in order to avoid confusion with Nuprl's dependent-product type.

- A *dependent-product* type constructor $\times$ . If $A$ is a type and $B_x$ is a family of types, indexed by $x \in A$, then $x{:}A \times B_x$ is the type of pairs $\langle a, b\rangle$, such that $a \in A$ and $b \in B_a$. If $B_x$ is the same for all $x \in A$, I write the type as simply $A \times B$. Sometimes I write $A \times A$ as $A^2$. I assume that $\times$ associates to the right.

  Dependent-product types are sometimes known as $\Sigma$ or *dependent sum* types, in which case a notation commonly used is $\Sigma\ x{:}A.\ B_x$.

- A binary (disjoint-) *union* type $+$. If $A$ and $B$ are types, then $A + B$ is a type. Its canonical elements are of form $\mathrm{inl}(a)$ (read 'in left') and $\mathrm{inr}(b)$ (read 'in right') for $a \in A$ and $b \in B$.

- A *set* type constructor $\{\cdot{:}\cdot|\cdot\}$. If $A$ is a type and $P_x$ is a proposition in which $x$ of type $A$ may occur free, then $\{x{:}A|P_x\}$ is the type of those elements $x$ of $A$ for which $P_x$ is true.

- Recursive types. These include $A$ $\mathsf{List}$ for finite sequences of elements of type $A$. The operation $a{::}s$ appends element $a$ to the front of sequence $s$, and the empty sequent is denoted by $[]$. The type theory also includes a type constructor for building types of tree-like data-structures.

- *Universes* of types $\mathbb{U}_i$ for $i = 1, 2, 3\ldots$ . $\mathbb{U}_i$ includes as base types $\mathbb{U}_j$ for all $j < i$ and is closed under the type constructors listed above. Note that Section 2.1.5 describes the allowable expressions that may be used for the subscript $i$ and the conditions under which the subscript is often dropped.

- The type $\mathsf{Void}$. It has no elements. The type $\mathsf{Unit}$. It has the one element '$\cdot$' (read as 'it'). The boolean type $\mathbb{B}$ has two elements: $\mathsf{tt}$ for 'true' and $\mathsf{ff}$ for 'false'. $\mathsf{Unit}$ and $\mathbb{B}$ are defined types, but for nearly all intents and purposes, they can be thought of as being primitive.

Every type has an equality relation associated with it. A three-place atomic proposition $\cdot = \cdot \in \cdot$ is used to refer to this equality. The relation $x = y \in T$ means that $x$ and $y$ are members of type $T$ and are equal by the equality relation associated with $T$. Sometimes, I write $x = y \in T$ as $x =_T y$, and when $T$ is obvious from context, the reference to $T$ is dropped altogether. Functions can only be given a function type when they respect the type equalities of the components of the function type; if function $f$ has type $S \rightarrow T$, then $fx =_T fx'$ must hold whenever $x =_S x'$. Similarly, all members of a function type are assumed to to respect the equalities of the components of that type. This assumption of *function extensionality* is non-trivial because of the way in which the equality relation associated with a type can be changed.

Specifically, the equality associated with a type can be weakened using Nuprl's *quotient type* constructor; if $R$ is an equivalence relation on type $T$, then the quotient type constructed from $T$ and $R$ is written $x, y{:}T//xRy$. The inhabitants of $x, y{:}T//xRy$ are the *same* as the inhabitants of $T$; the quotient type does not group elements of $T$ into equivalence classes. Inhabitants are considered equal when they are related by $R$.

I use fairly usual notation for programming language constructs. Function application is designated by juxtaposition. For example, I write $f\ a$. Application is assumed to associate to the left, so $(f\ a)\ b$ is written $f\ a\ b$. Often I use infix notation for the application of binary curried-functions. For example, if $* \in T \rightarrow T \rightarrow T$, then for $(*\ a)\ b$ I write $a * b$. It should be obvious whenever infix notation is being used.

### 2.1.3 Propositions

Logic is injected into type theory using the propositions-as-types correspondence, so all propositional term constructors are defined from types [Con85b]. The definitions are:

$$\begin{array}{rcl} \perp & =_{def} & \mathsf{Void} \\ A \wedge B & =_{def} & A \times B \\ A \vee B & =_{def} & A + B \\ A \Rightarrow B & =_{def} & A \rightarrow B \\ \forall x{:}A.\ B_x & =_{def} & x{:}A \rightarrow B_x \\ \exists x{:}A.\ B_x & =_{def} & x{:}A \times B_x \\ \downarrow \exists x{:}A.\ B_x & =_{def} & \{x{:}A \mid B_x\} \\ \mathbb{P}_i & =_{def} & \mathbb{U}_i \end{array}$$

The symbol $\perp$ denotes falsity. Negation, $\neg A$, is defined as $A \Rightarrow \perp$, and bi-implication (if and only if) $A \iff B$ is defined as $(A \Rightarrow B) \wedge (B \Rightarrow A)$. The type $\downarrow \exists x{:}A.\ B_x$ is read as 'squash exists' Not shown is the definition of the propositional relation $a = b \in T$ since this is actually a primitive type in the Nuprl type theory (this type has one element when the equality is true and is otherwise empty). As one can see, the encoding is very direct.

Each predicate-logic expression corresponds to a type with the type being inhabited just when the predicate-logic expression is provable. The proof of a logical expression specifies exactly how to construct a term that inhabits the type corresponding to the logical expression. Sometimes the inhabitant is interesting; for example it might be a function that computes something useful. In this case, we can view the logical expression corresponding to the type it inhabits as a kind of program specification. When I talk about the *computational content* of a logical expression, I am referring to the possible inhabitants of the corresponding type.

In the discussions of computational content in this thesis, I recommend that the reader refer back to the above definitions and try to imagine what kinds of terms might inhabit the types that correspond to the propositions being discussed.

Nuprl's logic is well-suited to constructive mathematics, but it also can support classical styles of reasoning.

### 2.1.4 Sequents, Rules and Proofs

Nuprl's rules are formulated in a *sequent calculus*. A sequent in Nuprl consists of a list of 0 or more hypotheses $H_1, \ldots, H_n$ and a conclusion $C$. It is usually written as:

$$H_1, \ldots, H_n \vdash C.$$

Each hypothesis $H_i$ is either a proposition $P$ or a declaration $x{:}T$ declaring variable $x$ to be of type $T$. The conclusion is a proposition. Collectively, I refer to hypotheses and the conclusion as *clauses*. A declaration $x{:}T$ as hypothesis $H_i$ binds free occurrences of $x$ in hypotheses $H_{i+1} \ldots H_n$ and in conclusion $C$. For this reason, the order of the hypotheses is important. One can't arbitrarily permute hypotheses. Sequents are always closed; every free variable in some clause is bound by some declaration to the left. A sequent is considered true if one can prove the conclusion $C$ under the hypotheses $H_1, \ldots, H_n$.

Thinking purely type-theoretically, all clauses of a sequent are types. Hypotheses thought of as a propositions declare the type of a variable which is normally never visible. A sequent is true just when there exists a function from the types of the hypotheses to the type of the conclusion. In some Nuprl literature, the turnstile symbol $\vdash$ is written instead as $>>$ to be suggestive of a function arrow.

Rule in a sequent calculus are commonly written in the form:

$$\frac{\mathcal{A}_1 \quad \ldots \quad \mathcal{A}_n}{\mathcal{C}}$$

where $\mathcal{A}_i$ and $\mathcal{C}$ are sequents and $n \geq 0$. The $\mathcal{A}_i$ are the antecedents of the rule and $\mathcal{C}$ is the consequent. Such a rule can be read top down as saying that if all the $\mathcal{A}_i$ are true, then $\mathcal{C}$ is true. The rule can also be read bottom up as saying that in order to prove $\mathcal{C}$ is true, it is sufficient to prove that all the $\mathcal{A}_i$ are true.

The rules in the intuitionistic logic yielded by Nuprl's type theory are similar to those in Gentzen's LJ system [Pra71]. Nearly all the logic rules, when read top down, tell us how to introduce a logical connective or quantifier in a hypothesis or the conclusion. When read bottom-up they explain how to break down or decompose the connective. I refer to such rules in Nuprl as *decomposition* rules, because rules are always applied in a bottom-up fashion.

Slightly simplified versions of the rule for decomposing $\Rightarrow$ in a hypothesis and in the conclusion are

$$\frac{\Gamma, A \;\Rightarrow\; B, \Delta \vdash A \qquad \Gamma, A \;\Rightarrow\; B, \Delta, B \vdash C}{\Gamma, A \;\Rightarrow\; B, \Delta \vdash C}$$

and

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \;\Rightarrow\; B}$$

Another rule, the hypothesis rule, states that

$$\frac{}{\Gamma, A,\ \Delta \vdash A}$$

Here $A$ and $B$ stand for arbitrary propositions, and $\Gamma$ and $\Delta$ stand for arbitrary (maybe empty) lists of hypotheses. Note that the hypothesis rule is an example of a rule with no antecedents.

A proof of some proposition $P$ in Nuprl's logic is usually constructed by starting with the sequent $\vdash P$. One then applies rules bottom-up, building the proof tree upwards. Since most of the rules when viewed bottom-up decompose a connective, propositions generally get simpler as one moves from the root of the tree out along the branches. Branches of a proof tree terminate with such rules as the hypothesis rule above.

This style of theorem proving bears a close resemblance to the tableau method for proving theorems [Smu68], which is commonly taught in logic courses, and which students usually find the simplest to use.

Because rules are applied bottom-up, it is common to present Nuprl rules upside down. The general form of a rule is then:

$$\begin{array}{l} \mathcal{C} \\ \texttt{BY}\ \textit{rulename} \\ \quad \mathcal{A}_1 \\ \quad \vdots \\ \quad \mathcal{A}_n \end{array}$$

With this style of rule, proof trees have their root at the top, and their branches grow downwards. The full set of rules for the Nuprl type theory can be found in the Nuprl book [C$^+$86] and in the system library.

Thinking of the conclusions of sequents as types, all Nuprl rules have information about how to create an element of the conclusion of the rule consequent, given elements of the conclusions of each of the antecedents. When a proof of some proposition is completed, this information can be used to synthesize an inhabitant of the proposition, considered as a type. This synthesis process goes by the name of *extraction*.

### 2.1.5 Universe Polymorphism

An initially unappealing aspect of Nuprl's type theory is the stratification of types (and hence propositions) into universes, in a style reminiscent of Whitehead and Russell's in Principia Mathematica [WR27, Rus08], but much simpler.

In Nuprl V3 and before, the levels of universe terms were constants, which was inconvenient because often variants of the same lemma had be introduced which differed only in the levels of their universe terms.

In Nuprl V4.1, a kind of *universe polymorphism* was introduced, where levels in universe terms are replaced by *level expressions*. Level expressions are of form:

- $k$ where $k$ is a natural number constant $\geq 1$.
- $v$ where $v$ is a level expression variable. Level expression variables are implicitly quantified over levels; natural numbers $\geq 1$.
- $e\ k$ where $e$ is a level expression and $k$ is a natural number constant $\geq 0$. $e\ k$ is understood to be level $e$ plus $k$.
- $[e_1|\ldots|e_n]$ where $e_1 \ldots e_n$ are level expressions. $[e_1|\ldots|e_n]$ is understood to be the maximum of levels $e_1 \ldots e_n$.

The rules and semantics for this universe polymorphism were proposed by Howe [How91b]: a rule with clauses involving level variables is considered to be true just when the rule is true for all instantiations of level variables by constants. This semantics for universe polymorphism is different from that proposed by Allen [All87b].

The level expression '$e$ 1' is often abbreviated as $e'$.

Note that it is often convenient to suppress the explicit mention of the universe levels, especially when a level expression is simply the level variable $i$ or the level constant 1. Also, I sometimes transfer the prime character ($'$) to the term being subscripted. For example, I write $\mathbb{P}'$ instead of $\mathbb{P}_{i'}$.

### 2.1.6 Well-Formedness Checking

Nuprl's type theory is sufficiently complex that the problem of determining whether a term has a given type is in general undecidable: the halting problem [HU79] can be reduced to the type membership problem by constructing a type whose inhabitants are the numbers of those Turing machines that halt on zero input. A consequence of this is that there is no general way to check the well-formedness of arbitrary terms, since well-formedness of a term is expressed in the type theory by saying that the term has a type.

Instead, the semantics of sequents and the rules of Nuprl's type theory are set up so that the well-formedness of expressions is shown by proof. Every complete proof of a theorem in Nuprl contains not only a proof that the theorem is valid, but also a proof that the theorem is well-formed. The well-formedness proof is distributed through the proof of validity by adding extra premises to many of the Nuprl rules. For example, the rule for $\Longrightarrow$ decomposition on the right is (prettied up using the propositions-as-types correspondence):

$\Gamma \vdash A \Longrightarrow B$
`BY` implies decomposition on right at level $i$
  $\Gamma \vdash A \in \mathbb{P}_i$
  $\Gamma, A \vdash B$

The obligation to show that the proposition $A$ is well-formed is phrased as the universe membership sub-goal $\Gamma \vdash A \in \mathbb{P}_i$. As explained in Section 3.2, such well-formedness obligations are almost always solved automatically so the user need not be concerned with them.

Checking well-formedness by proof is unfortunately much slower than checking by some completely automatic type checker, and is a major source of inefficiency in the Nuprl system. Furthermore the nature of the rules causes the well-formedness of expressions to be rechecked many times over. Nearly all other theorem provers do their well-formedness checking entirely by completely automatic means, distinct from proof generation.

## 2.2 Mechanization

### 2.2.1 Overview

The Nuprl V4.1 system is currently used on Unix-based workstations that run X-Windows and hopefully will soon be ported to run on Macintoshes and PC's. All Nuprl code is either written in Common Lisp or a the functional language ML (see Section 2.2.8 for a description of the ML dialect used). The ML compiler is written in Common Lisp and compiles ML code by first translating it into Lisp and then compiling the Lisp code.

Mathematics in Nuprl is organized into blocks called *theories*. A theory is a linear list of various kinds of objects including definitions, theorems, and comments. Theories are stored as Unix files. Users load theories into the Nuprl environment called the *library* as and when needed.

Nuprl is an interactive system. The user develops theories by carrying on a dialog with a Nuprl session via special purpose editors as well as an ML top-loop. The editors are briefly described in Section 2.2.2 and Section 2.2.7.

### 2.2.2 Terms and Structured Editing

In Nuprl, a *term* is a general-purpose uniform tree-shaped data-structure. Terms are Nuprl's equivalent of Lisp's S-expressions, though they have more intrinsic structure; terms have provisions for specifying variables to be bound in subterms, and for *parameters* that allow the injection of families of constants such as natural numbers into the term language. Terms have a variety of uses:

- All propositions in Nuprl's logic are represented as terms, as are all expressions and types in its type theory.
- All kinds of objects in theories except proofs are represented as terms.

Note that this use of the word *term* is more general than the use introduced in Section 2.1.2, where it only refers to the constituents of Nuprl's type-theoretic language.

The definition of terms assumes the existence of syntactic classes of *variables* and *opids* (operation identifiers). The elements of these classes are alphanumeric strings starting with a letter. The '_' character is counted as a letter. The definition of terms also assumes that there is a collection of *parameter kinds* and that there is a set of *parameter values* associated with each parameter kind. The main parameter kinds are described later on.

The set of terms is inductively defined as the least set such that:

- if $v$ is variable, then $v$ is a term,
- if $n \geq 0$ and $m \geq 0$, if for $1 \leq i \leq n$ we have that $x^i_1, \ldots, x^i_{a_i}$ are variables and $t_i$ is a term, and if we define

  $$s_i \quad = \quad x^i_1, \ldots, x^i_{a_i}.t_i \quad ,$$

  then

  $$opid\{p_1{:}k_1, \ldots, p_m{:}k_m\}(s_1; \ldots ; s_n)$$

  is a term.

The parts of a term are:

- $opid\{p_1{:}k_1, \ldots, p_m{:}k_m\}$ is the *operator*.

  The parts of the operator are:

  - *opid* is the operator identifier.
  - $p_j{:}k_j$ is the $j$th parameter. $k_j$ is a parameter kind and $p_j$ is some parameter value appropriate to kind $k_j$.

- The tuple $\langle a_1, \ldots, a_n \rangle$ where $a_j \geq 0$ is the *arity* of the term.
- $s_i = x^i_1, \ldots, x^i_{a_i}.t_i$ is the $i$th *bound-term* of the term. This bound-term binds free occurrences of the variables $x^i_1, \ldots, x^i_{a_i}$ in $t_i$. Frequently, $a_i$ is 0, in which case I omit the '.' (period) preceding $t_i$.

The parameter kinds include:

- `token` for character strings
- `nat` for natural numbers
- `level-expression` for level expressions (level expressions are described in Section 2.1.5).

These kinds are abbreviated respectively as `t`, `n` and `l`.

Using the parameter mechanism, the number 3 is injected into the term language as the term `natural{3:n}`. Parameters are actually also used to inject variables into the term language: the variable `foo` when considered as a term is represented by the term `variable{foo:t}`. To improve readability, I never show a variable term written out in this way. Likewise, I abbreviate natural-number terms: I write `3` for the natural-number term 3.

When writing terms, I sometimes omit the brackets around the parameter list if it is empty.

### 2.2.3 Term Display and Entry

The visual appearance of each term constructor is governed by *display form* objects in the Nuprl library. Display forms give one control over

- The order in which binding variables, parameter values and subterms are displayed.
- The text separating each binding variable, parameter and subterm.
- Line-breaking and indentation
- Parenthesization. Display forms can be set up to introduce parentheses based on the relative precedences assigned to display forms and subterm slots of display forms.
- Iteration of terms. Often it is desirable to use special notation when similar terms are nested inside one another.
- Elision of subterms, binding variables and parameter values that are deemed uninteresting.

Display forms greatly increase the readability of terms. For example, the term:

```
all(int();i.all(int();j.exists(int();k.ge(k;multiply(i;j)))))
```

is usually displayed as:

$\forall$ `i,j:`$\mathbb{Z}$`.` $\exists$ `k:` $\mathbb{Z}$`. k` $\geq$ `i * j`

In this example, a special display form has been used for the nested `all` term constructors. Currently, all displays are generated using characters from a fixed-width ASCII font, extended with roughly 60 graphics characters. At some stage in the near future, it should be possible to use for example Display PostScript technology to generate displays multiple sizes and kinds of fonts, and two dimensional layout of formulae.

All terms shown in this thesis have been automatically formatted by Nuprl's current display routines.

Terms are interactively edited and viewed exclusively using a structured editor. The structured editor supports a variety of tree editing operations on terms. It also supports the editing of paragraphs of text within terms, with these paragraphs themselves having term trees embedded within them. This feature is particularly useful for typing in ML text that often has terms from Nuprl's object language embedded within it. Numerous examples can be seen of this throughout the thesis.

The structured editor deliberately has no capabilities for parsing the displayed text of a term back into the underlying term data-structure. This gives the user much greater freedom in designing notations and means that display forms can be changed independently of one another; when designing a grammar to be parsed, careful attention has to be paid to the inter-relations between the grammar constructs.

It is common mathematical practice to try to use as concise notation as possible. Conciseness enhances comprehension (and also speeds writing). Apparent ambiguities are resolved by the reader's knowledge of the context the notation is presented in, and of what does and doesn't make sense semantically.

In several theorem proving projects (MIZAR [Rud92] and Isabelle [Pau90]), for example), much effort has been expended on designing parsers so that reasonably concise notation can be typed using character-based text-editors [1]. These parsers often use type checking to resolve ambiguities and type inference to infer implicit type arguments. The automatic inference of implicit type information is common too in such systems as HOL [GM93] and Coq [DFH$^+$91]. Still, such an approach limits notation to being in one font without a full range of mathemtical symbols, subscripts and superscripts, and doesn't support two-dimensional notation. A

[1] By *character-based* I mean editors such as `emacs` or `vi` in Unix systems where there is usually a one-one correspondence between the set of character byte-codes in files that are edited, and the set of glyphs that are used to display those characters.

partial solution is to separate notations for input and output, in which case fancier formatting can be used for output. This approach is now common in computer algebra systems.

The structured editor approach has the advantage that very concise notations can be used for both input and output. With Nuprl's editor, short mnemonic alphanumeric key sequences are assigned to various constructs so entry is possible both by touch-typing and picking constructs from menus.

Another advantage of the Nuprl editor is that it allows display forms to be changed in the middle of a Nuprl session, with these changes taking immediate effect. This feature is frequently used when many display forms elide unimportant arguments of terms. When eliding display forms are defined, non-eliding backup display forms are usually also defined. Users then in the middle of a session can ask for the backup display forms to be used when they want to see what the elided arguments are.

This structured editing approach has several disadvantages. In theorem-proving there are far fewer conventions for notation and users often invent new notation to hide formal detail. One user's concise notation frequently will be hard for another to read. When working interactively with Nuprl, it is possible to mouse-click on notation and ask for it to be explained, but this option isn't available on paper. It can be impossible to figure out in printouts of notation what precisely is meant; at least with machine-parseable notation, the reader knows that he or she should be able (in principle) to figure out what is going on.

Another disadvantage is that it takes a lot of work to get the ergonomics of a structured editor right. Currently, new users of Nuprl take some time getting used to the editor. Hopefully, this situation will improve as both the editor and tutorials on it improve.

### 2.2.4 Abstractions

Terms are either *primitive* or *abstract*. Primitive terms have fixed pre-defined meanings. Abstract terms or *abstractions* are defined in abstraction objects as being equal to other terms. For example, here is an abstraction for the 'divides' relation on the integers:

```
b | a == ∃c:ℤ. a = b * c
```

I call the process of replacing an instance of the left-hand side of an abstraction by the right-hand side *unfolding* and the reverse process *folding*.

Throughout this thesis, I often refer to abstractions as simply 'definitions' or sometimes 'notational abbreviations'. Abstractions are used not only for Nuprl's object language, but also for example in terms that occur in display-form definitions and in ML code. The graph of the dependencies of abstractions on one another should always be acyclic. However, recursive definitions can be introduced as described in Section 2.2.5.

### 2.2.5 Recursive Definitions

Recursive definitions in Nuprl are coded using the `Y` combinator; since Nuprl's computation language is untyped, the standard $\lambda$-calculus definition of the `Y` combinator can be used:

```
ycomb:
  Y == λf.(λx.f (x x)) (λx.f (x x))
```

Previously, Nuprl had primitive recursion terms for each of the base types (integers, lists and 'simple' recursive types) which were both awkward to use and unnecessary.

In this thesis, I sometimes use the notation *lhs* `==r` *rhs* to introduce a recursive definition. For example, here is the definition of the Fibonacci function:

```
  fib(n) ==r
    if (n =z  0) ∨b  (n =z  1)
    then 1
    else fib(n - 1) + fib(n - 2)
    fi
```

This *lhs* `==r` *rhs* notation is a notational abbreviation for a call to an ML function called `add_rec_def` with *lhs* and *rhs* as arguments. `add_rec_def` takes care of setting the actual abstraction objects for recursive definitions. The abstraction for the `fib(n)` function looks like:

```
fib:
  fib(n)
  == Y
     (λfib,n.
       if (n =z  0) ∨b (n =z  1)
       then 1
       else fib (n - 1) + fib (n - 2)
       fi )
     n
```

Normally, in Nuprl theories, such abstractions are made invisible and the *lhs* `==r` *rhs* ML function calls are retained, both because they document the recursive definitions in a cleaner fashion, and because they inform definition folding and unfolding tactics about the special nature the the definitions. Various tactics and rewrite conversions (see Chapter 4) unfold and fold instances of recursive definitions in a single step so the user is normally never aware of the `Y` combinator representation.

The introduction of recursive definitions using the `Y` combinator is only possible in Nuprl because of a unusual feature of the induction rules that permits the proof of lemmas that characterize when a recursive definition defines a total function. This issue is discussed more in Section 3.5.

### 2.2.6 Theories

The Nuprl V4.1 data-base of definitions and theorems is divided into *theories*. I commonly present listings of parts of theories. Figure 2.1 shows a listing of part of a theory dealing with functions.

A theory contains a sequence of *library objects* or *objects* for short. Object descriptions in theory listings often start with a symbolic character (usually *) and a capital letter. The symbolic character gives the status of the object. * means that the object is complete and has been verified. Other status characters include `#` for incomplete, and `-` for bad in some sense.

The capital letter gives the kind of the object. Kinds of objects include:

`D` for display form definitions.

`C` for comments.

`M` for ML code. ML code in theories is commonly used to introduce theory-specific ML definitions for tactics and rewrite rules, and to provide extra information about definitions to the tactic system.

`T` for a theorem object. A theorem object contains a proposition that has been proven just when the status of the object is *.

`A` for abstractions.

Following the kind of an object is the object's name and the contents of an object. For conciseness, the contents of theorem objects are abbreviated; only the statement of the theorem is shown. Theorem objects also contain proof scripts and extract terms. See Section 2.2.7 for details. Complete listings showing extract terms and proofs can also be generated.

For the purposes of this thesis, it is not necessary to understand the formatting directives given in display form objects and usually I'll not show these. When necessary, I give informal accounts of notational conventions I have chosen to use.

Nuprl sessions always have a library window which allows the user to view segments of the loaded library in a format similar to that described above.

Currently over 30 theories have been defined in Nuprl V4.1. These These are summarised in Section 2.3, and many parts of these theories are described in this thesis..

```
*C tidentity_com
               The type argument of tidentity is never used
               on the right-hand side of the definition,
               but it helps with type inference.
*D tidentity_df               Id{<T:T:*>}== tidentity{}(<T>)
*A tidentity                  Id{T} == λx.x
*T tidentity_wf               ∀A:𝕌. Id{A} ∈ A → A
*M tidentity_ml
               let tidentityC =
                 SimpleMacroC  `tidentityC`
                  ⌈Id{T} x⌉  ⌈x⌉
                 ``tidentity identity``  ;;

               add_AbReduce_conv `tidentity` tidentityC ;;
*D compose_df
               Prec(inop)::Parens ::
                 <f:fun:L>{\\?} o <g:fun:L>
                 == compose{}(<f>; <g>)
*A compose                    f o g == λx.f (g x)
*T compose_wf
               ∀A,B,C:𝕌. ∀f:B → C. ∀g:A → B.  f o g ∈ A → C
*M compose_ml
               let rem_composeC,add_composeC =
                 DoubleMacroC `composeC`
                 (SemiNormC ``compose``) ⌈(f o g) x⌉
                  IdC ⌈f (g x)⌉  ;;

               add_AbReduce_conv `compose` rem_composeC;;
*T comp_assoc
               ∀A,B,C,D:𝕌. ∀f:A → B. ∀g:B → C. ∀h:C → D.
                 h o (g o f) = (h o g) o f ∈ A → D
*T comp_id_l
               ∀A,B:𝕌. ∀f:A → B.  Id{B} o f = f ∈ A → B
*T comp_id_r
               ∀A,B:𝕌. ∀f:A → B.  f o Id{A} = f ∈ A → B
```

Figure 2.1: Partial Listing of Theory on Functions

### 2.2.7 Theorems and Proofs

The heart of the Nuprl system is a piece of code called the *refiner*. Its responsibility is to build proofs by iteratively applying primitive rules of inference from Nuprl's type theory. The correctness of proofs relies almost exclusively on the correctness of the refiner and of the implementation of the primitive rules themselves.

One rarely build proofs by selecting individual rules to use, one-by-one. Instead, one invokes programs written in ML(see Section 2.2.8) called *tactics* that automatically select and sequence appropriate rules. Tactics can be quite sophisticated, but still the correctness of any proof doesn't depend on them, only on the underlying refiner. This tactic paradigm was introduced in the LCF system [GMW79] and has also been adopted in theorem provers such as HOL [GM93] and LEGO [Pol90]. Other interactive theorem provers such as IMPS [FGT92a], and PVS [ORS92] have proof development languages that have many similarities with tactic languages.

Ideally, all the rules the refiner implements should be in some sense straightforward and obviously correct. Most of Nuprl's rules have a fairly simple structure; they are specified by rule objects in a preliminary theory that is always loaded in any Nuprl library. Each rule object contains a pattern for a rule that is instantiated to give rule instances. Nuprl also has some more sophisticated rules which are implemented by Lisp and ML procedures rather than pattern matching. Theoretically, the reasoning accomplished by most of these procedures could also by accomplished by pattern-matching rules, but at a considerable loss of efficiency and convenience.

I describe the tactic collection I created for Nuprl V4.1 in Chapter 3.

A simple proof in Nuprl is that all functions in the type $\mathbb{N} \to \mathbb{N}$ are not enumerable. Figure 2.2 shows a printout of this proof. Here, the function `f` is considered to give a putative enumeration of all the functions. The theorem states that for any `f`, there will always a function `g` that `f` misses out. I generated this proof using Nuprl's proof editor by first entering the goal of the theorem:

$$\vdash \forall \mathtt{f}{:}\mathbb{N} \to \mathbb{N} \to \mathbb{N}.\ \exists \mathtt{g}{:}\mathbb{N} \to \mathbb{N}.\ \forall \mathtt{i}{:}\mathbb{N}.\ \neg(\mathtt{f\ i} = \mathtt{g} \in \mathbb{N} \to \mathbb{N})$$

and then entering each of the tactics after the word `BY`. For brevity, this proof printout only shows at each step of the proof the clauses of the sequent that have changed.

The proof editor generates a window that shows a sequent at some point of a proof, the tactic (if any) run on that sequent, and any subgoals generated. For example, one window onto the above proof is shown in Figure 2.3.

One difference between Nuprl's refiner and that of most other theorem provers is that Nuprl's maintains a proof-tree data-structure. Others just maintain the fringe of the proof tree. Maintaining whole proof trees makes interactive development of proofs considerably easier and makes it simple to go back and experiment with different proof strategies. Also, proof trees serve to document proofs; Nuprl

```
⊢ ∀f:ℕ → ℕ → ℕ. ∃g:ℕ → ℕ. ∀i:ℕ. ¬(f i = g ∈ ℕ → ℕ)
|
BY (Unfold ‘not‘ 0
|    THENM D 0
|    THENM InstConcl [’λn.f n n + 1’] ...’)
↓
1. f: ℕ → ℕ → ℕ
2. i: ℕ
3. f i = (λn.f n n + 1) ∈ ℕ → ℕ
⊢ False
↓
BY (With ’i’ (EqHD 3) THENM Reduce 3 ...a)
↓
3. f i i = f i i + 1 ∈ ℕ
↓
BY Auto
```

Figure 2.2: Example Proof Printout

```
EDIT THM cantor
* top 1
1. f: ℕ → ℕ → ℕ
2. i: ℕ
3. f i = (λn.f n n + 1) ∈ ℕ → ℕ
⊢ False

BY (With ’i’ (EqHD 3) THENM Reduce 3 ...a)

1* 3. f i i = f i i + 1 ∈ ℕ
```

Figure 2.3: Example Proof Window

users frequently study proof techniques and learn tactic behavior by browsing existing proof trees. This explanatory capability of tactics is much harder to take advantage of in theorem provers without proof trees, because there the tendency is to store all the tactics that generate a proof as an unreadable monolithic block. To understand an existing proof in these systems, often the only option the user has is to interactively replay the proof, line by line, and still, the replay will not make the branching structure of proofs clear.

Nuprl V4.1 has an option for maintaining proof-trees at the primitive rule level as well at the tactic level. This is necessary for extraction purposes, although this option is often disabled since the primitive rule trees can be very large. An interesting use of this feature is to maintain proof trees simultaneously at different levels of abstractions. Tactics can easily be set up to generate such multi-level proofs. Then, when a user is trying to follow a proof, she can select the level of explanation as she wants.

Tactics are always expressions in the ML language of type `tactic`, although it is often convenient to use notational abbreviations so that tactics do not always appear to be in the ML syntax. For example, the '...' and '...a' notations at the end of tactics are generated by 'tactic-wrapper' notational abbreviations. It is a Nuprl convention that tactic wrappers with '...' indicate application of variants of Nuprl's `Auto` tactic. These wrappers cause variants of the `Auto` tactic to carry out obvious steps of reasoning and solve trivial subgoals after the execution of the tactics contained in the wrapper. For example, one action of the '...'' wrapper around the tactic:

```
Unfold `not` 0
THENM D 0
THENM InstConcl ['λn.f n n + 1']
```

is to prove that the function `λn.f n n + 1` really has type $\mathbb{N} \longrightarrow \mathbb{N}$ (this is done by using the `SupInf` tactics discussed in Section 3.9).

Currently, proofs are stored in the form of *proof scripts* that contain the tactics necessary for regenerating the proofs. Occasionally, the regeneration of a proof breaks because of minor changes to tactics between the regeneration time and the time the proof was originally created. Usually, it is pretty easy to fix such proofs.

### 2.2.8 The ML Language

The ML language used in Nuprl is a functional language closely-related to the ML of the Edinburgh LCF theorem proving system [GMW79] and is a predecessor of ML used in Cambridge LCF [Pau85] and the SML language [MTH91, Pau91]. As in the LCF system, ML is used for writing all the tactics (ML originally stood for *meta-language*; it was designed for writing tactics). ML is also used as a top-level language for interacting with Nuprl: the user can load and save theories, and invoke term and proof editors from an ML top-loop.

It is assumed that the reader is familiar with some functional programming language. For the purposes of this thesis, the main features of the ML language that the reader needs to know are:

- It has a polymorphic type system. All terms are strongly typed. Types can always be inferred and it is rare that code explicitly contains type annotations. Occasionally, for clarity when documenting functions, I indicate explicitly argument types. For example, I might write an argument as `i:int` indicating that argument `i` has the type `int` of integers.

- The primitive types constructors include `->` for functions, `#` for products, `+` for disjoint sums, and `list` for lists. Atomic types include `bool`, `int` and `tok`. `tok` is a type of tokens. A token is any string of characters, enclosed in '`'s (back-quotes). Function application is denoted by juxtaposition, pairing by an infix '`,`' (comma), and list consing by '`.`' (period). The notation for lists uses `[` and `]` to delimit the list and `;` to separate elements. A sequence of characters delimited by ``` `` ``` on each end is interpreted as a list of tokens. For example ``` ``this is a token list`` ``` is synonymous with ``[`this`,`is`,`a`,`token`,`list`]``.

  Nuprl also has a primitive type `string` of strings. These strings are delimited by doublequotes (`"`) on either side. Strings and tokens are implemented by different data structures in Lisp (strings and symbols respectively) that have different performance characteristics for their elementary operations.

- A kind of abstract data types is supported. In an ML abstract-data-type definition, a representation type is specified along with a set of functions for creating and operating on elements of the representation type. Outside the definition, the only way of manipulating elements of the abstract data type are via the functions set up in the definition. Recursive abstract types can be defined. The types `term` and `proof` are abstract types. Instances of type `term` are either delimited by ''s (forward quotes) or by a $\ulcorner$ on the left and a $\urcorner$ on the right (half way through producing the thesis, I wanted more distinctive tem delimiters so I changed the display form that produces them). The use of the abstract type `proof` for proofs guarantees that only the refiner can build new proofs.

- Concrete type abbreviations are supported. Here, one gives a name for a type pattern and the name is then used for occurrences of the pattern. For example, the type `tactic` is an concrete type abbreviation for `proof -> proof list # validation` and `validation` is an abbreviation for `proof list -> proof`.

- User-defined binary functions can be declared to be infix. Most infix functions are easily recognized because their names use only capital letters. Normal function application binds more strongly than infix function application. The keywords `let` and `letrec` introduce respectively non-recursive and recursive declarations. Two semicolons (`;;`) always terminate declarations and expressions to be evaluated.

- Exceptions can be thrown and caught. This is essential for the behaviour of tacticals such as `ORELSE` and conversionals such as `ORELSEC`.

- Comments are delimited by percent (`%`) characters.

## 2.3 Summary of Libraries

Here is a summary of the main theories I had set up for Nuprl V4.1, by the time that this thesis was completed. There is not space in this thesis to give full listings for all these theories. They should be all publically available in the Nuprl V4.1 release sometime before the end of 1994 and also should be browseable on the World-Wide-Web [BCL+94]. Contact the author for details. Other developments in Nuprl V4.1 include theories on the semantics of an imperative programming language [All94] and constructive real analysis [For93].

- `core_1`: Display forms for primitive terms. Definitions for propositions-as-types correspondence.

- `core_2`: Intuitionistic propositional and predicate logic.

- `bool_1`: The boolean type and boolean operators. Demorgan laws and simplification theorems and tactics. Tactics for case splitting on the value of boolean expressions, especially when they are arguments of `if then else` terms.

- `fun_1`: Identity function and function composition. Standard predicates on functions (injective, surjective, bijective) and theorems relating them.

- `well_fnd`: Basic theory of well-founded relations. Definition of tactics for induction on the rank of some expression.

- `int_1`: Definition of standard subsets of the integers. Theorems and tactics for linear and complete induction over subsets.

- `list_1`: Common list functions that do not involve equality testing (for example `map, hd, tl, length`).

- `num_thy_1`: Divisibility in $\mathbb{Z}$. GCDs, coprimality, Chinese remainder theorem. Does not involve any iterated operations over integers.

- `rat_1`: Rational numbers and arithmetic operations on rationals. This theory has been much extended by Forester in proving the Intermediate Value Theorem.

- `gen_algebra_1`: Common algebraic predicates (for example, associativity commutativity). Order and equivalence relations.

- `sets_1`: Class of discrete types (a type is *discrete* when its equality relation is decidable) and types with decidable order relations.

- `groups_1`: Classes for groups and monoids. Abelian and discrete variants. Iterated operations over a monoid. Homomorphisms. Normal subgroups and quotient groups.

- `rings_1`: Classes of rings, integral domains and fields. Ideals and quotient rings. Lifting of iterated operations to sum and product operations.

- `perms_1`: The permutation group on any type. Permutations on a finite type. Building permutations from swaps. Invariance of iterated operations over abelian monoids.

- `perms_2`: The permutation relation on lists.

- `list_2`: List functions over discrete sets and monoids (for example, `member`, `reduce`). Gives many monoid-related and permutation-related lemmas.

- `factor_1`: Conditions for existence and uniqueness of factorizations in cancellation monoids.

- `mset`: Multisets built from lists using quotient type. Show that have a free abelian monoid.

- `algebras_1`: Classes of modules and algebras.

- `polynomials_1`: Class definitions for free abelian monoid, free monoid algebra and polynomial algebra. Development of multivariate polynomial implementation.

The theories I have developed contain roughly 500 definitions (not counting those that support ML and the editor), and 1300 theorems (not counting the primary well-formedness theorem that nearly always accompanies each definition). When assessing these numbers, remember that what counts as a theorem or definition in formal mathematics is often much less than what counts in the normal practice of mathematics. By studying this thesis, the reader should gain some idea of the granularity of Nuprl theories and the level of detail.

# Chapter 3

# Tactics

## 3.1 Introduction

A variety of algorithms have been implemented in Nuprl's 'meta-language' ML to automate parts of proof development. These algorithms are encapsulated into these procedures that are called *tactics* which users select amongst and apply when they want to construct formal proofs in Nuprl.

In this chapter, I survey the tactic collection that I have created for Nuprl V4.1 and describe a few of the algorithms that underlie these tactics. Where appropriate, I point out the relationship between these tactics and those found in Nuprl V3, as well as in other theorem-proving systems. Note that the rewrite tactics are described separately in Chapter 4. For more details on the tactics, consult the Nuprl V4.1 Reference Manual [Jac94b].

I have ordered the sections of this chapter roughly according the the complexity of the tactics that they describe. The last section of the chapter, Section 3.10, covers the matching routines that are used in several of the tactic families described in this chapter, as well as extensively in the the rewrite tactics.

In total, over 3000 functions have been defined in over 20,000 lines of ML code (not counting blank lines and comments). Fortunately, the average user need only be familiar with perhaps 20 or 30 of these functions in order to complete most proofs.

## 3.2 Well-formedness Reasoning

In practice, most well-formedness goals can be solved automatically by a set of heuristics built into Nuprl's `Auto` tactic. The advantage of this approach over a syntactic one is that one can easily experiment with extensions to the type regimen that one commonly works in. For example, we often work in Nuprl with terms that have several alternative typings and `Auto` selects the appropriate typings in different instances.

A serious disadvantage with Nuprl's approach is that it is tremendously slow compared to syntactic methods; frequently, 90% or more of tactic execution time is spend in type checking. Efforts to streamline things with caching schemes have had some success so far, but it's not clear whether one can ever approach the efficiency of syntactic methods.

Another disadvantage is that showing the initial well-formedness of new definitions can be a significant burden to the user. When proving well-formedness lemmas, one is severly restricted in what rewriting and chaining tactics one can make use of. Part of the work I describe in Section 3.5 below has been in developing methodologies for making the proof of these lemmas more straightforward.

In presenting the behaviour of tactics in this chapter, I often suppress the well-formedness subgoals generated. `Auto` is almost always run on well-formedness subgoals created by tactics, so such goals are rarely seen in practice.

## 3.3 Decomposition

The decomposition tactics provide access to the decomposition rules of Nuprl's logic — the rules which as described in Section 2.1.4 correspond to the left and right introduction rules of a Gentzen L-style sequent calculus for predicate logic. The simplest decomposition tactic is the tactic `D` of type `int -> tactic`. The tactic `D` $i$ decomposes the outermost constructor in clause $i$. For example, remembering that the conclusion is considered to be 'clause 0', here is the `D` tactic decomposing an implication in the conclusion:

$\ldots \;\vdash\; P \Rightarrow Q$

`BY D 0`

`main:` $\ldots, P \;\vdash\; Q$

This tactic example is read as indicating that if the tactic `D 0` is run on a goal $\ldots \;\vdash\; P \Rightarrow Q$, then the subgoal $\ldots, P \;\vdash\; Q$ is generated. The `main` indicates a *goal label* on this subgoal. Some tactics generate labels on subgoals and other tactics discriminate on these labels. Goal labels are visible to the user when doing interactive proof and often give hints as to where otherwise mysterious subgoals have originated from. The `D` tactic above also generates a second well-formedness subgoal with label `wf`, but as mentioned in Section 3.2, I'll frequently not show such goals.

The `D` tactic works on the standard logical connectives and quantifiers as well as on hypothesis types. Sometimes, an additional argument needs to be specified. For example:

$\ldots,\ j.\forall x{:}A.P_x, \ldots\ \vdash\ \ldots$

`BY With` $a$ (`D` $j$)

`main:` $\ldots,\ j.P_a, \ldots\ \vdash\ \ldots$
`wf:` $\ldots,\ j.\forall x{:}A.P_x, \ldots\ \vdash\ a \in A$

I use the $j$. here before the $\forall x{:}A.P_x$ to indicate that this is the $j$th hypothesis.

The `D` tactic usually does nothing more than select an appropriate primitive rule to do the decomposition and take care of perhaps unfolding an abstraction or two before applying the primitive rule. The user can specify extensions to `D` for cases when some other course of action would be most useful.

Several decomposition tactics do repeated decomposition. For example, the `RepD` tactic repeatedly decomposes any $\forall$ and $\Rightarrow$ propositions in the conclusion and $\wedge$ hypotheses. The `GenExRepD` tactic in addition decomposes $\exists$ and $\vee$ hypotheses and $\wedge$ conclusions. The `ProveProp` tactic repeatedly and exhaustively applies the `D` tactic everywhere, backtracking when necessary. As the name implies, it is good for proving goals that involve just propositional reasoning. A variant on `ProveProp` allows alternative tactics to be run on the leaves of the search tree that `ProveProp` generates.

## 3.4 Member and Equality Decomposition

A family of decomposition tactics work on the arguments of membership ($t \in T$) and equality ($t = t' \in T$) terms. `MemCD` works on the arguments of membership terms in the conclusion. For example:

$\ldots\ \vdash\ < a, b > \in x{:}A \times B_x$

`BY MemCD`

`subterm1:` $\ldots\ \vdash\ a \in A$
`subterm2:` $\ldots\ \vdash\ b \in B_a$
`wf:` $\ldots,\ x{:}A\ \vdash\ B_x \in \mathbb{U}$

`MemCD` tries to use *well-formedness lemmas* or *wf-lemmas* for its reasoning. For example, a wf-lemma for rational plus function is:

$\vdash\ \forall a{:}\mathbb{Q}.\ \forall b{:}\mathbb{Q}.\ a +_q b \in \mathbb{Q}$

A more complicated wf-lemma is for an operator $\boldsymbol{\Sigma}$ for summing up an integer-indexed sequence of elements of a monoid g:

$$
\begin{array}{ll}
\vdash & \forall g{:}\mathsf{Monoid} \\
& \forall m{:}\mathbb{Z}\ \forall n{:}\{m\ldots\} \\
& \forall E{:}\{m\ldots n-1\} \to |g| \\
& (\Sigma(g)\ m \le k < n.\ E[k]) \ \in \ |g|
\end{array}
$$

An example use of this wf-lemma is:

$$\vdash\ (\Sigma(\langle\mathbb{Z},+,0\rangle)\ 1 \le i < 10.\ i * i) \ \in\ \mathbb{Z}$$

BY MemCD

$$
\begin{array}{ll}
\texttt{subterm1:} & \vdash\ \langle\mathbb{Z},+,0\rangle \in \mathsf{Monoid} \\
\texttt{subterm2:} & \vdash\ 1 \in \mathbb{Z} \\
\texttt{subterm3:} & \vdash\ 10 \in \{1\ldots\} \\
\texttt{subterm4:} & i{:}\{1\ldots 10\}\ \vdash\ i * i \in |\langle\mathbb{Z},+,0\rangle|
\end{array}
$$

This example illustrates a couple of subtleties of the operation of MemCD. Firstly, it handles smoothly operators with binding. Secondly, types often have to be juggled to make them match up correctly. This is taken care of by the Inclusion tactic that MemCD invokes. Here, the Inclusion tactic recognizes that that the type expressions $|\langle\mathbb{Z},+,0\rangle|$ and $\mathbb{Z}$ are the same thing. MemCD is at the heart of the Nuprl tactics such as Auto that are used to prove well-formedness subgoals.

Closely related to MemCD is EqCD which used for congruence reasoning. It decomposes equalities in the conclusion of terms with the same outermost term constructor into equalities between the corresponding subterms. For example:

$$\ldots\ \vdash\ p +_q 1. = q +_q 1. \in \mathbb{Q}$$

BY EqCD

$$
\begin{array}{ll}
\texttt{subterm1:} & \vdash\ p = q \in \mathbb{Q} \\
\texttt{subterm2:} & \vdash\ 1. = 1. \in \mathbb{Q}
\end{array}
$$

Wf-lemmas implicitly contain information about congruence properties of terms, and as in the above example, EqCD is able to use this information.

Other decomposition tactics include MemHD and EqHD for decomposing terms in hypothesis memberships and equalities, and MemTypeCD and EqTypeHD for decomposing just types in membership terms and equalities. The latter are applicable to set types, quotient types, recursive types and type abstractions built using these types.

## 3.5 Induction

Induction is commonly used over well-founded orders. I desgined theorems and tactics for showing that successively richer orders are well-founded and doing induction over these orders.

One problem with Nuprl's type theory is that when proving well-formedness lemmas, it is very easy to generate well-formedness subgoals that are as hard to solve as the one being proved. This happens whenever one tries to use an induction lemma as a step in a proof a well-formedness lemma. Fortunately, if the pattern of primitive rules used in proving the induction lemma and previous lemmas is duplicated, the induction can be done without creating these unwanted subgoals. To simplify matters, I added in a proof analogy mechanism whereby a tactic executes by copying a portion of an existing proof, but perhaps uses different specific parameters.

An example of such a tactic is is the `RankInd` rank induction tactic. Figure 3.1 illustrates its use in a proof of the well-formedness of a `gcd` (greatest-common-divisor) function. The invocation pattern for `RankInd` is `RankInd` $\rho$ $R$ $Tac$ $i$,

```
1. b: ℤ
⊢ ∀a:ℤ. gcd(a;b) ∈ ℤ
|
BY (OnVar ‘b’ (RankInd ’λi.|i|’ ’ℕ’ CompNatInd)
|   THENM D 0 ...a)
|
2. ∀b1:ℤ. |b1| < |b| ⇒ (∀a:ℤ. gcd(a;b1) ∈ ℤ)
3. a: ℤ
⊢ gcd(a;b) ∈ ℤ
```

Figure 3.1: Use of `RankInd` Induction Tactic

where $i$ indicates the hypothesis declaring the variable that the induction is over, $\rho$ is the rank function, $R$ is the range of the rank function, and `Tac` is a tactic for complete induction over the range type $R$. The `OnVar ‘b’` is a tactical for applying tactics that apply to hypotheses. It allows reference to declarations by name rather than by number.

## 3.6 Chaining

Forward and backward chaining involve treating a component of a universal formula as a derived rule of inference. I consider a *universal* formula to be one generated by the grammar

$$\begin{array}{rcl} P & ::= & \forall x{:}A.\ P \mid Q \Rightarrow P \mid P \Leftarrow Q \\ & \mid & P \wedge P \mid P \iff P \\ & \mid & R \end{array}$$

where $A$ is a type and $R$ is a propositional term not of the above form. They are sometimes called positive definite formulae or horn clauses. I call the proposition $R$, a *consequent* and each $Q$, an *antecedent*. I call the formulae generated by this grammar without the $\wedge$ and $\iff$ connectives, *simple universal* formulae. A universal formula is logically equivalent to one or more simple universal formulae, one for each consequent. I consider these simple universal formulae to be the *components* of a universal formulae.

Universal formulae used for chaining appear either as a hypotheses in proofs one is working on or as previously proven lemmas in the Nuprl library.

Backward chaining involves matching the conclusion of a sequent against the consequent of a universal formula. The antecedents of the universal formula, instantiated using the substitution resulting from the match, then become new subgoals. The tactics for 1-step backchaining are `BackThruLemma` and `BackThruHyp`, often abbreviated to `BLemma` and `BHyp` respectively. An example use of `BLemma` is:

```
i:ℤ⊢3 * i ≤ 3 * (i + 1)

BY BLemma ‘mul_preserves_le‘

    main:  ⊢ i ≤ (i + 1)
```

where `mul_preserves_le` is the lemma:

```
⊢ ∀a:ℤ. ∀b:ℤ. ∀n:ℕ. a ≤ b ⇒ n * a ≤ n * b
```

There are several tactics for repeating backchaining steps using lemmas and hypotheses. These tactics allow optionally for backtracking and so can be used for prolog-style proof search. They also have some basic loop detection built in to prevent some kinds of unbounded backchaining (for example, when backchaining through commutativity lemmas).

Forward chaining involves matching hypotheses of a sequent against antecedents of a universal formula. The consequent of the universal formula, instantiated using the substitution resulting from the match, then becomes a new hypothesis. The tactics for 1 step forward chaining are `FwdThruLemma` and `FwdThruHyp`, often abbreviated to `FLemma` and `FHyp` respectively. An example use of `FLemma` is:

```
1.i:ℤ, 2.  3 * i ≤ 3 * (i + 1) ⊢...

BY FLemma ‘mul_cancel_in_le‘ [2]

    main:  3.  i ≤ i + 1 ⊢ ...
```

where `mul_cancel_in_le` is the lemma:

$\vdash$ `∀a:ℤ. ∀b:ℤ. ∀n:ℕ+. n * a ≤ n * b ⇒ a ≤ b`

The forward chaining tactics take a list of numbers of hypotheses to try to match against the lemma antecedents (the `[2]` in the example above).

# 3.7 Constructive and Classical Reasoning

I review here the mechanisms I've put in place for assisting constructive reasoning. In the last section, I discuss how one can reason classically.

## 3.7.1 Constructive Reasoning

The intrinsic constructivity of Nuprl's logic manifests itself in three main ways with the tactics:

1. For any proposition $P$, the goal $P \vee \neg P$ is not in general provable.

2. When applying the `D` tactic to a hypothesis that has a set term outermost, the predicate part of the set term becomes a *hidden* hypothesis. For example:

   $\ldots i.x{:}\{y{:}T|P_y\},\ \ldots\ \vdash\ \ldots$

   `BY D` $i$

   $\ldots i.x{:}T,\ [i+1].P_x,\ \ldots\ \vdash\ \ldots$

   Here, the [] surrounding the hypothesis number $i+1$ indicate that this hypothesis is hidden. A hidden hypothesis is not immediately usable though there are ways in which it might become usable later in a proof. The need for hiding is a consequence of the constructivity of Nuprl's type theory. Without hiding, the rule invoked by the `D` tactic above would be unsound.

3. To prove a conclusion $C$ it is not in general legitimate to assume the negation of $C$ and prove falsity; The goal $\ldots \vdash C$ cannot in general be refined to the goal $\ldots \neg C \vdash \mathsf{False}$.

Tactics to simplify dealing with these issues are described in the next three sections.

### 3.7.2 Decidability

Many useful instances of $P \vee \neg P$ are provable constructively and the `ProveDecidable` tactic is set up to construct these proofs in a systematic way. To discuss it, I first introduce the `decidable` abstraction:

$$\mathsf{Dec}(P) \quad =_{def} \quad P \vee \neg P$$

It turns out that the property $\mathsf{Dec}(P)$ can be inferred for many $P$ from knowing that $\mathsf{Dec}(Q)$ for the immediate subterms $Q$ of $P$. For example, if $\vdash \mathsf{Dec}(Q_i)$ for $i = 1, 2$, then $\vdash \mathsf{Dec}(Q_1 \vee Q_2)$ and $\vdash \mathsf{Dec}(Q_1 \iff Q_2)$. Decidability over product types can also be inferred from decidability over the component types. This can be stated by the lemma:

$$\begin{aligned}&\vdash \forall A, B{:}\mathbb{U}.\\ &\quad (\forall a, a'{:}A.\ \mathsf{Dec}(a = a' \in A))\\ &\quad \Rightarrow (\forall b, b'{:}B.\ \mathsf{Dec}(b = b' \in B))\\ &\quad \Rightarrow (\forall c, c'{:}A \times B.\ \mathsf{Dec}(c = c' \in A \times B))\end{aligned}$$

`ProveDecidable` takes advantage of this property of decidability, and attempts to prove goals of the form

$$\ldots \vdash \mathsf{Dec}(P)$$

by backchaining with a user-extensible set of *decidability* lemmas. `ProveDecidable` is usually invoked via the `Decide` tactic which is used to case-split on whether a proposition $Q$ is true or false. It generates two main subgoals; one with the new assumption $Q$ and the other with the new assumption $\neg Q$. It also generates a subgoal $\ldots \vdash \mathsf{Dec}(Q)$ and runs the `ProveDecidable` tactic on this subgoal.

### 3.7.3 Squash Stability and Hidden Hypotheses

*Squash stability* is defined in the Nuprl library as:

$$\mathsf{SqStable}(P) \quad =_{def} \quad \downarrow P \Rightarrow P$$

The proposition $\downarrow P$ (read 'squash P') is considered true exactly when $P$ is true. However, $P$'s computational content when true can be arbitrary whereas $\downarrow P$'s computational content when true can only be the trivial constant term $\cdot$ (read 'it') that inhabits the unit type. ($\downarrow P$ is defined as $\{x{:}\mathsf{Unit}|P\}$ where $x$ does not occur free in $P$.)

Informally, a proposition is *squash stable* if it is possible to figure out what its computational content is, given that it is known that some computational content exists (in the classical sense). The *computational content* of a proposition is some term that inhabits the proposition when it is considered as a type.

Squash stability is a useful concept because it characterizes exactly when a hidden hypothesis can be unhidden. Specifically, a hidden hypothesis $P$ in a sequent $\sigma$ can be unhidden if one of two conditions are met:

1. The proposition $P$ is squash stable.

2. The conclusion of $\sigma$ is squash stable.

As with decidability, it turns out that the property $\mathsf{SqStable}(P)$ can be inferred for many $P$ from knowing that $\mathsf{SqStable}(Q)$ for the immediate subterms $Q$ of $P$. It is also true that $\mathsf{Dec}(P) \Rightarrow \mathsf{SqStable}(P)$ for any $P$. The tactic `ProveSqStable` takes advantage of these facts and attempts to prove goals of the form

$$\ldots \vdash \mathsf{SqStable}(P)$$

by backchaining with a user-extensible list of lemmas about squash stability and also resorting to checking whether $P$ is decidable. Various tactic that unhide hypotheses create $\mathsf{SqStable}$ subgoals that are proven using the `ProveSqStable` tactic.

### 3.7.4 Stability and Negating the Conclusion

Squash stability is closely related to the stability predicate in constructive logic; a proposition $P$ is *stable* if $\neg\neg P \Rightarrow P$. As explained at the end of the next section, the two predicates can be considered equivalent.

The refinement step of proving a conclusion $C$ by assuming the negation $\neg C$ and proving a contradiction is only constructively valid when $C$ is stable. The `NegateConcl` tactic carries out this step and checks the stability of $C$ using the `ProveSqStable` tactic.

### 3.7.5 Classical Reasoning

To reason non-constructively, one needs to have as an explicit hypothesis the excluded middle proposition

$$\forall P{:}\mathbb{P}.\, P \vee \neg P$$

(or at least some instances of it). The `xmiddle` abstraction provides an abbreviation for this:

$$\texttt{xmiddle:}\ \ \mathsf{XM}\ \ =_{def}\ \ \forall P{:}\mathbb{P}.\, \mathsf{Dec}(P)$$

The `Decide`, `NegateConcl` and hypothesis unhiding tactics all recognize whenever the `xmiddle` abstraction occurs as some hypothesis, and in this case never fail.

A non-constructive theorem is stated by using `xmiddle` as a precondition of the theorem. Such theorems are then usually of form $\vdash \mathsf{XM} \Rightarrow P$.

There are two common cases when in proving a part of a constructive theorem, classical reasoning becomes allowable:

1. If the conclusion is the *squashed exists* term $\downarrow \exists x{:}T.\ P_x$ and the existential is about to be instantiated using the tactic `With` $t$ `(D 0)`. Squashed exists is defined as:

   $$\texttt{sq\_exists:}\ \ \downarrow \exists x{:}T.\ P[x] \quad =_{def} \quad \{x{:}T \mid P[x]\}$$

   When the tactic `AddXM` is used first, the refinement has form:

   $\ldots\ \vdash\ \downarrow \exists x{:}T.\ P_x$

   `BY AddXM 1 THEN With` $t$ `(D 0)`

   | | |
   |---|---|
   | `wf` | $\mathsf{XM}\ldots\ \vdash\ t \in T$ |
   | `main` | $\mathsf{XM}\ldots\ \vdash\ P_t$ |
   | `wf` | $\mathsf{XM}\ldots x{:}T\ \vdash\ P_x \in \mathbb{P}$ |

   `AddXM 1` adds the hypothesis XM as a hidden hypothesis, so there are no soundness problems here. XM becomes unhidden in the first and third subgoals since here the conclusion is recognized as being trivially squash stable. XM becomes unhidden in the second subgoal since from here on, any computational content in the proof cannot contribute to the computational content of the original goal of the theorem.

2. If the conclusion is squash stable. When `AddXM 1` is run, the proposition XM is added as a hidden first hypothesis and one of the unhiding tactics unhides it.

The `AddXM` tactic assumes that the proposition

$\vdash\downarrow (\forall P{:}\mathbb{P}.\ P \vee \neg P)$

is true; that is, the corresponding type is inhabited. This is not true according to the set-theoretic semantics that Allen gave, but is true according to the model of Howe in which the computation language is enriched with non-computable oracles. The outermost squash operator ensures that the oracular inhabitants of the above proposition never make their way into any other extracts of theorems and so upset Nuprl's constructivity. When $\downarrow (\forall P{:}\mathbb{P}.\ P \vee \neg P)$ is assumed true, stability and squash stability are equivalent notions. The `NegateConcl` tactic described in the previous section takes advantage of this fact.

## 3.8 Relational Reasoning

For the rewrite package described in Chapter 4, I established conventions for building libraries so that tactics would have easy access to information about various characteristics of binary relations (see Section 4.3).

I took advantage of this accessibility in the design of a tactic `RelRST` [1] that automates solving goals that depend on these characteristics. The heart of this tactic is a routine that builds a directed graph based on the binary relations in a sequent and finds shortest paths in the graph. Extensions were made to this routine to allow it to handle strict order relations and relations with differing strengths.

I ended up adding features to `RelRST` so that it also could take advantage of antisymmetry, irreflexivity and linearity properties of relations.

`RelRST` generalizes the `Eq` tactic in previous versions of Nuprl that only handled such reasoning with the equality relation of Nuprl's type theory.

Here are a couple of examples of `RelRST`'s use from a theory of divisibility over the integers:

```
1. a: ℤ
2. a': ℤ
3. b: ℤ
4. b': ℤ
5. ...
6. a' | a
7. b | b'
8. ...
9. a | b
⊢ a' | b'
|
BY (RelRST ...)
```

and

```
1. a: ℤ
2. b: ℤ
3. y1: ℤ
4. ...
5. gcd(a;b) = y1
6. y2: ℤ
7. ...
8. gcd(b;a) = y2
9. ...
10. y1 ~ y2
⊢ gcd(a;b) ~ gcd(b;a)
```

[1] standing for Rel(ation) R(eflexivity), S(ymmetry) and T(ransitivity)

```
|
BY (RelRST ...)
```

Here, I have elided hypotheses that were not required by `RelRST` to solve the goals. The $\sim$ relation is the *associated* relation and `gcd(a;b)` is a function that computes the greatest common divisor of `a` and `b`. The second example illustrates how `RelRST` is able to cope with relations of differing strengths.

## 3.9 Arithmetic Reasoning

The integers and subsets of the integers such as the naturals are amongst the most common data-types in theorem proving. Consequently, goals involving arithmetic reasoning come up frequently. These goals can be very tedious to solve by manual application of sets of lemmas derived from Peano-like axioms and recursive definitions of the arithmetic functions, despite the fact that the goals are often very obvious to the theorem prover user.

There are several standard algorithms for solving some kinds of arithmetic problems. For example, for deciding the satisfiability of conjunctions of inequalities over linear rational expressions, there is Fourier's technique of variable elimination that has been known for over a century [Chv83]. This problem can easily be rephrased as a linear programming problem for which the commonly used method in the operations research community is the simplex algorithm [Chv83].

Nuprl inherited from the PL-CV system built at Cornell a procedure called *arith* for solving arithmetic problems over the integers [CJE82, C$^+$86]. Roughly speaking, arith tries to solve a goal by putting arithmetic expressions into a normal form and then applying congruence closure. It also has some basic capabilities for solving inequalities.

However, arith cannot solve general sets of linear inequalities over the integers though such problems are abundant when for example doing array bounds checking. Solving linear inequalities over the integers is a strictly harder problem; polynomial time algorithms are known for the solving linear inequalities over the rationals, but integer linear programming is NP complete. In practice in theorem proving, simple adaptations of methods over the rationals have worked well for the integers.

I chose to implement in Nuprl a tactic that uses the *Sup-Inf* method for solving integer inequalities [Ble75]. The basic algorithm considers a conjunction of inequalities $0 \leq e_1 \wedge \ldots \wedge 0 \leq e_p$ where the $e_i$ are linear expressions over the rationals in variables $x_1 \ldots x_n$ and determines whether or not there exists an assignment of values to the $x_j$ that satisfies the conjunction. The algorithm works by determining upper and lower bounds for each of the variables in turn — hence the name 'sup-inf'. The bound calculations are always conservative, so that if some upper bound is strictly below some lower bound, then the conjunction is unsatisfiable.

Shostak [Sho77] showed that the calculated bounds are the best possible, and hence that the algorithm is complete for the rationals. He proposed a simple modification that made the algorithm return an explicit satisfying assignment when the conjunction is satisfiable.

When used over the integers, the Sup-Inf algorithm is sound, but not complete; if there is no satisfying assignment over the rationals, then there is also none over the integers. However, there are cases when the algorithm finds a rational-valued satisfying assignment even though none exists that is integer valued. There are standard techniques for restoring completeness, but it has been both Shostak's and our experiences to date that examples for which the algorithm is incomplete are rare in practice.

The procedure I implemented currently does the following:

1. Takes a goal $g$ and extracts a logical expression $P$ built from the logical connectives $\wedge, \vee, \neg$, the order relations on the integers $\leq$ and $<$, and the equality relation $=$ on the integers, such that if $\neg P$ is not satisfiable, then the goal $g$ is true. If the goal has the form

   $$x_1, \ldots x_n{:}\mathbb{Z},\ r_1, \ldots r_k \vdash r_0$$

   where the $r_i$ are all instances of the $\leq, <, =$ relations over the integers involving expressions over the integer variables $x_1, \ldots, x_n$, then $\neg P$ has form

   $$r_1 \wedge \ldots \wedge r_k \wedge \neg r_0.$$

2. The expression $\neg P$ is put into disjunctive normal form. Occurrences of $=$ and $<$ relations are eliminated in favour of $\leq$. Where possible, $=$'s are eliminated by substitution rather than splitting into inequalities.

3. The left-hand argument of each $\leq$ is moved to right-hand side and the integer expressions are put into a sum of products normal form. Each product has any constant coefficient brought out to the left of the product.

4. Each distinct non-linear expression is generalized to a new rational variable. These non-linear expressions might involve $*$ and $\div$, as well as integer-valued functions (for example, the list length function). The arithmetic expressions are now all linear.

5. Each disjunct is augmented with extra arithmetic information suitably normalized that comes from various sources including:

   (a) typing of variables and generalized non-linear expressions. If variable $i$ has type $\{j \ldots\}$, then $j \leq i$ can be added.

(b) arithmetic property lemmas. An example is a lemma stating that the length of two lists appended is the sum of the lengths of each list.

This augmentation is in general a recursive procedure; the inferred arithmetic propositions can themselves contain variables and non-linear expressions about which further information can be inferred.

6. The Sup-Inf algorithm is run on each disjunct. If none is satisfiable, then the original goal is true. If a satisfying assignment is found, then it is returned to the user as a counter-example.

7. When no disjunct is satisfiable, the procedure creates several well-formedness subgoals. Some of these check the well-formedness of the arithmetic expressions in the conclusion of the original goal $g$. Others check that the arithmetic property lemmas can be instantiated as the procedure assumed they could be.

The inference of arithmetic properties from typing and from property lemmas greatly increases the procedure's usefulness.

Unlike most other tactics, but like the `arith` rule which `SupInf` largely supercedes, `SupInf`'s inferences are not refined down to primitive rule level, so Nuprl's soundness now depends on the soundness of a core part of `SupInf`'s implementation. Efforts are now underway to see if the functions used in the `SupInf` tactic can be formally verified in Nuprl.

I give a couple of examples of uses of the `SupInf` tactic. It is able to prove the goal:

```
1. x: ℤ
2. y: ℤ
3. z: ℤ
4. 2 * y + 3 ≤ 5 * z
5. z ≤ x - y
6. 3 * x ≤ 5
⊢ 2 * y ≤ 3
```

but on:

```
1. x: ℤ
2. y: ℤ
3. 3 * x ≥ y
4. y ≥ 2
⊢ x + y > 3
```

finds the counterexample `x`= 1 and `y`= 2. Examples of arithmetic property lemmas are:

```
⊢ ∀ i:ℤ. ∀ j:ℤ. i ≥ 0 ⇒ j > 0 ⇒ 0 ≤ i rem j < j
```

where `rem` is the remainder function and:

```
⊢ ∀A:𝕌. ∀as:A list. ∀n:ℕ|as|. (|nth_tl(n;as)| = |as| - n)
```

where `nth_tl(n;as)` takes the `n`th tail of list `as`, `|·|` is the list length function and `ℕ|as|.` is an abbreviation for the integer segment `{0...|as|-1}`. The latter lemma is invoked when `SupInf` proves the goal:

```
1. T: 𝕌
2. as: T List
3. m: ℕ
4. n: ℕ
5. |as| ≤ m + n
⊢ |nth_tl(n;as)| ≤ m
```

# 3.10 Matching

## 3.10.1 Second-Order Matching and Substitution

Matching routines are at the heart of several tactics such as the rewriting tactics (see Chapter 4) and the chaining tactics (see Section 3.6).

Nuprl V4.1's matching routine is based on a second-order restriction [HL78] of Huet's higher-order unification algorithm [Hue75]. This second-order routine handles patterns with bound variables, in contrast to Nuprl V3's first-order routine which did not. The advantage of using this second-order algorithm, rather than the full higher-order algorithm, is that it is much more controlled; unique most-general substitions (or unifiers) exist with it. With Huet's full algorithm, a potentially infinite lazy stream of unifiers are generated, even though nearly always, all but the first one are not needed. I have found that second-order matching is adequate nearly all the time so far, Miller reports similar positive experiences when working in the system $\lambda$-prolog with a similar second-order unification algorithm that he calls $\beta_0$-*unification* [Mil91], and I know that Paulson and Nipkow have had success with something similar in Isabelle [Pau90].

To discuss second-order matching, I first introduce the notions of second-order terms and second-order substitutions. *Second-order terms* are a generalization of terms. They can be thought of as 'terms with holes', terms with zero or more subtrees missing. It is both convenient to fill the holes in a second-order term with distinct variables, so forming a first-order term. The notation for a second-order term is then $w_1, \ldots, w_n.t$ where the occurrences of the variables $w_i$ in $t$ indicate the holes.

A *second-order variable* is a new kind of variable that in addition to a name has a natural-number *arity*. A *instance* of second-order variable $v$ of arity $n$ has form: $v[a_1; \ldots; a_n]$, where $a_1, \ldots, a_n$ are its arguments.

A *second-order substitution* is a list of *second-order bindings*, pairs of second-order variables and the second-order terms they are bound to. The result of applying the binding $[v \mapsto w_1, \ldots, w_n.t]$ to the variable instance $v[a_1; \ldots; a_n]$, is the term $t[a_1, \ldots, a_n / w_1, \ldots, w_n]$, where the notation $\cdot[\cdot/\cdot]$ denotes capture-avoiding first-order substitution.

Second-order substitution is useful for instantiating pattern terms involving binding structure. For example the second-order substitution $[P \mapsto i.i \geq 0]$ applied to the pattern $\forall x{:}\mathbb{N}.\ P[x]$ yields the instance $\forall x{:}\mathbb{N}.\ x \geq 0$.

*Second-order matching* involves taking a pattern term $p$ and an instance term $i$ and determining whether there is a second-order substitution $\theta$ such that $\theta p = i$.

The matching algorithm implemented requires that least one occurrence of every second-order variable in the pattern have as arguments only variables and further that all these variables be bound in the pattern. The pattern term $\forall x{:}\mathbb{N}.\ P[x]$ is an example that satisfies this restriction.

Second-order variable instances are allowed in abstraction definitions, but not in theorems and their proofs. There, second-order variable instances are simulated by applications of first-order variables, and second-order substitution is simulated by a combination of first-order substitution and beta-reduction.

### 3.10.2 Match Extension

I used a match extension routine, much as described by Howe [How88a]; instantiating bindings for universally quantified expressions cannot always be figured out solely from initial matches. in some cases one has to go through an iterative process of matching types of existing bindings against the types of the corresponding variables in the universally quantified expressions. This need for match extension occurs frequently when reasoning with polymorphic functions. For example, consider this lemma about collapsing two `map` functions:

```
⊢ ∀A,B,C:𝕌
   ∀f:A → B
    ∀g:B → C
     ∀as:A List. map(g;map(f;as)) = map(g o f;as) ∈ C List
```

If this were used as a left-to-right rewrite rule, then Nuprl would try using its matching function to find expressions that matched the pattern `map(g;map(f;as))`. Assume a match is found and the matching routine generates bindings for `g`, `f` and `as`. The match extension process then finds bindings for `A`,`B` and `C` as explained above.

### 3.10.3 Universe Polymorphism

It turned out to be non-trivial to modify the matching procedures to accomodate the level expressions found in universe terms and abstractions involving universe terms. The problem was that I needed the matching routines to in general solving sets of inequalities and that one couldn't begin to solve these inequalities until any match extension had been completed. The main matching routines had all to be modified to propagate and collect these inequalities.

Solving the inequalities themselves was fairly straightforward; I came up with an algorithm that always finds a match if there is one, and further which finds an equality match if possible [Jac94c].

# Chapter 4

# Rewriting

## 4.1 Introduction

Rewriting [DJ90], the process of using equations as transformational rules, is a common technique in theorem-proving. In resolution theorem provers, rewriting is often accomplished using the demodulation and paramodulation rules [WOEB84]. Most interactive theorem provers have some kind of rewriting facility. For example, the NQTHM prover uses rewriting heavily for simplification and for application of inductive hypotheses [BM88a]. Rewriting is also common in computer algebra systems. For example, Mathematica [Wol91] allows users to phrase transformation and simplification strategies as sets of rewrite rules.

The repeated exhaustive application of a set of rewrite rules at all nodes of a term tree can often be an inefficient method for achieving simplification, though it is a topic commonly studied in the rewriting literature. Often rewrite rules have structure that can be taken advantage of. For example, a set of simplifying rewrite rules might do all the work they might ever do in a single pass over a term tree starting from the leaves and ending at the root. Other problems with exhaustive application include the difficulty of obtaining guarantees that a set of rules doesn't cycle and of figuring out whether to turn equations into left-to-right rewrite rules or or right-to-left rules. An example of the fruits of careful analysis of a rewriting problem can be found in the work of Bundy and others; starting with a study of the rewriting strategies used in inductive proofs by NQTHM, they developed an elegant family of strategies that they have called *rippling* strategies, since the changes effected by rewrite rules propagate around term trees like ripples on a pond [BvHH+89, BvHSI90].

To provide systematic control of rewriting in the LCF proof development system, Paulson introduced a language of *conversions* and *conversionals* [Pau83a, Pau87], reminiscent of the language of tactics and tacticals, which allows the piecing together of rewrite strategies from sets of rewrite rules. Conversional languages have been adopted in the HOL system [GM93], in Paulson's Isabelle system, in rewrite tactics written for Nuprl V3 [CH90, Bas89], and in the rewrite package that I have developed for Nuprl V4.1. In Section 4.2, I introduce the notion of conversions and give examples of conversionals that I commonly use. Many examples of the use of this rewrite package can be found in later chapters.

Features of this rewrite package include that it it uses second-order matching for rewrite rules( see Section 3.10.1), that it supports rewriting with respect to a variety of equivalence relations of varying strengths. It also assists in reasoning with order relations, in which case it automatically checks monotonicity properties of term constructors.

## 4.2 Conversions and Conversionals

To convey the idea of conversions and conversionals in this section, I present a simplified implementation of them. In later sections, I describe the actual conversions that I implemented. Note however that the conversionals introduced here have the same names and same behaviours as those in Nuprl V4.1. I therefore do not survey the main conversionals I created for Nuprl V4.1 elsewhere.

Let `convn` be an ML concrete type alias for the type of conversions. Later on, I describe the type that `convn` is an alias for in Nuprl V4.1. In this section, assume that `convn` is an alias for the type `term -> term`, where `term` is a type of terms that we want to rewrite. If $c$ is of type `convn`, then we can use $c$ to rewrite $t$ of type `term` by simply running the ML evaluator on the application $c\ t$.

For the purposes of the section only, I introduce a basic conversion called `RuleC : term -> convn`. The conversion `RuleC` expects its term argument to be of form $a = b$ where the free variables of $b$ are a subset of those in $a$. If the conversion `RuleC '`$a = b$`'` is applied to a term $t$, `RuleC` tries to find a substitution $\theta$ such that $\theta a = t$. If it succeeds, it returns the term $\theta b$. If a substitution cannot be found, `RuleC` raises an exception. The conversion `RuleC '`$a = b$`'` therefore rewrites instances of $a$ to corresponding instances of $b$. For example:

`RuleC '`$x + 0 = x$`'`

when applied to the term $(2 \times 3) + 0$ yields the term $2 \times 3$.

`RuleC` cannot by itself rewrite subterms of a term; if

`RuleC '`$x + 0 = x$`'`

is applied to the term $(1 + 0) \times 3$, it fails. There are a variety of higher-order conversions that map a conversion such as `RuleC` over all subterms of a term. An example of a conversional is `SweepUpC : convn -> convn`. If $c$ is a conversion, then `SweepUpC` $c$ is also a conversion. if `SweepUpC` $c$ is applied to some term $t$, an attempt is made to apply $c$ once to each subterm of $t$ working from the leaves of term $t$ up to its root. `SweepUpC` $c$ only fails every every application of $c$ fails. So if

`SweepUpC (RuleC '`$x + 0 = x$`')`

is applied to term $(1 + 0) \times 3$, it succeeds and returns the term $1 \times 3$.

The basic conversion for sequencing conversions is `ANDTHENC : convn -> convn -> convn`. In Nuprl, we reserve all-capital names for infix functions so a normal application of `ANDTHENC` to conversions $c_1$ and $c_2$ has form $c_1$ `ANDTHENC` $c_2$. When applied to a term $t$, $c_1$ `ANDTHENC` $c_2$ first applies $c_1$ to $t$. If $c_1$ succeeds, returning a term $t'$, then $c_2$ is applied to $t'$ and the result is returned. If either $c_1$ or $c_2$ fails, then $c_1$ `ANDTHENC` $c_2$ also fails. By analogy with *tacticals* being higher-order tactics, `ANDTHENC` is called a *conversional*.

The `ORELSEC : convn -> convn -> convn` conversional is for combining alternative conversions. When $c_1$ `ORELSEC` $c_2$ is applied to a term $t$, it first tries applying $c_1$ to $t$, and if this succeeds returns the result. If the application of $c_1$ fails, then it tries applying $c_2$ to $t$, failing if $c_2$ fails.

The definition for `SweepUpC` is:

```
letrec SweepUpC c t = (SubC (SweepUpC c) ORTHENC c) t
```

`SubC : convn -> convn` when applied to a conversion $c$ and a term $t$, applies $c$ to each of the immediate subterms of $t$. It fails only when $c$ fails on every immediate subterm. Hence, it always fails when $t$ is a leaf node and has no immediate subterms. $c_1$ `ORTHENC` $c_2$ is similar to $c_1$ `ANDTHENC` $c_2$ in that it first tries $c_1$ and then $c_2$. However `ORTHENC` only fails if both $c_1$ and $c_2$ fail. So, a call of `SweepUpC` $c$ on argument $t$ first tries to apply `SweepUpC` $c$ to the immediate subterms of $t$ and then then tries to apply $c$ to $t$ itself. Note that without the `t` argument on the left and right sides of the definition, `SweepUpC` in ML's call-by-value evaluation scheme would recurse indefinitely.

The definition for `ORTHENC` is:

```
let c1 ORTHENC c2 = (c1 ANDTHENC TryC c2) ORELSEC c2
```

where `TryC` $c$ is defined as $c$ `ORELSE IdC` and `IdC: convn`, when applied to any $t$, always returns $t$.

Other conversionals that are commonly used in the work described in this thesis are:

- `FirstC : convn list -> convn` which is an n-ary version of `ORELSEC`,
- `RepeatC : convn -> convn` which repeatedly tries applying a conversion till no further progress is made,
- `HigherC : convn -> convn` which applies a conversion to only nodes higher in a term tree. What I mean by 'higher' is probably best understood by studying the definition of HigherC:

```
letrec HigherC c t = (c ORELSEC SubC (HigherC c)) t
```

- `SweepDnC : convn -> convn` which sweeps a conversion down over a term tree from the root towards the leaves. Its definition is:

```
letrec SweepDnC c t = (c ORTHENC SubC (SweepDnC c)) t
```

- `NthC int -> convn -> convn`. `NthC` $i$ $c$ $t$ tries $c$ on each node in $t$ in pre-order order, but only on the $i$th success of $c$ does it go through with the rewrite that $c$ suggests. This is very useful during interactive proof when for example you want to unfold one instance of a definition but not any others.

## 4.3 Relations

The rewrite package supports rewriting with respect to both primitive and user-defined equivalence relations. Some examples are:

- $\sim$, the computational equality relation,
- $\cdot = \cdot \in \cdot$, the primitive equality relation of the type theory,
- $\iff$, if and only if,
- $\equiv$ `mod` , equality on the integers, mod a positive natural,
- $=_q$, equality of rationals represented as pairs of integers,
- $\equiv$, the permutation relation on lists.

The package also supports 'rewriting' with respect to any relation that is transitive but not necessarily symmetric or reflexive. This needs a bit of explaining. Proofs involving transitive relations and monotonicity properties of terms can be made very similar in structure to those involving equivalence relations and congruence properties.

For example, consider the following proof step that came up Forester's development of real analysis in Nuprl [For93].

$i{:}\mathbb{N}^+$
$j{:}\mathbb{N}^+$
$f{:}\mathbb{N}^+ \to \mathbb{N}^+$
$\mathtt{mono}(f)$
$\vdash$
$1/fi +_q 1/fj \;\; \leq_q \;\; 1/i +_q 1/j$

BY RWH (RevLemmaC 'monotone_le') 0

$\vdash$
$1/i +_q 1/j \;\; \leq_q \;\; 1/i +_q 1/j$

Here, the definition `mono(`$f$`)` is:

$$\texttt{mono}(f) =_{def} \forall a, b{:}\mathbb{N}^{+}.\ a < b \Rightarrow f\ a < f\ b$$

and the theorem `monotone_le` is:

$$\vdash \forall f{:}\mathbb{N}^{+} \rightarrow \mathbb{N}^{+}.\ \texttt{mono}(f) \Rightarrow \forall n{:}\mathbb{N}^{+}.\ n \leq f\ n$$

The tactic `RWH` $c$ $i$ tries to apply the conversion $c$ once to each subterm of clause $i$ of the sequent and the conversion `RevLemmaC` *name* converts lemma *name* into a right-to-left rewrite rule. Other examples of monotone rewriting can be found in Section 10.7.1.

It is interesting to note that logical implication $\Rightarrow$ can be treated a rewrite relation, since it is transitive. When it is, we have a generalization of forward and backward chaining.

For each user-defined relation, the user provides the rewrite package with lemmas about transitivity, symmetry, reflexivity and strength (a binary relation $R$ over a type $T$ is stronger than a relation $R'$ over $T$ if for all $a$ and $b$ in $T$, the relation $a\ R\ b$ implies that $a\ R'\ b$). These lemmas are used by the package for the justification of rewrites (see Section 4.4).

The user also provides a declaration in an ML object that identifies relation families and extra properties of relations. For example, here is relation family declarations for the standard order relations on integers:

```
lt_family:
  Relation Family
   <: i < j
   ≤: i ≤ j
   ≡: i = j
   ≥: i ≥ j
   >: i > j
```

and here is the `divides` relation from the theory of cancellation monoids described in Chapter 8:

```
mdivides_order_fam:
  Relation Family
   <: a p| b in g
   ≤: a | b in g
   ≡: a ∼{g} b
   ≥: a |by b in g
   >: ?
```

## 4.4 Justification of Rewrites

In a theorem-proving setting, rewriting has to be rigorously justified. Firstly, rewrite rules must be generated from previously proven lemmas or from hypotheses; a conversion such as `RuleC` described in the previous section is not allowed. Secondly, there must be reason to believe that congruence properties are respected. For example, if the conclusion of a sequent is $C[t]$ where $C[\,]$ is a context, and term $t$ is rewritten to $t'$, then we might expect the sequent:

$$t = t' \vdash C[t] \Longleftrightarrow C[t']$$

to be provable. Some logics guarantee that equal terms can always be substituted for one another in any context, so in these logics congruence properties need not be explicitly checked during proof.

Nuprl has a rich variety of equality and equivalence relations, some primitive and some user defined. The strongest, a computational equality relation, has been shown to be a congruence relation everywhere [How89]; it is always valid to replace some term by a computationally equivalent one. However, many other relations are only congruence relations in certain contexts.

Rewriting with respect to the computational equality is justified using *direct-computation* primitive rules. All other rewrites are justified by tactics that construct congruence proofs. The direct computation rules are described in Section 4.4.1 and congruence proofs are described in Section 4.4.2.

### 4.4.1 Direct Computation

The main direct-computation rule enable one to select arbitrary redices in clauses of sequents and contract them. A *redex* is the left-hand side of some computation rule such as the $\beta$ rule:$(\lambda x.b_x)\ a \longrightarrow b_a$. The right-hand-side of such a rule is sometimes called the *contractum* and the process of replacing instances of the left with instances of the right *contraction*.

Redices within a clause are identified to the rule by giving the rule a copy of the clause with the selected redices *tagged*. Section 4.6 gives examples of tagged terms. A variant on the main rule allows a clause to be replaced with another computationally-equal term with new redices. Here, one gives the rule the clause with the new redices tagged, and the rule checks that reducing the tagged redices gives the original clause.

The direct-computation rules are extremely useful. Rewrites using direct computation are 10–100 times faster than similar rewrites using congruence proofs.

The same tagging scheme is also used to select abstractions in clauses for folding and unfolding.

### 4.4.2 Congruence Proofs

An example of a rewrite justified by a congruence proof is as follows. This example comes from an auxiliary theorem that was used to prove the Chinese Remainder Theorem. Say we want to prove the goal:

$$x, y, a, b{:}\mathbb{Z},\ m{:}\mathbb{N}^+,\ x \equiv 1 \texttt{ mod } m,\ y \equiv 0 \texttt{ mod } m \vdash a * x + b * y \equiv a \texttt{ mod } m$$

then a first step might be to rewrite using the hypotheses to eliminate $x$ and $y$ from the conclusion, giving the new sequent:

$$x, y, a, b{:}\mathbb{Z},\ m{:}\mathbb{N}^+,\ x \equiv 1 \texttt{ mod } m,\ y \equiv 0 \texttt{ mod } m \vdash a * 1 + b * 0 \equiv a \texttt{ mod } m$$

This can be justified because $*$, $+$ and $\cdot \equiv \cdot \texttt{ mod } m$ all respect $\cdot \equiv \cdot \texttt{ mod } m$. However, if the conclusion were instead $a{*}x{+}b{*}y = a \in \mathbb{Z}$, the rewrites could not be justified, since $\cdot = \cdot \in \mathbb{Z}$ does not respect $\cdot \equiv \cdot \texttt{ mod } m$.

Such rewrites are justified by executing tactics that do top-down congruence reasoning. Part of the justification for the above rewrite is shown in Figure 4.1.

$\vdash a * x + b * y \equiv a \texttt{ mod } m \iff a * 1 + b * 0 \equiv a \texttt{ mod } m$<br>
`BY` *functionality*<br>
  $\ldots \vdash a * x + b * y \equiv a * 1 + b * 0 \texttt{ mod } m$<br>
  `BY` *functionality*<br>
    $\ldots \vdash a * x \equiv a * 1 \texttt{ mod } m$<br>
    `BY` *functionality*<br>
      $\ldots \vdash a \equiv a \texttt{ mod } m$<br>
      $\langle$*continued*$\rangle$<br>
      $\ldots \vdash x \equiv 1 \texttt{ mod } m$<br>
      $\langle$*continued*$\rangle$<br>
    $\ldots \vdash b * y \equiv b * 0 \texttt{ mod } m$<br>
    $\langle$*continued*$\rangle$<br>
  $\ldots \vdash a \equiv a \texttt{ mod } m$<br>
  $\langle$*continued*$\rangle$<br>
  $\ldots \vdash m = m \in \mathbb{Z}$<br>
  $\langle$*continued*$\rangle$

Figure 4.1: Rewrite Justification

At each `BY`, a *functionality* tactic is applied which reduces the goal of proving the equivalence of two terms with the same outermost term constructor to the proving of the equivalence of corresponding immediate subterms. These tactics draw on *functionality* information about terms from primitive rules and lemmas. Ignoring parameters that relations can have, (like the $m$ above), a functionality lemma has the basic form:

$$\begin{array}{l}\forall x_1, y_1{:}T_1, \ldots, x_n, y_n{:}T_n.\ x_1\ r_1 y_1 \Rightarrow \ldots \Rightarrow x_n r_n y_n \\ \quad \Rightarrow op(x_1; \ldots; x_n)\ R\ op(y_1; \ldots; y_n)\end{array}$$

Most commonly the relations $R$ and $r_i$ are equivalence relations, but in general they needn't be. See Section 4.3 for details. Also, whenever subterm $i$ of $op$ is thought of as a parameter that never normally would get rewritten, then the antecedent involving $r_i$ can be dropped and $x_i$ can be used on both the left and right of the consequent of the formula.

Other parts of justifications contain tactics that draw on transitivity information and information on the relative strengths of relations. Continuing the previous example, suppose we included rewrite rules for arithmetic simplification based on lemmas such as:

$$\vdash \forall i{:}\mathbb{Z}.\ i * 1 = i \in \mathbb{Z}$$

into our rewrite. Part of the justification might then look like as shown in Figure 4.2. The *transitivity* step draws on the transitivity of $\cdot \equiv \cdot$ `mod` $m$ and the

$\vdash a * x \equiv a$ `mod` $m$
`BY` *transitivity*
  $\vdash a * x \equiv a * 1$ `mod` $m$
  `BY` *functionality*
    $a \equiv a$ `mod` $m$
    ⟨*continued*⟩

    $x \equiv 1$ `mod` $m$
    `BY` *hypothesis*

  $\vdash a * 1 \equiv a$ `mod` $m$
  `BY` *strengthening*
    $\vdash a * 1 = a \in \mathbb{Z}$
    `BY` *lemma*
    ⟨*continued*⟩

Figure 4.2: Justification with Transitivity Step

*strengthening* step draws on the fact that $\cdot = \cdot \in \mathbb{Z}$ is a stronger relation than $\cdot \equiv \cdot$ `mod` $m$. Again, this information comes from both primitive rules and lemmas.

Tactics that create justification proofs are generated automatically by Nuprl conversions. Section 4.5 gives some basic details on how this is done.

## 4.5 Operation of Nuprl Conversions

In Nuprl the conversion type `convn` is an alias for

```
env -> term -> (term # reln # just)
```

Here, `env` is an ML abstract type for *environments*, `reln` is an ML abstract type for *rewrite relations* and `just` is an ML abstract type for *justifications*. I explain what each of these abstract types is below.

If a conversion $c$ of type `convn` is applied to an environment $e$ and a term to be rewritten $t$, it returns a triple $\langle t', r, j\rangle$. The environment $e$ tells the conversion $c$ about the context of the term $t$. This includes the types of all the variables that might be free in $t$, as well as propositions that the conversion can assume true. The term $t'$ is the rewritten version of the term $t$. The relation $r$ specifies the relationship between $t$ and $t'$. The justification $j$ indicates how the relation $t\ r\ t'$ may be proven.

The function

```
Rewrite : convn -> int -> tactic
```

is the basic tactic for applying conversions.

`Rewrite` $c\ i$ applies conversion $c$ to $Q_i$, clause $i$ of the sequent. `Rewrite` works in two phases; firstly it constructs an environment from the sequent and passes that and $Q_i$ to the conversion. The conversion constructs a justification $j$ in a bottom-up fashion, starting from successful instances of rewrite rule applications, and also returns the term $Q'_i$ and the relation $r$. Secondly, `Rewrite` executes justification $j$. If $j$ is of the direct computation kind, it is passed to a direct computation tactic to rewrite clause $i$. If $j$ is a tactic justification, then it is used to prove the goal $\vdash Q_i\ r\ Q'_i$, and some simple logical reasoning allows clause $i$ to be replaced by $Q'_i$[1].

Using `Rewrite`, rewrite tactics are constructed that rewrite with respect to sets of lemmas, hypotheses and direct-computation rules (Section 4.6 and Section 4.7 describe these rules).

A single set of conversionals handles justification of rewrites both by direct computation and by tactic-driven congruence proofs. Where possible computation justifications are used, but sometimes it is necessary to convert a computation justification to a tactic justification.

[1] occasionally, when one hypothesis is rewritten using another hypothesis as a rewrite rule, $Q'_i$ contains free variables declared to the right of position $i$ in the hypothesis list. In this case, $Q'_i$ is usually placed at the end of the hypothesis list.

## 4.6 Direct Computation Conversions

A justification for a direct-computation conversion takes the form of a *compute sequence*. A compute sequence is of type `(tok # term) list`, where each pair in a sequence is of form *op*,$t$. The token *op* indicates a direct-computation operation. The possible values of *op* are `NOP` for a null operation, `FWD` for a forward computation step, and `REV` for a reverse computation step. The term $t$ is a possibly tagged term. A compute sequence

$$[op_1, t_1; op_2, t_2; \ldots; op_n, t_n]$$

is a justification for rewriting term $t_a$ to term $t_b$ if all the following hold:

1. $op_1, t_1 = \texttt{NOP}, t_a$
2. $op_n, t_n = \texttt{NOP}, t_b$
3. if $op_i$ is `FWD`, then executing a forward computation step as indicated by the tags on term $t_i$ results in a term equal to term $t_{i+1}$ with tags removed.
4. if $op_i$ is `REV`, then executing a forward computation step as indicated by the tags on term $t_i$ results in a term equal to term $t_{i-1}$ with tags removed.
5. sequences of consecutive `FWD` and `REV` operations are separated from one another by `NOP` operations.

An example of a compute sequence that justifies the unrolling of a Y combinator is:

$$
\begin{array}{ll}
[\texttt{NOP}, & \texttt{Y F} \\
;\texttt{FWD}, & \boxed{\texttt{Y}}\ \texttt{F} \\
;\texttt{FWD}, & \boxed{\texttt{(}\lambda\texttt{f.(}\lambda\texttt{x.f(xx)) (}\lambda\texttt{x.f(xx))) F}} \\
;\texttt{FWD}, & \boxed{\texttt{(}\lambda\texttt{x.F(xx)) (}\lambda\texttt{x.F(xx))}} \\
;\texttt{NOP}, & \texttt{F ((}\lambda\texttt{x.F(xx)) (}\lambda\texttt{x.F(xx)))} \\
;\texttt{REV}, & \texttt{F}\ \boxed{\texttt{(}\lambda\texttt{f.(}\lambda\texttt{x.f(xx)) (}\lambda\texttt{x.f(xx))) F}} \\
;\texttt{REV}, & \texttt{F (}\boxed{\texttt{Y}}\ \texttt{F)} \\
;\texttt{NOP}, & \texttt{F (Y F)}]
\end{array}
$$

Here, the $\boxed{\phantom{x}}$'s indicate tags.

Compute sequences have a couple of nice properties. Firstly, they can be specialized; if a compute sequence justifies rewriting term $t_a$ to term $t_b$, then a compute sequence that justifies rewriting $\theta t_a$ to $\theta t_b$ for any substitution $\theta$ can be formed by applying $\theta$ to each term in the compute sequence. Secondly, they can be reversed; a compute sequence justifying rewriting term $t_a$ to term $t_b$ can be easily transformed into one that justifies rewriting $t_b$ to $t_a$.

I exploited both these properties in defining a direct computation conversion:

```
MacroC : convn →term →convn →term →convn
```

`MacroC` $c_a$ $t_a$ $c_b$ $t_b$ creates a conversion for rewriting instances of $t_a$ to instances of $t_b$, providing that applying the conversion $c_a$ to $t_a$ and $c_b$ to $t_b$ results in the same term.

`MacroC` allows the easy construction of quite complicated yet well controlled conversions; often the conversions used for $c_a$ and $c_b$ can be much simpler than if an equivalent conversion were built without `MacroC`. For example, consider the following definitions in Nuprl V4.1's theory about the booleans:

```
bool:
  𝔹 == Unit + Unit

bfalse:
  ff == inr ·

ifthenelse:
  if b then t else f fi  == case b of inl() => t | inr() => f

band:
  p ∧b  q == if p then q else ff fi
```

and the conversion definition:

```
let SimpleMacroC t1 t2 names =
                    MacroC (SemiNormC names) t1 (SemiNormC names) t2
```

Here, `SemiNormC names` is a normalization conversion that unfolds abstractions named in `names` and reduces all redices. A simplification conversion for `band` with false left-hand-subterm could then be written:

```
let band_ff_lC =
    SimpleMacroC 'ff ∧b u' 'ff' [band;ifthenelse;bfalse] ;;
```

Without `MacroC` this conversion would have to be written as

```
let band_ff_lC =
    UnfoldTopC `band`
    ANDTHENC UnfoldTopC `ifthenelse`
    ANDTHENC AddrC [1] (UnfoldTopC `bfalse`)
    ANDTHENC RedexC ;;
```

Here, `RedexC` is the basic conversion for contracting primitive redices, `AddrC` applies a conversion to an addressed subterm and `UnfoldTopC` unfolds a named abstraction.

`MacroC` can be thought as providing a means of proving and then applying direct-computation 'lemmas'. Unfortunately, the gain in efficiency expected from using lemmas is not realized, because `MacroC` still has to replay all the primitive intermediate steps of the direct-computation calculation that it encapsulates, every time it is applied.

I have constructed a variety of direct-computation conversions. Probably the most widely used is `AbReduceC` which reduces both primitive and abstract redices. By an *abstract redex* I mean some abstract term that unfolds to a primitive redex. For example, `ff` $\wedge$ $x$ is an abstract redex which unfolds to the primitive redex `case inr` $\cdot$ `of inl() =>` $x$ `| inr() => ff`. Each time a new non-canonical abstraction is introduced, the user can define conversions for it like the `band_ff_lC` above.

Whenever recursive function definitions are created (see Section 2.2.5), the system automatically uses `MacroC` to create a conversion for unrolling the definition. These conversions completely hide the underlying Y-combinator definition of the recursive functions.

## 4.7 Tactic Conversions

The basic tactic conversions are `LemmaC` and `HypC` which take lemmas and hypotheses respectively and convert them into rewrite rules. The lemmas and hypotheses usually have form:

$$\forall x_1{:}T_1, \ldots x_n{:}T_n.\ A_1 \Rightarrow \ldots \Rightarrow A_m \Rightarrow t\ r\ t'$$

`LemmaC` and `HypC` convert such formulae into rules for rewriting instances of $t$ to instances of $t'$. There are variants on these conversions that allow for right-to-left rewriting, checking the assumptions $A_i$ before allowing the rule to be applied, providing explicit bindings for one or more of the $x_j$ and handling formulae with a conjunction of rewrite relations instead of just one.

The `SubC : convn -> convn` conversional is responsible for adding functionality steps to the justification. If `SubC` $c$ is applied to some term $\phi(a_1; \ldots; a_n)$, then $c$ is applied to each of the $a_i$. `SubC` succeeds if at least one application of $c$ succeeds. let $b_i$ be the term that $c$ rewrites $a_i$ to when $c$ succeeds, and be $a_i$ otherwise. Let $T_i$ be the tactic that can justify the goal $\vdash a_i \equiv b_i$ and `Fun(`$\phi$`)` be a tactic that incorporates functionality information about $\phi$. Then `SubC` $c$ constructs a tactic `Fun(`$\phi$`) THENL [`$T_1$`;`$\ldots$`;`$T_n$`]` as the justification for $\phi(a_1; \ldots; a_n) \equiv \phi(b_1; \ldots; b_n)$.

In practice for a single term $\phi$ there might be many different relations that subterms of $\phi$ are rewritten with respect to. For example, subterms of $+_{\mathbb{Z}}$ might be rewritten with respect to $<_{\mathbb{Z}}$, $\leq_{\mathbb{Z}}$, $=_{\mathbb{Z}}$, $\geq_{\mathbb{Z}}$, $>_{\mathbb{Z}}$ or $\cdot \equiv \cdot$ `mod` $m$. For each combination of subterm relations, `SubC` has to come up with some appropriate tactic. It would be impractical to have a distinct tactic for each combination. Instead, `SubC` keeps a small set of tactics for each term $\phi$ and adapts these as and when necessary. If the subterms of $\phi$ are rewritten with respect to relations $r_1 \ldots r_n$, then `SubC` picks a tactic that expects subterm relations $s_i$, where each $s_i$ is no stronger than $r_i$. It then uses relation strengthening tactics when necessary to bridge the gap between each $s_i$ and the $r_i$. The strengthening step shown in Section 4.4.2 is generated by `SubC` in this manner. If there is a choice of tactics, `SubC` picks the one that expects the strongest relation between $\phi(a_1; \ldots; a_n)$ and $\phi(b_1; \ldots; b_n)$. If there is no unique strongest relation then currently, the first which occurs in the Nuprl library is chosen. So far, this hasn't been a problem

For the $+_{\mathbb{Z}}$ operator, `SubC` currently uses tactics that can prove the following goals:

$$
\begin{array}{l}
x = y,\ x' = y' \vdash x + x' = y + y' \\
x < y,\ x' \leq y' \vdash x + x' < y + y' \\
x \leq y,\ x' < y' \vdash x + x' < y + y' \\
x \leq y,\ x' \leq y' \vdash x + x' \leq y + y' \\
x \equiv y \ \texttt{mod}\ m,\ x' \equiv y' \ \texttt{mod}\ m \vdash x + x' \equiv y + y' \ \texttt{mod}\ m
\end{array}
$$

When using order relations, `SubC` is able to invert relations when necessary to match the available tactics.

The `ANDTHENC : convn -> convn -> convn` conversional adds transitivity steps to justifications. In a similar way to `SubC`, the `ANDTHENC` conversion picks a suitable transitivity tactic for each pair of relations it finds that terms have been rewritten with respect to.

## 4.8 Abbreviations and Extensions

In this thesis, I use several abbreviations for tactics that apply conversions. Here are a few of them:

- ```
  let RW = Rewrite ;;
  let RWH c = Rewrite (HigherC c) ;;
  let RWD c = Rewrite (SweepDnC c) ;;
  let RWU c = Rewrite (SweepUpC c) ;;
  ```

- `RWW` *string* (standing for R(e)W(rite) W(ith)) is a rewrite tactic for using a set of rewrite rules named in *string*. There are conventions for selecting hypotheses, lemmas, definitions and direct-computation conversions as rewrite rules to be repeatedly applied. Names can be annotated to indicate that equations should be turned into right-to-left rewrite rules rather than left-to-right rules.

- `RWO` (standing for R(e)W(rite) O(nce)) is like `RWW` except that it uses `HigherC` to apply the named set of rules in a more restricted fashion.
  ```

In Section 10.7.1 there are also several instances given of tacticals and conversionals with the word `Force` in them; for example, `ForceReduceC '5'`. Here I was experimenting with assigning Nuprl terms differing *strengths*, strengths being arranged into a partial order. Simplification conversions could be passed a *force* as an extra parameter, and rewrite rules in the simplification conversion would only be enabled when applied to terms with strength no greater than this force.

## 4.9 Discussion

This rewrite package has been invaluable for the work described in this thesis. I created conversions for my algebraic theories that put expressions over monoids, abelian monoids, groups, abelian groups, rings and commutative rings into normal form. I also experimented with conversions that worked well in conjunction with `NthC` for the precise and repeated application of certain rules (see the end of Section 4.2 for a description of this conversional and Section 10.7.1 for examples of its use). In the theories described in this thesis, roughly two-thirds of all tactic invocations involved some kind of rewriting. The most widely used rewrite tactic was `AbReduce`, based on `AbReduceC`, that reduces both primitive and abstract redices. I give several examples later on of how I exploited features such as handling differing strengths of rewrite relations, handling monotone reasoning, and using the second-order matching.

This rewrite package has was also extensively used in Forester's implementation of constructive real analysis in Nuprl V4.1 [For93].

This package differs from that developed by Basin for Nuprl V3 [Bas89] in that he only implemented support for rewriting with respect to the $\cdot = \cdot \in \cdot$ equality relation of Nuprl's type theory and the if and only if relation $\iff$. He also didn't implement any direct computation conversions. Howe also experimented with rewriting with respect to $\cdot = \cdot \in \cdot$ and $\iff$, and in constructing simplification conversions that grouped together sets of rewrite rules [CH90]. He did implement some direct-computation conversions, but these were not integrated with the tactic-based conversions and were much more basic than those that I developed.

To my knowledge, the issue of dealing with multiple strengths of rewrite relations has not yet been addressed anywhere other than in Nuprl. Researchers in the field of rewriting have begun to consider monotone rewriting [LA93, BG94]. It will be interesting to see whether these ideas can be adapted to work well in an interactive theorem proving context.

# Chapter 5

# Methodology for Algebra

## 5.1 Introduction

I discuss in this chapter some of the issues involved in defining classes of algebraic objects in Nuprl's type theory. I intend the phrase *algebraic objects* here to include:

- groups, rings and modules; mathematical structures that have a carrier and a set of operations over the carrier.
- Categories, topologies, orders.
- ADT's: for example, stacks or parsers. A class corresponds to a specification of an ADT and the objects in the class are implementations of the ADT. Closely related are the implementations of modules in languages such as SML.

In general, a class definition can be considered to contain

- A signature. This specifies the types of the components of instances of the class. Sometimes I call these instances *objects* of the class, or *implementations* of the class. I include in the notion of *component* constants, relations, functions and sets. Sometimes I refer to the component of an object and the corresponding type in its signature as a *field* of that object or signature respectively.
- A predicate on instances of the signature that specifies properties that components of instances should have.

Classes can be parameterized, often by objects of other classes. For example, a class for vector spaces could take a field as a parameter.

In theory it should be quite straightforward to use Nuprl's dependent-product type and set type to construct such classes. In practice, a number of issues have to be carefully considered.

One major one is the approach towards constructivity. Choices often have to be made about how to deal with the computational content of classes. For example, classically, in the class of integral domains, a partial division function can always be defined. Constructively, we could set up definitions for two kinds of integral domain: one requiring that a computable division function be supplied and one not. In Nuprl's type theory there are a couple of options as to how we could design an integral domain class to require such a function:

1. a type could be explicitly reserved in the class signature for the function. Perhaps the type
$$T^2 \to (T + \mathsf{Unit})$$
would be used. The class predicate might then include the following predicate on inhabitants of the type $T^2 \to (T + \mathsf{Unit})$:
$$\begin{array}{ll} \lambda d{:}T^2 \to (T + \mathsf{Unit}). & \forall x, y, z{:}T.\ x = z * y \Longrightarrow d\ \langle x, y\rangle = \mathsf{inl}(z) \\ & \forall x, y{:}T.\ \neg\exists z{:}T.\ x = z * y \Longrightarrow d\ \langle x, y\rangle = \mathsf{inr}(\cdot) \end{array}$$
I sometimes refer to this as the *explicitly constructive* approach.

2. A type considered as a proposition could be included in the signature. The proposition might be
$$\forall x, y{:}T.\ (\exists z{:}T.\ x = z * y \vee \neg\exists z{:}T.\ x = z * y).$$
The division function is then implicit in the computational content of this proposition. This becomes clearer if the proposition is written out as the corresponding type:
$$x{:}T \to y{:}T \to ((z{:}T \times x = z * y) + ((z{:}T. \times x = z * y) \to \mathsf{Void})).$$
I sometimes refer to this as the *implicitly constructive* approach.

The issue of computational content also comes up when considering the equality relation on the carrier of algebraic objects and membership in subsets of the carrier. I haven't seen this discussed in the literature on constructive algebra, but as I explain in Section 5.2 and Section 5.3 this content could be considered significant. These sections also discuss the decidability and stability of the equality and membership relations.

Section 5.4 explores how class design affects the subtyping relationship between classes and Section 5.5 discusses the approach I adopted for the work in this thesis.

## 5.2 Equality Relations

### 5.2.1 Computational Content

One significant issue when considering the design of classes for *implicitly constructive* mathematics is the importance of the computational content of the equality relation on the carriers of implementations of classes. A few examples of equality relations that have non-trivial computational content when true, and examples of their uses in implementations of classes are as follows:

- the $\iff$ relation on propositions, used when forming a Heyting algebra. The computational content of $A \iff B$ when true is a pair of functions of types $A \to B$ and $B \to A$.

- the permutation relation $\mathsf{perm}$ on lists, used when forming an abelian monoid of multisets using lists. The computational content of the proposition

  $$\mathsf{perm}(as, bs)$$

  when true might be a bijective function $f$ on $\{1 \ldots n\} \to \{1 \ldots n\}$, $n$ being the length of both $as$ and $bs$, that indicates how the $bs$ are a permutation of the $as$.

- the associate relation $\sim$ in the factorization theory of cancellation monoids or integral domains. The computational content of $a \sim b$ might be considered to be a pair $\langle u, v \rangle$ such that $a \times u = b$ and $b \times v = a$.

- the equality relation on a group quotiented by a normal subgroup where membership in the normal subgroup is computationally interesting. Let a normal subgroup of a group $G$ be represented by a unary predicate $H$ on the group carrier $|G|$. We are then assuming that $H\ a$ for $a$ in $|G|$ is in general computationally interesting (see Section 5.3.6 for examples of such relations). The computational content of the relation $a \equiv b(\mathsf{mod}\ H)$ is then the computational content of $H\ (a - b)$.

When an equality relation over an implementation carrier has non-trivial computational content, then the predicates stating that the relation is reflexive, symmetric and transitive all have computational content, as do predicates stating that functions over the carrier respect the equality. It is easy to imagine that the computational content of such predicates might find its way into theorem extracts, especially when one is reasoning by rewriting with respect to such relations.

An example of a predicate involving a computationally-interesting equality relation is the following, which says that the permutation relation on lists $\mathsf{perm}$ is symmetric:

$$\forall a, b{:}T\ \mathsf{List}.\ \mathsf{perm}(a, b) \Rightarrow \mathsf{perm}(b, a)\ .$$

The computational content might be a function

$$\lambda a, b, p.\ \mathsf{inv}(p),$$

where inv is a function for inverting a bijective function. Note that a bijective function is commonly represented constructively by a *pair* of functions that are mutual inverses of each other. With such a representation, the constructive implementation of inv is trivial; it simply swaps elements of pairs.

Another computationally-interesting predicate involving the perm relation is one stating the functionality of the list append operation @ with respect to the perm relation:

$$\forall a, b, c, d{:}T\ \mathsf{List}.\ \mathsf{perm}(a, b) \Rightarrow \mathsf{perm}(c, d) \Rightarrow \mathsf{perm}(a\ @\ c,\ b\ @\ d).$$

The computational content of this proposition might be the function

$$\lambda a, b, c, d, p, q.\ \mathsf{permappend}(p, q),$$

where permappend is a function for suitably combining the permutations $p$ and $q$. For permappend to be properly constructive it usually will also take as argument the length of one of the lists $a, b, c$ or $d$. I describe a development of permutations in Nuprl in Chapter 7.

The design strategy for classes in Nuprl's type theory is very different depending on whether or not one wants the capability of extracts accessing the computational content of class predicates such as described above. The alternative design strategies are explored in the next two sections.

### 5.2.2 Ignoring Computational Content

If the computational content of the carrier equalities is always to be ignored, then the equality relation can be taken to be the one that is naturally associated with the carrier type by the type theory. One suitable definition for the class of abelian semigroups is then:

$$\begin{array}{lll} \mathsf{AbSGrp} \doteq & G{:}\mathbb{U} & \\ & \times\ \{*{:}(G^2 \to G) \mid & \forall a, b, c{:}G.\ (a * b) * c = a * (b * c) \in G \\ & & \wedge\ \forall a, b{:}G.\ a * b = b * a\ \in G\} \end{array}$$

There can never be a problem here using the set type for the associativity predicate since the predicate is stable. The type theory guarantees that for any pair $\langle G, *\rangle$ in the AbSGrp class, the operation $*$ respects the equality on $G$.

One nice feature of this approach is that we can take advantage of Nuprl's quotient type when forming instances. For example, assume we wanted to show that lists of integers under the uncurried append function formed an abelian group. We cannot prove that the pair

$$\langle \mathbb{Z}\ \mathsf{List}, \mathsf{append}\rangle$$

is an instance of AbSGrp, because append is not commutative when considered as a function over lists. However, it is commutative when considered as a function on multisets, so if we define a multiset type using Nuprl's quotient type:

$$T \ \mathsf{MSet} \doteq x, y{:}T \ \mathsf{List}//\mathsf{perm}(x, y)$$

then we could show that the pair

$$\langle \mathbb{Z} \ \mathsf{MSet}, \mathsf{append} \rangle$$

is an instance of the AbSGrp class.

### 5.2.3 Maintaining Computational Content

Here we assume that the equality relation of instances of classes might sometimes have significant computational content. The equality relation associated with the carrier type by the type theory is always free of computational content, so it cannot be used in class definitions. Instead, the equality relation on the carrier instances has to supplied as an explicit part of instances. One suitable definition for the class of abelian semigroups is then:

$$\begin{array}{lcl} \mathsf{SGroup} & \doteq & G{:}\mathbb{U} \\ & & \times \ \ eq{:}(G^2 \to \mathbb{P}) \\ & & \times \ \ *{:}(G^2 \to G) \\ & & \times \ \ \forall a, a'b, b'{:}G.\ (a \ eq \ b) \Rightarrow (a' \ eq \ b') \Rightarrow (a \ * \ a') \ eq \ (b \ * \ b') \\ & & \times \ \ \forall a, b, c{:}G.\ ((a * b) * c) \ eq \ (a * (b * c)) \\ & & \times \ \ \forall a, b{:}G.\ (a * b) \ eq \ (b * a) \end{array}$$

We need explicitly a predicate stating that $*$ respects the equality $eq$. Both this functionality predicate, the associativity predicate, and the commutativity predicate might have significant computational content, so neither may be hidden in the right-hand side of a set type.

### 5.2.4 Stability Considerations

The notion of *stability* in Nuprl is introduced in Section 3.7.3.

If an equality relation on the carrier of an instance of a class is stable, then it would be possible to set up rewrite tactics so that the class definition ignores the computational content of the equality relation, but also so that the computational content is generated when rewriting in contexts in which it is needed for an extract. This generated extract might not be as efficient to execute as when using classes that maintain computational content.

Stable equality relations that have interesting computational content are not uncommon. For example, the permutation relation on lists is stable when the lists are over a discrete type, and the associated relation is stable when there exists a division function. Also, the $\iff$ relation is stable when its arguments are stable.

### 5.2.5 Discreteness

It is standard practice in constructive mathematics to say that an equality relation is *discrete* when it is decidable. However, in constructive type theory, there are two clearly-defined distinct kinds of decidability.

Let a predicate $P$ on a variable $x$ of type $T$ be *constructively decidable* if the proposition $\forall x{:}T.\ P_x \vee \neg P_x$ is true, or, by the propositions-as-types correspondence, the type $x{:}T \to (P_x + (P_x \to \mathsf{Void}))$ is inhabited. Let a predicate $P$ be *classically decidable* if there is a function of type $T \to \mathbb{B}$ that returns $\mathsf{tt}$ on argument $a$ when the type $P_a$ is inhabited and $\mathsf{ff}$ otherwise. It is trivial to show that a predicate is constructively decidable just when it is classically decidable *and* it is stable.

Computer algebra systems such as Axiom commonly require that instances of algebraic classes come equipped with boolean-valued equality functions. From the constructive type-theory point-of-view, such classes are therefore classically decidable, but not necessarily constructively stable.

## 5.3 Subsets

### 5.3.1 Introduction

Here I discuss how we might talk generally about the 'subsets' of a type, thinking in particular about subsets of the carrier type of implementations of classes, though much of this discussion is more general. Approaches to subsets should allow the straightforward definition of

- a power-type, a type of all subsets of a type.
- common operations such as intersection, union and complement.
- relations such as the subset relation between subsets.
- families of subsets and operations such as intersection and union over these families.

I consider several approaches in Section 5.3.4 to Section 5.3.5. For simplicity, these approaches all just consider defining subsets of a type whose equality is the natural one associated with the type. However, the equality naturally associated with the power-type definition is almost always not subset equivalence, so these power-type definitions, as presented, cannot be applied to themselves to generate types of families of subsets. This point is returned to in Section 5.5.

Some approaches consider the computational content of subset membership to be important and some don't. In Section 5.3.6 I give several instances of subsets whose membership could be considered to be computationally interesting.

### 5.3.2 Using Type-Valued Predicates

Here we consider a subtype of some type $T$ to be a type-valued predicate on $T$. The type of all subtypes of type $T$ is then the function type $T \rightarrow \mathbb{P}$. Membership in a subtype is expressed simply by predicate application; An element $x$ of type $T$ is in a subtype $S$, where $S \in T \rightarrow \mathbb{P}$, just when the type $S(x)$ is inhabited. In general this inhabiting term could be computationally interesting.

It is very straightforward to develop some basic 'set theory' based on predicates, treating $T$ as a 'universal' type. For example, binary union and intersection are easily defined, as are the membership relation and the subtype relation. These sets are not exactly like classical sets in that the complement of the complement of some set is in general a super-set of that set rather than equivalent to it. Huet observed that such subsets can be thought of as topological open sets, in which case the double complement operation corresponds to set closure.

It is slightly inconvenient that subtypes are now functions rather than types. To form the type corresponding to some subtype $S$, we have two options: if we don't care about the computational information related to subtype membership, then we can use the set type constructor to form the type $\{x{:}T \mid S\ x\}$. Otherwise, we use the dependent product type to form the type $x{:}T \times S\ x$.

Another drawback of this representation of subtypes is that the equality relation $\cdot = \cdot \in T \rightarrow \mathbb{P}$ provided by Nuprl's type theory on this subtype representation type is much stronger than the desired extensional equality.

### 5.3.3 Using Boolean-Valued Predicates

This is the standard approach in classical type theories such as HOL. A subtype of some type $T$ is a boolean-valued predicate on $T$. The type of all subtypes of $T$ is then $T \rightarrow \mathbb{B}$. The appeal of this approach is that if the built-in equality on $T$ is the one we care about, then the built-in equality on subtypes is then the desired one.

However, in Nuprl's constructive type theory, we are restricted as to what instances of such subtypes we can talk about. All instances have to be computable functions. We are prevented, for example, from using the quantifiers of predicate logic to define instances whenever the domains of quantification are not finite. This is a severe restriction.

### 5.3.4 Using Types

Unlike set theory, Nuprl's type theory has no true 2-place membership predicate $\cdot \in \cdot$, so there is no way to make the definition:

$$\mathsf{PwT}(T) \doteq \{S{:}\mathbb{U} \mid \forall x{:}S.\ x \in T\}$$

The proposition in Nuprl displayed elsewhere in this thesis as $t \in T$, is a notational abbreviation for $t = t \in T$. A consequence of this is that the proposition $t \in T$ in Nuprl's type theory is only considered well-formed when it also happens to be true! Furthermore, there are serious problems trying to add a fully-fledged membership predicate. One is that there is no obvious way to work a type corresponding to this predicate into the semantics for Nuprl given by Allen. Another is that even if such a type could be introduced, the semantics of Nuprl's sequents would make it very difficult to formulate any rules about it.

The current type theory provides no primitive power-type constructor that is the analog of the power-set constructor in set theory. Perhaps one could be added. However, if a subtype relation type $\subset$ were added to the type theory, we could make the definition:

$$\mathsf{PwT}(T) \doteq \{S{:}\mathbb{U} \mid S \subset T\}$$

The type $S \subset T$ would be inhabited by some trivial element just in case every element of type $S$ were a member of type $T$ and the equalities relations naturally associated with $S$ and $T$ were the same. Howe, Allen and Mendler [Men88] have each considered such an extension, and not thought it problematic.

Unfortunately, because equality of set-types is intensional in Nuprl's current type theory, the natural equality associated with $\mathsf{PwT}(T)$ would be stronger than desired no matter what the equality associated with the the $\subset$ relation is.

Assume a $\subset$ relation has been added and $\mathsf{PwT}(T)$ has been defined as above. A 3 place membership predicate can be defined as follows:

$$x \in_T S \doteq \{y{:}T \mid y = x \in T\} \subset S$$

This predicate would be well formed if *some* type $T$ were given that $x$ inhabits, but now there is no well-formedness constraint that $x$ be a member of $S$.

From a constructive point of view, the very notion of a power-type is not as useful as one might imagine. It is easy to give examples of families of subtypes where each member of a subtype has associated with it extra computational information justifying why it is in the subtype. See Section 5.3.6 for details.

### 5.3.5 Using Domains of Injections

Following Bishop, we could define a subtype of a type $T$ as a pair of a type $V$ and an injective function of type $V \to T$. A definition of the power-type of $T$ would then be:

$$\mathsf{PwT}(T) \doteq V{:}\mathbb{U} \times \{f{:}V \to T \mid f \text{ is injective}\}$$

The basic definitions are that of a subtype of $T$ defined by a predicate $P$ of type $T \to \mathbb{P}$:

$$\{x \mid P\ x\}_T \doteq \langle y{:}T \times P\ y,\ \lambda x.\ x.1\rangle$$

and the predicate expressing membership of an element $x$ of $T$ in a subtype $S$:

$$x \in_T S \doteq \exists z{:}(S_1).\ (S_2)\ z =_T x$$

Here, the $_1$ and $_2$ postfix operators project out the first and second elements, respectively, of pairs. From these definitions it is straightforward to define constructive functions for intersection, union and complement, and to define subset and equivalent predicates. Again, as with the subtype-as-predicates approach, the set equivalence relation is weaker than the equality relation naturally associated with the $\mathsf{PwT}(T)$ type.

The advantage of this approach over predicates is that the subtype relation is now much more wide-ranging. For example, the integers are expressible as a subtype of the rationals and the rationals of the reals, even though the representing types are quite different.

### 5.3.6 Examples with Interesting Computational Content

Subset membership can be computationally interesting. For simplicity, let us assume here that a subsets-as-predicates approach is taken. Membership in subsets of a type $T$, that are generated by finite collections of elements of $T$, is almost always computationally interesting. Consider principal ideals in commutative ring theory. Using the 'subtypes as type-valued predicates' approach, the natural definition of the principal ideal generated by the element $a$ of some ring $R$ is:

$$\mathsf{PrincIdeal}(a; R) \doteq \lambda b.\exists c{:}|R|.\ c *_R a =_R b.$$

where $|R|$ is notation for the carrier of ring $R$. If some element $x \in |R|$ is in the principal ideal $\mathsf{PrincIdeal}(a; R)$, then there must be some $c \in |R|$ such that $c *_R a =_R x$. The element $c$ is part of the interesting computational content associated with $x$ being in the ideal. The equality itself might also have interesting computational content.

Another example of interesting computational content in subset membership is that of membership in the image of a class morphism between two implementations $R$ and $S$ of a class. A suitable definition of 'image' is:

$$\mathsf{Image\ of}\ f\ \mathsf{from}\ R\ \mathsf{to}\ S \doteq \lambda s.\exists r{:}|R|.\ f\ r =_S s$$

Also, membership in kernels of morphisms is computationally interesting if equality is computationally interesting.

If membership in a subset is computationally interesting, then so are many of the predicates on subsets. For example, consider the predicate stating that a subset of a group carrier is a subgroup. Also, equality relations formed using subsets (such as by the quotienting operation) are then computationally interesting.

### 5.3.7 Stability

Again, as with equality, it can often be the case that membership in a subset is not only computationally interesting, but also stable. In these cases, the computational content can be recovered as and when needed, and doesn't have to be carried around explicitly. For example, the computational content of membership in a principle ideal of a ring is stable when the ring has a division function and equality is stable. However, as with equality, there might be efficiency penalties for extracts.

### 5.3.8 Detachability

A subset is detachable if membership in it is decidable. Again, if we are careful, we need to distinguish between constructive and classical decidability.

## 5.4 Class Subtyping

In algebra, natural subtyping relationships between algebraic classes are ubiquitous. Presentations of material from algebra are far more concise and readable, if it can be assumed that the reader understands the conventions about these relationships.

Similarly, in object-oriented programming and in programming languages with abstract data types, the exploitation of subtyping relationships leads to great economy and increases in understandability of code. Similarly too, many theorem-proving environments exploit subtyping relationships to economize on theory developments and increase the clarity and reusability of theories.

Nuprl's type theory provides no built-in support for defining classes and for handling subtyping, but there are several ways in which schemes for defining classes can be set up. Before discussing these, let me for convenience give names to three kinds of subtyping relationships between classes $A$ and $B$:

- $A$ is a *set-subtype* of $B$ just when every instance of $A$ is also an instance of $B$. I use the term *set* here, because in Nuprl, set-subtypes of a class can be created by using its *set type* which provides a method for creating subtypes of a type analogous to set comprehension in set theory.

- $A$ is a *signature-subtype* of $B$ just when $A$ is a set-subtype of $B$ *and* $A$ and $B$ share the same underlying signature. The reader might be thinking that this amounts to the same thing as *set-subtyping*, but as I explain below, in Nuprl's type theory, this is sometimes a more restricted form of subtyping.

- $A$ is a *forgetful subtype* of $B$ just when there is some functor $f$ between the signatures of $A$ and $B$ that maps every instance $x$ of $A$ into some instance of $B$. I call it forgetful subtyping, because prime examples of the functor $f$ are forgetful functors. Often, such as with groups and monoids, there is an obvious functor between classes. Other times, such as with rings and monoids, there is more than one.

The simplest approach to building class definitions, is to use the $\Sigma$ type of Nuprl's type theory as illustrated earlier in this chapter. This defining of classes as sets of tuples is analogous to the usual practice in set theory. However, when mechanizing formal algebra, it leads to a need for an abundance of forgetful functors. Every time one wants to add an extra component to a class, a new signature has to be created and a forgetful functor has to be created for mapping elements of this new class into the old class. These functors would quickly create tremendous clutter. Of course, the computer could come to the rescue too. Automatic 'functor inference' could insert the functors and display technology could hide them. Still, it seems that they might still get in the way.

With this approach, the only kind of subtyping that Nuprl's type theory provides is signature subtyping. This can still be quite useful. For example, classes of unique factorization domains, integral domains, commutative rings and rings can all share the same underlying signature. Note however, if we decide that the computational content of equalities should be maintained, then virtually every class will have a distinct signature and there will be next-to-no subtyping provided by the type theory.

Some schemes of classes can be set up in Nuprl's type theory where set subtyping relationships exist between classes with different underlying signatures. The most straightforward of these assumes that there are no dependencies between components of class signatures; a class signature with components labelled $a_1 \ldots a_n$ and with component $a$ having type $T_a$ can be represented by the $\Pi$ type: $x{:}\{a_1, \ldots, a_n\} \rightarrow T_x$. Any element of this signature is also an element of any other signature gotten by removing one or more of the $a_i$ from the domain. Such signature types are essentially equivalent to the record types found in languages such as Pascal. Note that we are taking essential advantage here of the non-set-theoretic nature of functions in Nuprl.

However, this approach breaks down when the types of some components of a signature depend on those of others. What is needed is a 'highly dependent' function type where the type of a function's value on some argument is dependent on its value at other arguments. To make such a type sensible, some well-founded ordering would need to be placed on the arguments. Perhaps such a type could be added. Hickey has been looking into this possibility [Hic94].

Alternatively, a combination of $\Sigma$ types and record types could be used, since the degree of dependency in most class signatures is fairly small. In single sorted algebras, if the the predicates are treated as computationally interesting types, then there are two levels of dependency: the operator types depend on the carrier and the predicates depend on the carrier and operators. If an explicitly constructive approach is taken, then the predicates don't figure in the signature at all. In this case any algebraic class would have the structure $T{:}\mathbb{U} \times (x{:}\{a_1, \ldots, a_n\} \rightarrow T_x)$ where each $T_x$ would be dependent on $T$.

## 5.5 Approach Adopted

The implicit constructive agenda of doing mathematics constructively, but making it look like classical mathematics seems frought with difficulties. As explained above, there is computational content floating around everywhere, and it's difficult to say definitely which ought and ought not be ignored. To try preserving all computational content just in case it might be interesting seems too unwieldy an approach, and would be anyway an approach guaranteed to lead to inefficient extracts of theorems.

One way of managing the implicit computational content would be to develop a theory of *setoids* that looks superficially like a theory of sets but that keeps track of the computational content of subset membership and equalities. Other features of setoids could include

1. noting when the subset membership and equality predicates of setoids are stable and in those cases not tracking content,

2. an optional function for deciding equality.

Algebraic classes would then be defined over setoids rather than types. The idea of setoids has been explored by members of the LEGO group including Pollack, Bailey and Barthe [Bar94, Bai93].

For simplicity, I chose in the work described described in this thesis to take an explicitly constructive approach as far as class definitions are concerned. This allowed me to always use the built-in equalities associated with types for the equality relations associated with the carrier component of algebraic class instances.

Again for simplicity, I chose to use $\boldsymbol{\Sigma}$ types for building class signatures. To decrease the frequency with which I would need forgetful functors between classes, I made an effort to minimize the number of distinct class signatures; for example, in my implementation of the monoid class, I use the group signature class rather than a monoid signature class. A drawback of this approach is having to supply dummy components in instances of classes; when specifying an inhabitant of the monoid class, I have to supply a dummy inverse function. However, as described in

Section 6.3, I adopted a fairly abstract approach to defining class signatures, and defining sets of functions for projecting out components of instances. Therefore, I anticipate that in the future it would be fairly straightforward to switch to some more flexible scheme.

Defining subsets of carrier types is still a problem, since as explained above, even those approaches that ignore computational content still all define subsets with stronger equalities than desired. I did experiment with using type-valued predicates to define such notions as ideals of rings. However, I wasn't too satisfied with this work because of this awkwardness with equality, even after I had made special definitions for subset and equality relations on subsets of carriers and had set up appropriate lemmas to support rewriting with respect to these relations. This work on type-valued predicates is not reported in this thesis.

# Chapter 6

# General Algebra

## 6.1 Introduction

I present here the introductory theories I have used in my algebraic work. From a mathematical point of view the definitions and theorems are all trivial, but they do serve to illustrate the approach I have taken and do identify some of the problems involved in implementing algebra in Nuprl.

I also describe in Section 6.12 general-purpose tactics and support functions I have written to support reasoning with classes.

## 6.2 Algebraic Predicates

### 6.2.1 Predicates on Class Components

Here are notational abbreviations I made for common algebraic properties.

```
IsEqFun(T;eq) == ∀x,y:T.  ↑(x eq y) ⟺ x = y ∈ T

Ident(T;op;id)
 == ∀x:T. x op id = x ∈ T ∧ id op x = x ∈ T

Assoc(T;op)
 == ∀x,y,z:T.  x op (y op z) = (x op y) op z ∈ T

Comm(T;op) == ∀x,y:T.  x op y = y op x ∈ T

Inverse(T;op;id;inv)
 == ∀x:T. x op (inv x) = id ∈ T ∧ (inv x) op x = id ∈ T

BiLinear(T;pl;tm)
 == ∀a,x,y:T.
      a tm (x pl y) = (a tm x) pl (a tm y) ∈ T
      ∧ (x pl y) tm a = (x tm a) pl (y tm a) ∈ T
```

```
IsBilinear(A;B;C;+a;+b;+c;f)
 == (∀a1,a2:A. ∀b:B.
       (a1 +a a2) f b = (a1 f b) +c (a2 f b) ∈ C)
    ∧ (∀a:A. ∀b1,b2:B.
         a f (b1 +b b2) = (a f b1) +c (a f b2) ∈ C)

IsAction(A;x;e;S;f)
 == (∀a,b:A. ∀u:S.  (a x b) f u = a f (b f u) ∈ S)
    ∧ (∀u:S. e f u = u ∈ S)

Dist1op2opLR(A;1op;2op)
 == ∀u,v:A.
      1op (u 2op v) = (1op u) 2op v ∈ A
      ∧ 1op (u 2op v) = u 2op (1op v) ∈ A

FunThru2op(A;B;opa;opb;f)
 == ∀a1,a2:A.  f (a1 opa a2) = (f a1) opb (f a2) ∈ B

Cancel(T;S;op)
 == ∀u,v:T. ∀w:S.  w op u = w op v ∈ T ⇒ u = v ∈ T
```

The prefix notation ↑ is for a function that converts boolean-valued ($\mathbb{B}$-valued) propositions to type-valued ($\mathbb{P}$-valued) propositions. All these predicates are stable, so there is no problem unhiding them as and when necessary.

### 6.2.2 Binary relations

Here are the basic definitions I used for predicates on binary relations:

```
Refl(T;x,y.E[x; y]) == ∀a:T. E[a; a]

Sym(T;x,y.E[x; y]) == ∀a,b:T.  E[a; b] ⇒ E[b; a]

Trans(T;x,y.E[x; y])
 == ∀a,b,c:T.  E[a; b] ⇒ E[b; c] ⇒ E[a; c]

EquivRel(T;x,y.E[x; y])
 == Refl(T;x,y.E[x; y])
    ∧ Sym(T;x,y.E[x; y])
    ∧ Trans(T;x,y.E[x; y])

AntiSym(T;x,y.R[x; y])
 == ∀x,y:T.  R[x; y] ⇒ R[y; x] ⇒ x = y

StAntiSym(T;x,y.R[x; y])
 == ∀x,y:T.  ¬(R[x; y] ∧ R[y; x])
```

```
Connex(T;x,y.R[x; y])
 == ∀x,y:T.  R[x; y] ∨ R[y; x]

Preorder(T;x,y.R[x; y])
 == Refl(T;x,y.R[x; y]) ∧ Trans(T;x,y.R[x; y])

Order(T;x,y.R[x; y])
 == Refl(T;x,y.R[x; y])
   ∧ Trans(T;x,y.R[x; y])
   ∧ AntiSym(T;x,y.R[x; y])

Linorder(T;x,y.R[x; y])
 == Order(T;x,y.R[x; y]) ∧ Connex(T;x,y.R[x; y])
```

The name `StAntiSym` is short for *strictly anti-symmetric*. The other names should all be obvious. All the `R[x; y]` expressions are second-order variables with two arguments. Second-order matching and substitution is used when unfolding these definitions. For example, the instance of the `Sym` predicate:

```
Sym(ℕ;i,j.i = j ∈ ℕ)
```

unfolds to

```
∀a,b:ℕ.  a = b ∈ ℕ ⇒ b = a ∈ ℕ
```

The well-formedness lemmas for these definitions were all very straightforward. For example, the well-formedness lemma for `EquivRel` was:

```
∀T:𝕌. ∀E:T → T → ℙ. (EquivRel(T;x,y.E[x;y]) ∈ ℙ)
```

I also created an alternative set of definitions which treated relations as higher order rather than first order objects; that is, as binary (curried) functions rather than terms with arguments supplied as subterms. For example,

```
Sym(T;E) == Sym(T;x,y.E x y)
```

Standard operations of taking the reflexive closure, symmetric closure and strict part of a relation were defined on these higher-order relations:

```
E°{T} == λx,y.x = y ∈ T ∨ E x y
E⇌ == λx,y.E x y ∧ E y x
E\ == λx,y.E x y ∧ ¬(E y x)
```

Commonly, I used a display form for the reflexive closure that hid the type argument; with this display form, `E°{T}` is instead displayed as simply `E°`.

Various theorems were proven about these operators on relations, including:

```
xxorder_split:
  ∀T:𝕌. ∀R:T → T → ℙ.
    Order(T;R)
    ⇒ (∀x,y:T.  Dec((x = y ∈ T)))
    ⇒ (∀a,b:T.  R a b ⟺ R\ a b ∨ a = b ∈ T)

xxtrans_imp_sp_trans:
  ∀T:𝕌. ∀R:T → T → ℙ.  Trans(T;R) ⇒ Trans(T;R\)

refl_cl_is_order:
  ∀T:𝕌. ∀R:T → T → ℙ.
    Irrefl(T;R) ⇒ Trans(T;R) ⇒ Order(T;R°)

irrefl_trans_imp_sasym:
  ∀T:𝕌. ∀R:T → T → ℙ.
    Irrefl(T;R) ⇒ Trans(T;R) ⇒ StAntiSym(T;R)

xxconnex_iff_trichot:
  ∀T:𝕌. ∀R:T → T → ℙ.
    (∀a,b:T.  Dec((R a b)))
    ⇒ (Connex(T;R) ⟺ {∀a,b:T.  R\ a b ∨ R⇌ a b ∨ R\ b a})

xxconnex_iff_trichot_a:
  ∀T:𝕌. ∀R:T → T → ℙ.
    Connex(T;R°) ⟺ (∀a,b:T.  R a b ∨ a = b ∈ T ∨ R b a)
```

The `xx` prefix on some of the lemma names here was to distinguish these lemmas from similar ones that involved predicates treating binary relations as first order.

While this higher order approach was mathematically more elegant and less verbose, it did have a few minor drawbacks: firstly, when reasoning about equivalence of relations, I had to define and work with a new equivalence relation:

```
binrel_eqv:
  E <≡>{T} E' ==  ∀x,y:T.  E x y ⟺ E' x y
```

This wasn't a serious problem since I designed the rewrite package to easily cope with handling new relations. Secondly, the convention adopted in all previously existing theories had been to define equivalence relations in a first-order style, and it was awkward to have to switch back and forth between styles. Thirdly, unfolding of definitions of specific higher-order relations required extra $\beta$-reduction steps. Fourthly, I had less flexibility with defining display forms for particular relations; the positions of a relation's arguments were determined by the display forms for binary function application, rather than the display form for the relation. With the first-order approach, I could define a display form so that an instance of a *mod* relation would be displayed as `a = b mod n`, whereas with the higher order approach I would have had to settle for maybe `a ={mod n} b`.

## 6.3 Discrete and Ordered Sets

I found it convenient to package a type with a boolean-valued equality function and an optional boolean-valued inequality relation. I introduced a signature class of all such packaged types with the ML library object shown in Figure 6.1. All

```
create_poset_sig:
  % (p)artially (o)rdered (set) (sig)nature.
    Includes equality function for discrete sets.%

  Class Declaration for: p ∈ PosetSig

    Long Name: poset_sig
    Short Name: set

    Parameters:


    Fields:
       (|p|)  car : 𝕌
       (=_b p)  eq : car → car → 𝔹
       (≤_b p)  le : car → car → 𝔹

    Universe: 𝕌'
```

Figure 6.1: Declaration for `PosetSig` Class

the text here from the `Class Declaration` down is generated by a display form for a call to an ML function. The first time this function is called in a given declaration, it creates definitions for a class signature and projection functions for each component of class instances. In this case, the definitions created are

```
poset_sig:
  PosetSig == car:𝕌 × eq:(car → car → 𝔹) × (car → car → 𝔹)

set_car:
  |p| == p.1

set_eq:
  =_b p == p.2.1

set_le:
  ≤_b p == p.2.2
```

Note that PosetSig has a level expression argument. I have set up display forms so that if this argument is not the level variable i, then it is explicitly displayed. For example, if it is j, then the class is displayed as PosetSig{j}. This convention is an extension of that adopted for universe terms in Section 2.1.2. I have used this convention for the displays of all class definitions.

Well-formedness lemmas are created automatically by class declarations. In this case, they were:

```
poset_sig_wf:
  PosetSig ∈ 𝕌'

set_car_wf:
  ∀p:PosetSig. |p| ∈ 𝕌

set_eq_wf:
  ∀p:PosetSig. =_b p ∈ |p| → |p| → 𝔹

set_le_wf:
  ∀p:PosetSig. ≤_b p ∈ |p| → |p| → 𝔹
```

Finally, class declarations create conversions for reducing the projection functions when they are applied to tuples. These conversions are automatically added to the to the AbReduceC conversion described in Section 4.6. Class declarations are stored in ML objects in theories to document classes and to ensure that the appropriate conversions are created each time the theory is loaded. As indicated above, a Class declaration only creates the set of definitions and theorems for a class once. Afterwards, these definitions and theorems are dumped to files and loaded from files along with the rest of the theory that they reside in.

The terms in parentheses at the start of the lines in the Fields section of a class declaration show the display forms adopted for each field component. For clarity in class definitions, the argument of every projection function is shown, but frequently it is useful to hide this argument. These terms and the term to the right of the Class Declaration heading effectively have the status of comments. When the Class Declaration abstraction is expanded to reveal the call to the underlying class declaration ML function, these terms disappear. I happen to have put these 'comments' in by hand, but it would be easy to have them added automatically.

With some class declarations (see Section 10.3 for examples), the field terms displays are not shown as comments. In these cases, the displays for the projection functions are the default displays. The default displays always consist of a postfix period, followed by the name of the field. For example, the default display for the eq field projection function applied to an instance p of the PosetSig class is p.eq.

I created several subtypes of the PosetSig class.

```
dset:
  DSet == {s:PosetSig| IsEqFun(|s|;=_b s)}
```

```
qoset:
  QOSet == {s:DSet| Preorder(|s|;a,b.a ≤s b)}

poset:
  POSet == {s:QOSet| AntiSym(|s|;a,b.a ≤s b)}

loset:
  LOSet == {s:POSet| Connex(|s|;x,y.x ≤s y)}
```

The prefixes to the class names stand for *discrete*, *quasi-ordered*, *partially-ordered*, and *linearly-ordered.*

I introduced a definition for the strict order function corresponding to the reflexive one provided by the PosetSig, and also found it convenient to define propositional (type-valued) versions of both order relations.

```
set_blt:
  a <_b p b == (a ≤_b p b) ∧_b ¬_b (b ≤_b p a)

set_leq:
  a ≤p b == ↑(a ≤_b p b)

set_lt:
  a <p b == ↑(a <_b p b)
```

Here for clarity I show the parameter p to these definitions, but often I hide it. When it is hidden, its value can always be inferred by the reader by looking at the types of the relation arguments.

I introduced lemmas about these relations such as:

```
set_lt_transitivity_1:
  ∀s:QOSet. ∀a,b,c:|s|.  a ≤s b ⇒ b <s c ⇒ a <s c

loset_trichot:
  ∀s:LOSet. ∀a,b:|s|.  a <s b ∨ a = b ∈ |s| ∨ b <s a

set_leq_complement:
  ∀s:LOSet. ∀a,b:|s|.  ¬(a ≤s b) ⟺ b <s a
```

This need to sometimes have both boolean *and* propositional versions of predicates is an unavoidable feature of Nuprl's constructive type theory.

For introducing instances of these classes, I added the definitions

```
mk_dset:
  mk_dset(T, eq) == <T, eq, eq>

mk_oset:
  mk_oset(T, eq, leq) == <T, eq, leq>
```

and created lemmas that characterized when they constructed instances of the various classes above. For instance, one lemma for `mk_oset` was:

```
mk_oset_wf:
  ∀T:𝕌. ∀eq,leq:T → T → 𝔹.
    IsEqFun(T;eq)
    ⇒ Linorder(T;a,b.↑(a leq b))
    ⇒ mk_oset(T;eq;leq) ∈ LOSet
```

Such definitions and lemmas are inessential, but with them, the internal implementation details of class declarations are localized to the theory where they are introduced.

## 6.4 Monoids and Groups

I made monoids and groups share the same underlying signature, to avoid having to create explicit coercion functions between them. The underlying class signature `GrpSig` is introduced by the class declaration shown in Figure 6.2. With the

```
Class Declaration for: g ∈ GrpSig

  Long Name: grp_sig
  Short Name: grp

  Parameters:


  Fields:
     (|g|)  car : 𝕌
     (=_b g)  eq : car → car → 𝔹
     (≤_b g)  le : car → car → 𝔹
     (*g)  op : car → car → car
     (eg)  id : car
     (~g)  inv : car → car

  Universe: 𝕌'
```

Figure 6.2: Declaration for `GrpSig` Class

definitions

```
IsMonoid(T;op;id) == Assoc(T;op) ∧ Ident(T;op;id)

IsGroup(T;op;id;inv) == IsMonoid(T;op;id) ∧ Inverse(T;op;id;inv)
```

the definitions of subclasses of GrpSig are:

```
IMonoid == {g:GrpSig| IsMonoid(|g|;*g;eg)}

Mon == {g:GrpSig| IsMonoid(|g|;*g;eg) ∧ IsEqFun(|g|;=_b g)}

IAbMonoid == {g:IMonoid| Comm(|g|;*g)}

AbMon == {g:Mon| Comm(|g|;*g)}

IGroup == {g:IMonoid| Inverse(|g|;*g;eg;∼g)}

Group == {g:Mon| Inverse(|g|;*g;eg;∼g)}

IAbGrp == {g:IGroup| Comm(|g|;*g)}

AbGrp == {g:Group| Comm(|g|;*g)}
```

The I prefix stands for indiscrete, since instances of these classes don't have their eq components constrained to agree with the equality relation associated with the carrier of the instances.

Obvious theorems about groups include:

```
grp_op_cancel_l:
  ∀g:IGroup. ∀a,b,c:|g|.  a * b = a * c ⇒ b = c

grp_inv_id:
  ∀g:IGroup. ∼ e = e

grp_inv_inv:
  ∀g:IGroup. ∀a:|g|.  ∼ (∼ a) = a
```


```
grp_inv_id:
  ∀g:IGroup. ∼ e = e

grp_inv_inv:
  ∀g:IGroup. ∀a:|g|.  ∼ (∼ a) = a
```


```
grp_inv_assoc:
  ∀g:IGroup. ∀a,b:|g|.
    a * ((∼ a) * b) = b ∧ (∼ a) * (a * b) = b
```


```
grp_op_cancel_l:
  ∀g:IGroup. ∀a,b,c:|g|.  a * b = a * c ⇒ b = c
```


```
grp_inv_diff:
  ∀g:IGroup. ∀a,b:|g|.  ∼ (a * (∼ b)) = b * (∼ a)
```

Note that here I have switched to suppressing the arguments of the group projection functions and the equality propositions.

A forgetful functor mapping from GrpSig to PosetSig is

```
g↓set == <|g|, =_b g, ≤_b g>
```

## 6.5 Ordered Monoids and Groups

In the work described in Chapter 10, several definitions of ordered monoids and groups were needed. Here are the class definitions:

```
OMon == {g:AbMon| Linorder(|g|;x,y.↑(x ≤_b g y))}

OCMon
 == {g:AbMon|
      Linorder(|g|;x,y.↑(x ≤_b g y))
      ∧ Cancel(|g|;|g|;*g)
      ∧ (∀z:|g|. Monot(|g|;x,y.↑(x ≤_b g y);λw.z *g w))}

OGrp == {g:OCMon| Inverse(|g|;*g;eg;∼g)}
```

where

```
Monot(T;x,y.R[x; y];f)
 == ∀x,y:T.  R[x; y] ⇒ R[(f x); (f y)]
```

As with the set class, several order relation definitions were introduced:

```
grp_blt:
  a <_b g b == a <_b g↓oset b

grp_lt:
  a <g b == a <g↓set b

grp_leq:
  a ≤g b == ↑(a ≤_b g b)
```

The definitions `grp_lt` and `grp_blt` were defined in terms of their set counterparts to simplify the specialization of set theorems about order relations to corresponding group theorems. Section 6.12.2 discusses lemma specialization. The theorem

```
grp_lt_trichot:
  ∀g:OCMon. ∀a,b:|g|.  a <g b ∨ a = b ∈ |g| ∨ b <g a
```

is an example of the specialization of the set theorem `set_lt_trichot` given above.

Theorems proven involving both the group operation and an order relation include:

```
grp_lt_op_l:
  ∀g:OGrp. ∀a,b,c:|g|.  a < b ⟺ c * a < c * b

grp_op_polarity:
  ∀g:OGrp. ∀a,b:|g|.  e ≤ a ⇒ e ≤ b ⇒ e ≤ a * b
```

## 6.6 Half Groups

The notion of a half group of a linearly-ordered group turned out to be a useful one in the work described in Chapter 10. The definitions are

```
hgrp_car:
  |g|+ == {x:|g|| eg ≤g x}

hgrp_of_ogrp:
  g↓hgrp == <|g|+, =_b g, ≤_b g, *g, eg, λx.x>
```

and typing lemmas are

```
hgrp_car_wf:
  ∀g:GrpSig. |g|+ ∈ 𝕌

hgrp_of_ogrp_wf2:
  ∀g:OGrp. g↓hgrp ∈ OCMon
```

Note that in forming the half group, the group inverse operation ~ cannot be used in the inverse slot of the half-group tuple, since it is not in general closed on the half-group domain.

The non-negative integers under addition are the half group of the group of integers under addition. Similarly, the non-negative rationals under addition are a half group.

## 6.7 Rings

The signature class for rings is given in Figure 6.3. here, the notation `?A` is a notational abbreviation for the type `A + Unit`.

The definitions of the classes of discrete rings and discrete commutative rings are

```
  IsRing(T;plus;zero;neg;times;one)
   == IsGroup(T;plus;zero;neg)
      ∧ IsMonoid(T;times;one)
      ∧ BiLinear(T;plus;times)

  Rng
   == {r:RngSig|
      IsRing(|r|;+r;0r;-r;*r;1r) ∧ IsEqFun(|r|;=_b r)}

crng:
  CRng == {r:Rng| Comm(|r|;*r)}
```

```
Class Declaration for: r ∈ RngSig

  Long Name: rng_sig
  Short Name: rng

  Parameters:


  Fields:
     (|r|)  car : 𝕌
     (=_b r)  eq : car → car → 𝔹
     (≤_b r)  le : car → car → 𝔹
     (+r)  plus : car → car → car
     (0r)  zero : car
     (-r)  minus : car → car
     (*r)  times : car → car → car
     (1r)  one : car
     (÷r)  div : car → car → ?car

  Universe: 𝕌'
```

Figure 6.3: Declaration for RngSig Class

Plenty of elementary theorems about rings were proven. It probably suffices for the reader to know that these exist. Definitions of forgetful functors from the `RngSig` class to the `GrpSig` class are:

```
mul_mon_of_rng:
  r↓xmn == <|r|, =_b r, ≤_b r, *r, 1r, λz.z>

add_grp_of_rng:
  r↓+gp == <|r|, =_b r, ≤_b r, +r, 0r, -r>
```

and example typing lemmas for these are

```
add_grp_of_rng_wf:
  ∀r:RngSig. r↓+gp ∈ GrpSig

add_grp_of_rng_wf_b:
  ∀r:Rng. r↓+gp ∈ AbGrp

mul_mon_of_rng_wf:
  ∀r:RngSig. r↓xmn ∈ GrpSig

mul_mon_of_rng_wf_a:
  ∀r:Rng. r↓xmn ∈ Mon

mul_mon_of_rng_wf_b:
  ∀r:CRng. r↓xmn ∈ AbMon
```

## 6.8 Modules and Algebras

The signature class for modules and algebras is given in Figure 6.4.

The definitions of module, algebra and commutative algebra classes are:

```
A-Module
 == {m:AlgebraSig(|A|)|
      IsGroup(|m|;+m;0m;-m)
      ∧ Comm(|m|;+m)
      ∧ IsAction(|A|;*A;1A;|m|;·m)
      ∧ IsBilinear(|A|;|m|;|m|;+A;+m;+m;·m)
      ∧ IsEqFun(|m|;m.eq)}

A-Algebra
 == {m:A-Module|
      IsMonoid(|m|;xm;1m)
      ∧ BiLinear(|m|;+m;xm)
      ∧ (∀a:|A|. Dist1op2opLR(|m|;·m a;xm))}
```

```
Class Declaration for: a ∈ AlgebraSig(A)

  Long Name: algebra_sig
  Short Name: alg

  Parameters:
     A : 𝕌

  Fields:
     (|a|)  car : 𝕌
     (a.eq)  eq : car → car → 𝔹
     (a.le)  le : car → car → 𝔹
     (+a)  plus : car → car → car
     (0a)  zero : car
     (-a)  minus : car → car
     (xa)  times : car → car → car
     (1a)  one : car
     (÷a)  div : car → car → ?car
     (·a)  act : A → car → car

  Universe: 𝕌'
```

Figure 6.4: Declaration for `AlgebraSig(A)` Class

```
calgebra:
  A-CAlgebra == {m:A-Algebra| Comm(|m|;xm)}
```

and forgetful functors are:

```
rng_of_alg:
  a↓rg == <|a|, a.eq, a.le, +a, 0a, -a, xa, 1a, ÷a>

grp_of_module:
  m↓grp == m↓rg↓+gp
```

Example typing lemmas for these are:

```
grp_of_module_wf2:
  ∀a:RngSig. ∀m:a-Module.  m↓grp ∈ AbGrp

rng_of_alg_wf2:
  ∀a:CRng. ∀m:a-Algebra.  m↓rg ∈ Rng
```

The definitions for module and algebra homomorphisms are:

```
module_hom_p:
  ∀a:RngSig. ∀m,n:AlgebraSig(|a|). ∀f:|m| → |n|.
    IsModuleHom{a,m,n}(f)
    = (FunThru2op(|m|;|n|;+m;+n;f)
      ∧ (∀u:|a|. fun_thru_1op(|m|;|n|;·m u;·n u;f)))
    ∈ ℙ

module_hom:
  ∀A:RngSig. ∀M,N:AlgebraSig(|A|).
    A-ModuleHom(M;N)
    = {f:|M| → |N|| IsModuleHom{A,M,N}(f)}
    ∈ 𝕌

alg_hom_p:
  ∀a:RngSig. ∀m,n:AlgebraSig(|a|). ∀f:|m| → |n|.
    IsAlgHom{a,m,n}(f)
    = (IsModuleHom{a,m,n}(f)
      ∧ FunThru2op(|m|;|n|;xm;xn;f)
      ∧ f 1m = 1n ∈ |n|)
    ∈ ℙ

algebra_hom:
  ∀A:RngSig. ∀M,N:AlgebraSig(|A|).
    A-AlgebraHom(M;N)
    = {f:A-ModuleHom(M;N)|
       FunThru2op(|M|;|N|;xM;xN;f) ∧ f 1M = 1N ∈ |N|}
    ∈ 𝕌
```

As explained in Section 10.2, these are *typed definitions.*

## 6.9 Common Instances of Algebraic Classes

Typing lemmas for standard class instances that were used elsewhere in this thesis included:

```
band_mon_wf:
  <𝔹,∧_b > ∈ AbMon{1}

bor_mon_wf:
  <𝔹,∨_b > ∈ AbMon{1}

int_add_grp_wf2:
  <ℤ+> ∈ OGrp{1}

lapp_mon_wf:
  ∀s:DSet. <s List, @> ∈ Mon
```

In `lapp_mon_wf`, the `@` symbol stands for the list append function. Its definition was:

```
append:
  as @ bs
   ==r case as of [] => bs | a::as' => a::(as' @ bs) esac
```

The definition of the equality function that made `<s List, @>` discrete was:

```
eq_list_ml:
  as =_b s bs
   ==r case as of
         [] => null(bs)
         a::as' => case bs of
                     [] => ff
                     b::bs' => a =_b s b ∧_b  as' =_b s bs'
                   esac
       esac
```

## 6.10 Iterated Operations on Integer Segments

### 6.10.1 Definitions

Being able to iterate a binary operation over a finite sequence of values is fundamental in mathematics. This theory can be developed generally either over arbitrary monoids or over arbitrary semigroups. When over arbitrary semigroups, the finite sequences must be non-empty. There is no such restriction with monoids. Since monoid structures were much more common than non-monoidal semigroup structures in the data-types I wanted to consider, I chose to develop this theory first over monoids.

Examples of monoid structures are abundant and include the booleans with $\wedge_b$ and $\vee_b$, the integers, rationals and reals, with + and *, and lists with the append operation.

Examples of non-monoidal semigroup structures include the integers with binary max and min functions, though the naturals with max do form a monoid.

In this section I describe functions I set up for iterating over sequences indexed by ranges of integers. In Section 6.11, I describe similar functions for working on lists.

Note that I use the terms 'iterated operator', 'product' and 'sum' interchangeably when discussing iterating a monoid operation on a finite sequence. Sometimes too I used the phrase 'general sum' and 'general product'. When dealing with rings and modules and algebras, I use 'sum' and 'product' in their normal senses to refer to iterating the 'plus' and 'times' operations.

The mon_itop operation iterator takes the product of a sequence of elements from a monoid. It was defined in two stages:

```
Π(op,id) lb ≤ i < ub. E[i]
 ==rif lb <z ub
    then Σ(op,id) lb ≤ i < ub - 1. E[i] op E[(ub - 1)]
    else id
    fi

Π(g) lb ≤ i < ub. E[i] == Π(*g,eg) lb ≤ i < ub. E[i]
```

The typing lemma for mon_itop was

```
∀g:IMonoid. ∀p,q:ℤ. ∀E:{p..q⁻} → |g|.
  Π(g) p ≤ i < q. E[i] ∈ |g|
```

where the notation {p..q⁻} was introduced by the definition

```
int_seg:
  {i..j⁻} == {k:ℤ| i ≤ k < j}
```

Note how {p..q⁻} was used for the domain of the indexed expression E. In type theories of total functions without subtyping such as HOL's, such a function would have to be defined over the whole of the naturals or integers, and often a default value would have to be supplied for E on indices that are out of some normal range of consideration.

### 6.10.2 Theorems

A variety of simple useful lemmas were proven about these iterated operators. For example:

```
mon_itop_unroll_base:
  ∀g:IMonoid. ∀i,j:ℤ.
    i = j
    ⇒ (∀E:{i..j⁻} → |g|. Πg i ≤ k < j. E[k] = eg ∈ |g|)

mon_itop_unroll_unit:
  ∀g:IMonoid. ∀i,j:ℤ.
    i + 1 = j
    ⇒ (∀E:{i..j⁻} → |g|. Πg i ≤ k < j. E[k] = E[i] ∈ |g|)

mon_itop_unroll_hi:
  ∀g:IMonoid. ∀i,j:ℤ.
    i < j
    ⇒ (∀E:{i..j⁻} → |g|
          Πg i ≤ k < j. E[k]
          = (Πg i ≤ k < j - 1. E[k]) *g E[j - 1]
          ∈ |g|)

mon_itop_unroll_lo:
  ∀g:IMonoid. ∀i,j:ℤ.
    i < j
    ⇒ (∀E:{i..j⁻} → |g|
          Πg i ≤ k < j. E[k]
          = E[i] *g (Πg i + 1 ≤ k < j. E[k])
          ∈ |g|)

mon_itop_shift:
  ∀g:IMonoid. ∀a,b:ℤ.
    a ≤ b
    ⇒ (∀E:{a..b⁻} → |g|. ∀k:ℤ.
          Πg a ≤ j < b. E[j]
          = Πg a + k ≤ j < b + k. E[j - k]
          ∈ |g|)

mon_itop_split:
  ∀g:IMonoid. ∀a,b,c:ℤ.
    a ≤ b
    ⇒ b ≤ c
    ⇒ (∀E:{a..c⁻} → |g|
          Πg a ≤ j < c. E[j]
          = (Πg a ≤ j < b. E[j]) *g (Πg b ≤ j < c. E[j])
          ∈ |g|)
```

```
mon_itop_split_el:
  ∀g:IMonoid. ∀a,b,c:ℤ.
    a ≤ b
    ⇒ b < c
    ⇒ (∀E:{a..c⁻} → |g|
          Πg a ≤ j < c. E[j]
          = (Πg a ≤ j < b. E[j])
            *g (E[b] *g (Πg b + 1 ≤ j < c. E[j]))
          ∈ |g|)

mon_itop_op:
  ∀g:IAbMonoid. ∀a,b:ℤ.
    a ≤ b
    ⇒ (∀E,F:{a..b⁻} → |g|.
          Πg a ≤ i < b. E[i] *g F[i]
          = (Πg a ≤ i < b. E[i]) *g (Πg a ≤ i < b. F[i])
          ∈ |g|)
```

When working in a mechanized formal environment, one dilemma with choosing group notation is deciding whether to use additive or multiplicative notation. The notation that is presented in this thesis is not quite consistent; fields of the group class are written multiplicatively, yet the natural action (see Section 6.10.3) is written as if groups were additive. A simple extension that will be made in the near future to Nuprl's current display algorithm is one that will allow the user to group display forms into named blocks and then enable or disable blocks all at once. With this extension, it will be possible to switch notations in seconds and assign preferred notation blocks to theories. With notation blocks for additive and multiplicative group notation, we should have a more acceptable solution to this notation problem for groups.

### 6.10.3 Natural and Integer Actions

The `mon_itop` operator was used to define natural and integer action (or exponentiation) operators:

```
nat_op:
  n x(op;id) a == Σ(op,id) 0 ≤ i < n. a

mon_nat_op:
  n ·g a == n x(*g;eg) a

int_op:
  i x(op;id;inv) a
  == if 0 ≤z i then i x(op;id) a else inv -i x(op;id) a fi
```

```
grp_int_op:
  i ·g a == n x(*g;eg;~g) a
```

Theorems proved about these operators included

```
mon_nat_op_op:
  ∀g:IAbMonoid. ∀n:ℕ. ∀a,b:|g|.
    (n ·g (a *g b)) = (n ·g a) *g (n ·g b) ∈ |g|

mon_nat_op_add:
  ∀g:IMonoid. ∀e:|g|. ∀a,b:ℕ.
    ((a + b) ·g e) = (a ·g e) *g (b ·g e) ∈ |g|

mon_nat_op_mul:
  ∀g:IMonoid. ∀m,n:ℕ. ∀e:|g|.
    (n ·g (m ·g e)) = ((n * m) ·g e) ∈ |g|

mon_nat_op_hom_swap:
  ∀g,h:IMonoid. ∀f:MonHom(g,h). ∀n:ℕ. ∀u:|g|.
    (n ·h (f u)) = f (n ·g u) ∈ |h|
```

### 6.10.4 Binomial Theorem

The binomial theorem is an example of a theorem that used a summation operator on rings, and natural actions over both the additive and multiplicative monoid of rings. I proved it in Nuprl as a simple exercise. The statement of it was

```
binomial:
  ∀r:CRng. ∀a,b:|r|. ∀n:ℕ.
    (a + b) ↑ n
    = Σ 0 ≤ i < n + 1.
       choose(n;i) · ((a ↑ i) * (b ↑ (n - i)))
```

Here, the · was the natural action the additive monoid of the ring r and ↑ was the natural action on the multiplicative monoid. The definition of choose was

```
  choose(n;i)
   ==r if (i =z 0) ∨b (i =z n)
       then 1
       else choose(n - 1;i - 1) + choose(n - 1;i)
       fi
```

## 6.11 Iterated Operations on Lists

### 6.11.1 Definitions and Basic Theorems

Sometimes, it was more convenient to work with finite sequences represented using lists rather than functions on ranges of integers. I first made the definitions:

```
reduce:
  reduce(f;k;as)
   ==r case as of [] => k | a::as' => f a reduce(f;k;as') esac

mon_reduce:
  Π(m) as == reduce(*m;em;as)
```

with the corresponding typing lemmas:

```
reduce_wf:
  ∀A,B:𝕌. ∀f:A → B → B. ∀k:B. ∀as:A List.
    reduce(f;k;as) ∈ B

mon_reduce_wf:
  ∀g:IMonoid. ∀as:|g| List.  Π(g) as ∈ |g|
```

Sometimes I suppressed the monoid argument to the `mon_reduce` operator when the monoid was obvious from context.

As is done in NQTHM [BM88a], and similar in style to the product operation `mon_itop` defined in Section 6.10, I found it very useful to define a `For` binding product operation over lists.

```
for:
  For{T,op,id} x ∈ as. f[x]
  == reduce(op;id;map(λx:T. f[x];as))

mon_for:
  For{T,g} x ∈ as. f[x] == For{T,*g,eg} x ∈ as. f[x]
```

The typing lemmas for these are

```
for_wf:
  ∀A,B,C:𝕌. ∀f:B → C → C. ∀k:C. ∀as:A List. ∀g:A → B.
    (For{A,f,k} x ∈ as. g[x]) ∈ C

mon_for_wf:
  ∀g:IMonoid. ∀A:𝕌. ∀as:A List. ∀f:A → |g|.
    (For{A,g} x ∈ as. f[x]) ∈ |g|
```

I showed the views of summation presented in this and the previous chapter to be equivalent

```
mon_reduce_eq_itop:
  ∀g:IMonoid. ∀as:|g| List.
    Π(g) as = Πg 0 ≤ i < ||as||. as[i] ∈ |g|
```

and proved basic lemmas about `mon_reduce` and `mon_for`, including

```
mon_reduce_append:
  ∀g:IMonoid. ∀as,bs:|g| List.
    Π(g) as @ bs = Π(g) as *g Π(g) bs ∈ |g|

mon_for_append:
  ∀g:IMonoid. ∀A:𝕌. ∀f:A → |g|. ∀as,as':A List.
    (For{A,g} x ∈ as @ as'. f[x])
    = (For{A,g} x ∈ as. f[x]) *g (For{A,g} x ∈ as'. f[x])
    ∈ |g|
```

### 6.11.2 Commutative Operations

Commonly, the binary operation used for the summing operation is commutative, and in this case, many more algebraic facts are true about `For`. For example, I proved

```
mon_for_functionality_wrt_permr:
  ∀g:IAbMonoid. ∀A:𝕌. ∀as,as':A List. ∀f,f':A → |g|.
    (as ≡(A) as')
    ⇒ (∀x:A. mem_f(A;x;as) ⇒ f[x] = f'[x] ∈ |g|)
    ⇒ (For{A,g} x ∈ as. f[x])
       = (For{A,g} x ∈ as'. f'[x])
       ∈ |g|

mon_for_map:
  ∀g:IAbMonoid. ∀A,B:𝕌. ∀e:A → B. ∀f:B → |g|. ∀as:A List.
    (For{B,g} y ∈ map(e;as). f[y])
    = (For{A,g} x ∈ as. f[e x])
    ∈ |g|

mon_for_of_op:
  ∀g:IAbMonoid. ∀A:𝕌. ∀e,f:A → |g|. ∀as:A List.
    (For{A,g} x ∈ as. e[x] *g f[x])
    = (For{A,g} x ∈ as. e[x]) *g (For{A,g} x ∈ as. f[x])
    ∈ |g|

mon_for_of_id:
  ∀g:IAbMonoid. ∀A:𝕌. ∀as:A List.
    (For{A,g} x ∈ as. eg) = eg ∈ |g|

mon_for_swap:
  ∀g:IAbMonoid. ∀A,B:𝕌. ∀f:A → B → |g|. ∀as:A List.
  ∀bs:B List.
    (For{A,g} x ∈ as. For{B,g} y ∈ bs. f[x;y])
    = (For{B,g} y ∈ bs. For{A,g} x ∈ as. f[x;y])
    ∈ |g|
```

The ≡(A) relation in the first theorem is the permutation relation on lists. It is described in Section 7.3.1.

### 6.11.3 Iterating over Heads and Tails

A variant on `For` called `HTFor` was defined that gives the expression being summed access to both the head and tail of each position in the sequence being summed over, rather than just the head. Its definition and definitions of auxiliary functions were:

```
mapconsl:
  mapcons(f;as)
  ==r case as of
        [] => []
        a::as' => (f a as')::mapcons(f;as')
      esac

for_hdtl:
  ForHdTl{A,f,k} h::t ∈ as. g[h; t]
  == reduce(f;k;mapcons(λh,t.g[h; t];as))

mon_htfor:
  HTFor{A,g} h::t ∈ as. f[h; t]
  == ForHdTl{A,*,e} h::t ∈ as. f[h; t]
```

and the corresponding typing lemmas were:

```
mapcons_wf:
  ∀A,B:𝕌. ∀f:A → A List → B. ∀l:A List.
    mapcons(f;l) ∈ B List

for_hdtl_wf:
  ∀A,B,C:𝕌. ∀f:B → C → C. ∀k:C. ∀as:A List. ∀g:A
                                                → A List
                                                → B.

mon_htfor_wf:
  ∀g:IMonoid. ∀A:𝕌. ∀as:A List. ∀f:A → A List → |g|.
    (HTFor{A,g} h::t ∈ as. f[h;t]) ∈ |g|
```

`HTFor` was used to define predicates charactering lists as ordered and as having distinct elements.

### 6.11.4 Conditional Iterated Operations

Frequently a binary operation is iterated over a sequence of elements subject to some condition being true of the elements. Rather than introduce a separate version of `For` to allow this, I took advantage of the fact that `For` summed over monoids and defined a simple construct I called `when`. The definition was

```
mon_when:
  when{g} b . p == if b then p else eg fi
```

and the typing lemma was

```
mon_when_wf:
  ∀g:IMonoid. ∀b:𝔹. ∀p:|g|.  when{g} b . p ∈ |g|
```

Basic lemmas proved about `when` included:

```
mon_when_of_id:
  ∀g:IMonoid. ∀b:𝔹.  when b. e = e

mon_when_thru_op:
  ∀g:IMonoid. ∀b:𝔹. ∀p,q:|g|.
    when b. p * q = (when b. p) * (when b. q)

mon_when_swap:
  ∀g:Mon. ∀b,b':𝔹. ∀p:|g|.
    when b. when b'. p = when b'. when b. p

mon_when_when:
  ∀g:Mon. ∀b,b':𝔹. ∀p:|g|.
    when b. when b'. p = when b ∧_b  b'. p
```

Lemmas about the interaction of `when` and `For` included:

```
mon_for_when_swap:
  ∀g:Mon. ∀A:𝕌. ∀as:A List. ∀b:𝔹. ∀f:A → |g|.
    (For{g} x ∈ as. when b. f[x])
    = when b. (For{g} x ∈ as. f[x])

mon_for_when_none:
  ∀s:DSet. ∀g:IMonoid. ∀f:|s| → |g|. ∀b:|s| → 𝔹.
  ∀as:|s| List.
    (∀x:|s|. ↑(x ∈_b  as) ⇒ ¬↑b[x])
    ⇒ (For{g} x ∈ as. when b[x]. f[x]) = e
```

```
mon_for_when_unique:
  ∀s:DSet. ∀g:IMonoid. ∀f:|s| → |g|. ∀b:|s| → 𝔹. ∀u:|s|.
    ↑b[u]
    ⇒ (∀as:|s| List
          ↑distinct{s}(as)
          ⇒ ↑(u ∈_b as)
          ⇒ (∀v:|s|. ↑b[v] ⇒ ↑(v ∈_b as) ⇒ v = u)
          ⇒ (For{g} x ∈ as. when b[x]. f[x]) = f[u])
```

The theorem `mon_for_when_unique` described exactly when a `For` and a `when` cancel. The preconditions of this theorem essentially say that the sequence being summed over must contain all distinct elements and that the predicate `b` must be true at exactly one of those elements.

The definition of `distinct` was:

```
distinct:
  distinct{s}(ps)
  == HTFor{|s|,<𝔹,∧_b >} q::qs ∈ ps. ∀_b r(:|s|) ∈ qs. ¬_b (r =_b q)
```

The function `HTFor` is described in Section 6.11.3.

### 6.11.5 Specializations

Specializations of the `For` operation to concrete domains included:

```
ball:
  ∀_b x(:A) ∈ as. f[x] == For{A,<𝔹,∧_b >} x ∈ as. f[x]

bexists:
  ∃_b x(:A) ∈ as. f[x] == For{A,<𝔹,∨_b >} x ∈ as. f[x]
```

Specializations to summation operations on the ring and algebra classes was done for the multiset version of `For`. Specializations to product operations would have been trivial, but were not needed for the work described in this thesis.

### 6.11.6 Iterated Operations Indexed by Multisets

Since the order of elements being summed up by the `For` operator is irrelevant when summing with a commutative binary operation, it is as natural to think of the elements as coming from a multiset rather than a sequence. In the development of finite multisets and sets (described in Chapter 9), I introduced a variant on `For` I called `msFor` that drew indices from a multiset rather than from a list. Since multisets there were implemented as lists, `msFor` was nothing other than a retyping of the `For` operator. Its definition, typing lemma and proof of typing lemma are all given in Section 9.3.2, along with several basic theorems about it corresponding to theorems given above for `For`.

This multiset summation operator and its specializations to summation operators over rings and modules were used extensively in the ADT case study on polynomial arithmetic described in Chapter 10. Relevant lemmas about the multiset summation operators can be found there.

## 6.12 ML Support

### 6.12.1 Simplifying Algebraic Expressions

Normal forms can be described for expressions constructed from the operators of a class instances and it is often straightforward to design normalization procedures. I created rewrite conversions for normalizing expressions over most of the classes described above. These conversions were made generic by supplying conversions for basic rewrite rules as arguments. Several examples are as follows. The examples hopefully illustrate the elegance of the notion of *conversion*.

The function

```
  RAssocC : convn -> convn
```

creates conversions for right-associating trees built from binary associative operators. Its definition is:

```
let RAssocC AssocC = TryC (SweepDnC (RepeatC AssocC)) ;;
```

where `AssocC` is assumed to have behavior

`AssocC` $(a\ +\ b)\ +\ c\ \longrightarrow\ a\ +\ (b\ +\ c)$ .

`RAssocC` works top-down repeatedly applying the `AssocC` conversion at each level until the left child is no longer an instance of the binary operator.

The function `BubbleSortC`

```
BubbleSortC
  EndSwapC : convn
  InsideSwapC : convn
  dest_op : term -> (term # term)
  tm_lt : (term # term) -> bool
  =
  c : convn
```

constructs a conversion for normalizing expressions built from a associative commutative operator. The arguments `EndSwapC` and `InsideSwapC` are assumed to have behaviours

`EndSwapC` $a + b \longrightarrow b + a$
`InsideSwapC` $a + (b + c) \longrightarrow b + (a + c)$ ,

`dest_op` is assumed to be a destructor function for the binary operator and `tm_lt` is assumed to be a total term order on terms. `BubbleSortC` assumes that the expressions are already right associated. It maintains the right-associatedness and orders the fringe of the tree formed by the binary operator left-right according to the `tm_lt` relation. As the name implies, the *bubble-sort* algorithm is used. I chose bubblesort because it is the simplest algorithm to implement by rewriting. Rewriting strategies for others (such as merge-sort) could have been coded too with a bit more work, but bubblesort proved to be good enough for the cases where I needed it.

### 6.12.2 Lemma Specialization

Frequently, a Lemma is proven about general instances of some class, and one then desires versions of the lemma that are specialized to either a specific instance of the class or to general instances of a more specific class. For example, consider the lemma `abmonoid_ac_1`:

⊢ ∀g:IAbMonoid. ∀a,b,c:|g|. a g.op (b g.op c) = b g.op (a g.op c)

One instance of the `IAbMonoid` class is the additive monoid over the rationals. The above lemma, instantiated with this instance is:

⊢ ∀a,b,c:$\mathbb{Q}$. a $+_q$ (b $+_q$ c) = b $+_q$ (a $+_q$ c)

Examples of the `abmonoid_ac_1` lemma about general instances over a more specific classes are

⊢ ∀r:IRng. ∀a,b,c:|r|. a +r (b +r c) = b +r (a +r c)

⊢ ∀r:ICRng. ∀a,b,c:|r|. a *r (b *r c) = b *r (a *r c)

Ideally, such instantiations would be computed on-the-fly by a fancy matching function. There is the issue here of how the matcher would guess appropriate instantiations, and in practice, there probably would be some way for the user to provide suitable hints to the matcher. The developers of the IMPS theorem prover have experimented with such ideas [FGT92b] and it would be interesting to see whether these could be adapted to Nuprl.

For simplicity, I have not touched the matcher, but have written utility functions that automatically instantiate sets of similar lemmas. One use of these functions is in creating sets of lemmas as needed by the simplification conversions described in the previous section.

# Chapter 7

# Permutations

## 7.1 Introduction

Section 7.2 presents a development of the basic theory of permutation functions, and Section 7.3 presents two approaches that were explored to defining a permutation relation on lists. One of these approaches involved defining a permutation relation in terms of permutation functions. This work is presented in Section 7.3.1. The other approach involved defining the relation recursively, and is presented in Section 7.3.2.

I developed first the definition in terms of permutation functions, because I was curious to explore the difficulties in pushing through an abstract development, close in style to that which might be found in an algebra textbook. Section 7.3.1 together with Section 7.2 might serve as the beginning of an exposition on the theory of (constructive) permutations.

This definition was adequate for use in the factorization work described in Chapter 8. Later however, in the development of the theory of finite multisets and sets (see Chapter 9), I needed a computable definition, and resorted to recursive definition of the permutation relation on lists.

Showing the two definitions equivalent was awkward, because I chose to characterize the auxiliary functions involved in the recursive definition in terms of the functional definition. This entailed proving many theorems about the properties of the list element select function on these auxiliary functions.

In retrospect, it probably would have been more elegant to have shown the equivalence of the two definitions by going through a third characterization of two lists being a permutation of one-another when the counts of each possible element in each list are the same. This third characterization was very useful in my development of the theory of finite sets and multisets (see Chapter 9).

## 7.2 Permutation Functions

Classically, a permutation on a set $S$ is a bijection of type $S \to S$. Implicit in the definition of bijection $f$ of type $A \to B$ is the existence of an inverse function $g$ of type $B \to A$. There is no way in general of computing $g$ from $f$, even though $g$ is a useful function, so constructively a bijection is commonly defined as a pair of functions $\langle f, g \rangle$ that are mutual inverses.

The definition for the type of permutations `Perm(T)` over type `T` was:

```
Perm(T) == {p:PermSig(T)| InvFuns(T,T,p.f,p.b)}
```

where the `PermSig` definition

```
PermSig(T) == (T → T) × (T → T)
```

was introduced by the class declaration shown in Figure 7.1, and the definition of

```
Class Declaration for: PermSig(T)

  Long Name: perm_sig
  Short Name: perm

  Parameters:
     T : 𝕌

  Fields:
     f : T → T
     b : T → T

  Universe: 𝕌
```

Figure 7.1: Signature Class for Permutation Functions

`InvFuns` was

```
InvFuns(A;B;f;g) == g o f = Id ∈ A → A ∧ f o g = Id ∈ B → B
```

Definitions for the components of the group of permutation functions on a type were:

```
Id == λx.x
```

```
f o g == λx.f (g x)
```

```
mk_perm(f;b) == <f,b>

id_perm() == mk_perm(Id;Id)

inv_perm(p) == mk_perm(p.b;p.f)

p O q == mk_perm(p.f o q.f;q.b o p.b)

perm_igrp(T) == mk_igrp(Perm(T);λp,q.p O q;id_perm();λp.inv_perm(p))
```

and the theorem that `perm_igrp(T)` was indeed a group was:

```
⊢∀T:𝕌. perm_igrp(T) ∈ IGroup
```

Note that `perm_igrp(T)` is in general an indiscrete group; there is no way of deciding the equality of functions when no deciding function is provided for their range or when the functions' domain is not finite and enumerable [1].

I then concentrated my efforts on proving properties about `Sym(n)`, the symmetry group on `n` elements:

```
ℕ<j:nat>== {0..<j>⁻}

Sym(n) == Perm(ℕn)
```

In particular, a result I wanted was showing that every permutation in `Sym(n)` is a composition of transpositions. I started with a definition of a swap function

```
  swap(i;j)
   == λn.if (n =z  i) then j
         if (n =z  j) then i
         else n
         fi
```

which had typing lemma

```
  ∀n:ℕ. ∀i,j:ℕn.  swap(i;j) ∈ ℕn → ℕn
```

I proved such theorems as:

```
swap_order_2:
  ∀n:ℕ. ∀i,j:ℕn.  swap(i;j) o swap(i;j) = Id ∈ ℕn → ℕn

swap_sym:
  ∀n:ℕ. ∀i,j:ℕn.  swap(i;j) = swap(j;i) ∈ ℕn → ℕn
```

[1]Not all constructive notions of finiteness imply enumerability, so this is not a redundant qualifier

```
triple_swap:
  ∀n:ℕ. ∀i,j,k:ℕn.
    ¬(i = j)
    ⇒ ¬(j = k)
    ⇒ swap(i;j)
       = swap(i;k) o (swap(j;k) o swap(i;k))
       ∈ ℕn → ℕn
```

The proof of the last theorem was made manageable by writing a tactic to automate case-splits (over 50 case splits were involved). The `swap` definition was used to define a transposition permutation

```
txpose_perm(i;j) == mk_perm(swap(i;j);swap(i;j))
```

and corresponding theorems were proven. For example:

```
  ∀n:ℕ. ∀i,j:ℕn.
    txpose_perm(i;j) O txpose_perm(i;j) = id_perm() ∈ Sym(n)
```

This tedious doubling of definitions and theorems, first over ℕ→ℕ, and then over `Sym(n)`, occurred throughout the development of permutations. The doubling was needed so that a computable function for taking inverses of permutations could be defined; in keeping with the constructive approach of Nuprl's type theory, no methods are provided for constructing non-computable functions. Note that Howe showed that it is consistent to add such functions [How91a] to the type theory, and also, one can always prove theorems with an explicit hypothesis about the existence say of an $\epsilon$ Hilbert choice function [HB70]. I did not explore either of these approaches here.

The theorem I proved about every element of `Sym(n)` being a composition of swaps was:

```
sym_grp_is_swaps:
⊢∀n:ℕ
  ∀p:Sym(n)
   ∃abs:(ℕn × ℕn) list
    (p
     = Π{(perm_igrp(ℕn)} map(λab.let a,b = ab in txpose_perm(a;b);abs)
     ∈ Sym(n))
```

where the generalized-product function `Π{·}` is introduced in Section 6.10 and `map(f;as)` was the usual polymorphic function for mapping a function `f` over a list `as`.

I proved this theorem by induction on `n`, using a number of auxiliary theorems about extending and restricting permutations such as

```
extend_perm_over_id:
  ∀n:ℕ. ↑{n}(id_perm()) = id_perm() ∈ Sym(n + 1)
```

```
extend_perm_over_comp:
  ∀n:ℕ. ∀p,q:Sym(n).
    ↑{n}(p O q) = ↑{n}(p) O ↑{n}(q) ∈ Sym(n + 1)

extend_perm_over_itcomp:
  ∀n:ℕ. ∀ps:Sym(n) List.
    ↑{n}(Π perm_igrp(ℕn) ps)
    = Π perm_igrp(ℕ(n + 1)) map(λp.↑{n}(p);ps)
    ∈ Sym(n + 1)

extend_perm_over_txpose:
  ∀n:ℕ. ∀i,j:ℕn.
    ↑{n}(txpose_perm(i;j)) = txpose_perm(i;j) ∈ Sym(n + 1)

extend_restrict_perm_cancel:
  ∀n:{1...}. ∀p:Sym(n).
    p.f (n - 1) = n - 1 ∈ ℕn
    ⇒ ↑{n - 1}(restrict_perm(p;n - 1)) = p ∈ Sym(n)

restrict_perm_using_txpose:
  ∀n:{1...}. ∀p:Sym(n).
    ∃q:Sym(n - 1)
     ∃i,j:ℕn. p = txpose_perm(i;j) O ↑{n - 1}(q) ∈ Sym(n)
```

where

```
extend_permf:
  extend_permf(pf;n) == λi.if (i =z n) then i else pf i fi

extend_perm:
  ↑{n}(p) == mk_perm(extend_permf(p.f;n);extend_permf(p.b;n))

restrict_perm:
  restrict_perm(p;n) == p
```

The `extend_perm` and `restrict_perm` definitions have the typing lemmas

```
extend_perm_wf:
  ∀n:ℕ. ∀p:Sym(n).  ↑{n}(p) ∈ Sym(n + 1)

restrict_perm_wf:
  ∀n:ℕ. ∀p:Sym(n + 1).
    p.f n = n ∈ ℕ(n + 1) ⇒ restrict_perm(p;n) ∈ Sym(n)
```

From the `sym_grp_is_swaps` theorem, I proved a couple of useful induction lemmas for permutations. One of them was:

```
perm_induction_a:
⊢ ∀n:ℕ
   ∀Q:Sym(n) → ℙ
    Q[id_perm()]
    ⇒ (∀p:Sym(n). Q[p] ⇒ (∀i:{1..n⁻}. Q[txpose_perm(i;0) O p]))
    ⇒ {∀p:Sym(n). Q[p]}
```

I used this induction lemma to prove the invariance of the value of sums of elements of abelian monoids under permutation of the order of the elements:

```
mon_itop_perm_invar:
  ∀g:IAbMonoid. ∀n:ℕ. ∀E:ℕn → |g|. ∀p:Sym(n).
    Πg 0 ≤ j < n. E[p.f j] = Πg 0 ≤ j < n. E[j] ∈ |g|
```

## 7.3 Permutation Relations

### 7.3.1 Defined using Permutation Functions

The definition of the permutation relation `permr` on lists says that two lists `as` and `bs` are a permutation of each other if they are the same length and the forward permutation function permutes the `bs` into the `as`. The definition was:

```
permr:
  as ≡(T) bs
   == (||as|| = ||bs||) c∧ (∃p:Sym(||as||)
                              ∀i:ℕ||as||. as[(p.f i)] = bs[i])
```

where `||·||` is the length function for lists, and `as[i]` is the `select` function for selecting the `i`th element from list `as` (counting the head of `as` as the 0th element). The definitions of these functions were

```
length:
  ||as||==r case as of [] => 0 | a::as' => ||as'|| + 1 esac

nth_tl:
  nth_tl(n;as)
  ==r if n ≤z 0 then as else nth_tl(n - 1;tl(as)) fi

select:
  l[i] == hd(nth_tl(i;l))
```

The notation `c∧` is for a *conditional and* constructor. Its definition is the same as the usual ∧ of Nuprl's type theory. However it is type checked slightly differently. In particular, in order to prove `P c∧ Q` well-formed, one has to prove `P` well-formed, and then only has to prove `Q` well-formed when `P` is assumed true. With `P c∧ Q`, one has to prove `Q` well-formed irrespective of th truth of `P`. The 'conditional and' is needed to guarantee that the index argument to the `select` functions are always in range.

The theorems stating that `permr` is an equivalence relation followed immediately from the group properties of permutations.

```
permr_weakening:
  ∀T:𝕌. ∀as,bs:T List.  as = bs ⇒ (as ≡(T) bs)

permr_inversion:
  ∀T:𝕌. ∀as,bs:T List.  (bs ≡(T) as) ⇒ (as ≡(T) bs)

permr_transitivity:
  ∀T:𝕌. ∀as,bs,cs:T List.
    (as ≡(T) bs) ⇒ (bs ≡(T) cs) ⇒ (as ≡(T) cs)
```

Given the `permr` relation, I immediately was able to restate the permutation invariance theorem given at the end of the last section in terms of lists and the generalized list product function $\boldsymbol{\Pi}$.

```
mon_reduce_functionality_wrt_permr:
  ∀g:IAbMonoid. ∀xs,ys:|g| List.
    (xs ≡(|g|) ys) ⇒ Π xs = Π ys
```

A theorem involving `permr` that I needed for the factorization work was:

```
select_reject_permr:
  ∀T:𝕌. ∀as:T List. ∀i:ℕ||as||.  ((as[i]::as\[i]) ≡(T) as)
```

where `as\[i]` is the `reject` function for removing the `i`th element from the list `as` and `::` is the list cons constructor. I could have proved this by figuring out the permutation function, but instead I chose to prove it using induction over the list `as` and the lemmas:

```
cons_functionality_wrt_permr:
  ∀T:𝕌. ∀a,b:T. ∀as,bs:T List.
    a = b ⇒ (as ≡(T) bs) ⇒ ((a::as) ≡(T) (b::bs))

hd_two_swap_permr:
  ∀T:𝕌. ∀as:T List. ∀a,a':T.  ((a::a'::as) ≡(T) (a'::a::as))
```

The first of these theorems was a little more work than I expected because the permutation extension function I defined before worked on the 'hi end' of permutations rather than the 'lo end': let `p` be the permutation that permutes a list `bs` into the list `as`. Then, the permutation I used for permuting the list `a::bs` into the list `a::as` was

```
conj{⇌p{|as| + 1}}(↑{|as|}(conj{⇌p{|as|}}(p)))
```

where the conjugate permutation operator conj and the reverse permutation ⇌ are defined as:

```
rev_permf(n) == λi.(n - 1 - i)
⇌p{n} == mk_perm(rev_permf(n);rev_permf(n))

conj{p}(q) == p O q O inv_perm(p)
```

With display forms that suppress parameters obvious from type-checking, this permutation looks like:

```
conj{⇌p}(↑(conj{⇌p}(p)))
```

A generalization of permr that was useful in the work on factorization in Chapter 8 was permr_upto:

```
as ≡ bs upto x,y.R[x; y]
 == (|as| = |bs|) c∧ (∃p:Sym(|as|). ∀i:ℕ|as|. R[as[(p.f i)]; bs[i]])
```

where R[x; y] could be instantiated with some arbitrary equivalence relation. In the factorization work, I instantiated R with the 'associate' relation in order to define the relation 'list *as* is a permutation of list *bs* up to associates'.

### 7.3.2 Defined Recursively

The recursive definition I used for the list permutation relation was:

```
as ≡_b bs
 ==r case as of
       [] => null(bs)
       a::as' => a ∈_b bs ∧_b as' ≡_b bs \ a
     esac
```

with typing lemma:

```
∀s:DSet. ∀as,bs:|s| List.  (as ≡_b bs) ∈ 𝔹
```

In this definition, $\in_b$ was the list member function, and bs \ a was a function for removing one occurrence of a from the list bs, simply returning bs if a was not a member of bs. Their definitions were:

```
mem:
  a ∈_b s as == ∃_b x(:|s|) ∈ as. x =_b s a

remove1:
  as \s a
   ==r case as of
         [] => []
         a'::as' => if a' =_b s a
                    then as'
                    else a'::(as' \s a)
                    fi
       esac
```

Note that the definitions of $\equiv_b$ , \, and $\in_b$ all take as an argument an element of the discrete set class `DSet`, but that this argument is normally hidden by the display forms I chose for these definitions. This argument supplies the computable equality function that these definitions need.

Standard properties of the $\equiv_b$ relation were proven by induction and the two definitions were shown equivalent.

## 7.4 Technical Details

The definitions described in Section 7.2 exposed a couple of weaknesses of the current Nuprl proof checking setup.

1. The match completion algorithm relies on being able to complete matches by inferring types of already found bindings. Occasionally matches failed because the algorithm couldn't infer `T` in the type `Perm(T)` of expressions such as:

   ```
   inv_perm(id_perm) O id_perm
   ```

   though `T` was inferable from the wider context in which such expressions occurred. I completed such steps by explicitly supplying a binding for $T$ to the matcher, though I don't consider this a solution. One solution would be to add a type tag to `id_perm`, so that we would have the definition:

   ```
   id_perm(T) == mk_perm(Id;Id)
   ```

   and corresponding typing lemma:

   ```
   ∀T:𝕌. (id_perm(T) ∈ Perm(T))
   ```

   Subsequent to noting this example, I encountered several others of a similar nature. A better long term solution would be to improve type inference so that it can look at surrounding contexts of matches in order to find types or to explore doing some automatic type-inference at term input time, and having the system fill in type tag slots of term. The display of some of these slots could be suppressed, so things would look the same as they do now.

2. Terms sometimes have well-formedness conditions that rely on predicates being true that are not generally provable by the `Auto` tactic. For example, the well-formedness lemma for `mk_perm` is:

   ```
   ⊢ ∀T:𝕌. ∀f,b:T → T. InvFuns(T,T,f,b) ⇒ mk_perm(f;b) ∈ Perm(T)
   ```

The rewriter creates a functionality tactic for `mk_perm` from this lemma that insists that the `InvFuns` predicate be checked every time the `f` or `b` argument to `mk_perm` is rewritten. I had examples where at stages before and after a rewrite, it was trivial to check that `mk_perm` well-formed because it was buried inside another abstraction. However during rewrite, it was exposed so I got the well-formedness antecedent to prove. It would be much cleaner if issues of well-formedness were separated from rewriting when it's known that rewriting doesn't affect well-formedness. However, this isn't possible in Nuprl's present type theory.

# Chapter 8

# Divisibility Theory

Divisibility theory in algebra is commonly first presented abstractly in integral domains, though much of the theory is only concerned with the multiplicative monoid of non-zero elements. For example, results concerning the existence and uniqueness of atomic factorizations and the properties of GCD's can first be developed over this multiplicative monoid.

A cancellation monoid with the property that every non-unit can be factored essentially uniquely into atoms is called a *unique-factorization monoid* (UFM). *Essentially uniquely* means up to permutations and associates. The monoid of non-zero elements of a unique-factorization domain (UFD) is an example of a UFM.

I present in this chapter my development of a theorem that characterizes when a cancellation monoid is a UFM. I also show that the fundamental theorem of arithmetic is an instance of this theorem.

The development drew on several other theories, most notably the basic theory of permutations developed in Chapter 7.

## 8.1 Factorization in Monoids

### 8.1.1 Basic Definitions

The basic definitions for divisibility theory over an abelian monoid g were all very straightforward:

```
b | a in g  == ∃c:|g|. a = b *g c ∈ |g|

g-unit(u) == u | eg in g

a p| b in g  == a | b in g  ∧ ¬(b | a in g )

Symmetrize(x,y.R[x; y];a;b) == R[a; b] ∧ R[b; a]
```

```
a ~{g} b == Symmetrize(x,y.x | y in g ;a;b)
```

Here, | is the divides relation, ~ is the associate relation and p| is the 'properly divides' relation. The notation ~ is potentially ambiguous, since it is also used to denote the group inverse. However, factorization in groups is trivial (every element is a unit), so both uses of this symbol should never occur in one theorem. In particular, all occurrences of the ~ symbol in this chapter denote the associate relation defined above.

I showed that the divides relation is a preorder, and that the associate relation is an equivalence relation. I proved the latter fact as an instance of a theorem that stated that any symmetrized preorder is an equivalence relation.

All factorization theory over monoids is modulo associates; all the basic predicates and functions respect the associate relation ~ and predicates concerning equality lift to predicates concerning associates. For example, I showed that the monoid operation * respects the associate relation ~ and that cancellation with respect to equality implies cancellation with respect to ~:

```
grp_op_ap2_functionality_wrt_massoc:
  ∀g:IAbMonoid. ∀a,a',b,b':|g|.
    a ~ b ⇒ a' ~ b' ⇒ a * a' ~ b * b'

massoc_cancel:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*) ⇒ (∀a,b,c:|g|.  a * b ~ a * c ⇒ b ~ c)
```

Other definitions I needed were for reducibility, atomicity (irreducibility) and primeness. For convenience I defined both types and predicates for atomicity and primeness.

```
IsPrime{g}(a)
== ¬g-unit(a)
   ∧ (∀b,c:|g|.
        a | b *g c in g  ⇒ a | b in g  ∨ a | c in g )

Prime{g} == {x:|g|| IsPrime{g}(x)}

Reducible{g}(a)
== ∃b,c:|g|. ¬g-unit(b) ∧ ¬g-unit(c) ∧ a = b *g c ∈ |g|

Atomic{g}(a) == ¬g-unit(a) ∧ ¬Reducible{g}(a)

Atom{g} == {a:|g|| Atomic{g}(a)}
```

A choice in making these definitions concerned reducibility and atomicity. I first defined reducibility so that if an element of |g| were composite, then two proper factors would be witnessed. Then I defined atomicity in terms of compositeness.

Note that whereas `Atomic` has no computational content and so can always be unhidden when it occurs in the `Atom` type, the `Prime` predicate does have significant computational content.

From these definitions I proved various facts such as that every prime is an atom in a cancellation monoid:

```
mprime_imp_matomic:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*g)
    ⇒ (∀a:|g|. Prime{g}(a) ⇒ Atomic{g}(a))
```

The notions of of primeness and atomicity are equivalent in UFM's (the non-zero integers under multiplication, for example), but are not equivalent in general. For example, consider the set of complex numbers of form $a + b\sqrt{-5}$ with $a$ and $b$ drawn from the integers and both not equal to zero. This set forms a cancellation monoid under normal multiplication. In this monoid, 9 has two factorizations: $3 \cdot 3$ and $(2 + \sqrt{-5})(2 - \sqrt{-5})$. Each of the factors is atomic, but none are prime ([Jac74], p136).

### 8.1.2 Existence Theorem

The main existence theorem I proved stated that in any cancellation monoid, if the 'properly divides' relation is well-founded and if reducibility is constructively decidable [1], then every non-unit factors into atomic elements. The statement of this theorem in Nuprl was

```
mfact_exists:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ WellFnd(|g|;x,y.x p| y )
    ⇒ (∀c:|g|. Dec(Reducible(c)))
    ⇒ (∀b:|g|. ¬g-unit(b) ⇒ (∃as:Atom{g} List. b = Π as))
```

where the `WellFnd` predicate was defined as:

```
  WellFnd(A;x,y.R[x; y])
  == ∀P:A → ℙ
       (∀j:A. (∀k:A. R[k; j] ⇒ P[k]) ⇒ P[j]) ⇒ {∀n:A. P[n]}
```

In classical mathematics, this predicate is equivalent to the statements that there are no infinite descending chains, and that every subset has a minimal element. Constructively, all three statements are inequivalent; the one above is the strongest. It is not implied by either of the other two. The advantage of the one above is

[1] I explain what I mean by *constructively* decidable in Section 5.2.5

that provides a means of doing a constructive well-founded induction in the proof of the theorem.

An abbreviated proof printout for the theorem is shown in Figure 8.1.

The abbreviated proof is presented in a style very similar to that in which Nuprl proofs are normally presented:

At `BY`, one or more inference steps are explained which refine the goal immediately above the `BY` into zero or more subgoals below the `BY`. For compactness, the proof only shows those parts of the sequent that have been changed by the refinement. The printout starts after a trivial initial step that uses the `RepD` tactic. I have described each refinement in English, so hopefully the proof can be read without further explanation.

The full proof when printed out is less than two pages long, but the formal tactic language makes it less accessible. It can be found in Section A.1 of Appendix A.

A trivial corollary did away with the restriction about non-units, providing that being a unit was decidable.

```
mfact_exists_a:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ WellFnd(|g|;x,y.x p| y)
    ⇒ (∀c:|g|. Dec(Reducible(c)))
    ⇒ (∀c:|g|. Dec(g-unit(c)))
    ⇒ (∀b:|g|. ∃as:Atom{g} List. b ∼ Π as)
```

The `mfact_exists` and `mfact_exists_a` theorems have the classic ∀∃ structure of theorems with interesting computational content; read constructively, the theorem claims that given a monoid that satisfies all the preconditions, and given an arbitrary element `b` of that monoid, a factorization into atomic elements can be *computed.*

### 8.1.3 Uniqueness Theorem

The main uniqueness theorem I proved stated that in any cancellation monoid, if the 'divides' relation is constructively decidable, then every element of the monoid factors into primes in an essentially unique way. The statement of this theorem in Nuprl was

```
unique_mfact:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ (∀a,b:|g|.  Dec(a | b))
    ⇒ (∀ps,qs:Prime(g) List.  Π ps ∼ Π qs ⇒ ps ≡ qs upto ∼)
```

```
1. g: IAbMonoid
2. Cancel(|g|;|g|;*)
3. WellFnd(|g|;x,y.x p| y)
4. ∀c:|g|. Dec(Reducible(c))
5. b: |g|
6. ¬(g-unit(b))
⊢ ∃as:Atom{g} List. b = Π as
│
BY  Induction on b  using hyp 3.
│
5. j: |g|
6. ∀k:|g|. k p| j ⇒ ¬(g-unit(k)) ⇒ (∃as:Atom{g} List. k = Π as)
7. ¬(g-unit(j))
⊢ ∃as:Atom{g} List. j = Π as
│
BY  Decide if  j is atomic or reducible.
│
├─8. Reducible(j)
│  │
│  BY  Hyp 8 implies that  j has two proper divisors:  b and  c.
│  │
│  8. b: |g|
│  9. c: |g|
│  10. ¬(g-unit(b))
│  11. ¬(g-unit(c))
│  12. j = b * c
│  13. b p| j
│  14. c p| j
│  │
│  BY  Apply hyp 6 to  b and  c
│  │
│  15. as1: Atom{g} List
│  16. b = Π as1
│  17. as2: Atom{g} List
│  18. c = Π as2
│  │
│  BY  Use  '(as1 @ as2)' for  as in concl.
│      Concl then follows by hyps 12, 16 and 18.
│
└─8. ¬Reducible(j)
   │
   BY  Use  'j::[]' for  as in concl and then concl is obvious.
```

Figure 8.1: Abbreviated Proof of Existence of Atomic Factorizations

Here, the notation '`ps ≡ qs upto ∼`' is for a 'permutation and associates' relation. It can be read as saying "`ps` is equal to `qs` up to permutations and associates." Its two-stage definition was:

```
permr_upto:
  as ≡ bs upto x,y.R[x; y]
  == (||as|| = ||bs||) c∧ (∃p:Sym(||as||)
                            ∀i:ℕ||as||. R[as[(p.f i)]; bs[i]])

permr_massoc:
  as ≡ bs upto ∼{g} == as ≡ bs upto x,y.x ∼{g} y
```

An abbreviated proof of the theorem is shown in Figure 8.2. The full proof printout is less than 3 pages long and can be found in full in Section A.2 of Appendix A.

One of the more interesting steps in the full proof is reproduced in Figure 8.3. This corresponds to the penultimate step of the abbreviated proof. The lemma referred to is

```
select_reject_permr:
  ∀T:𝕌. ∀as:T List. ∀i:ℕ||as||.  ((as[i]::as\[i]) ≡(T) as)
```

In order to rewrite the conclusion, the rewriter looked up a lemma, verifying that the `≡ upto` relation respected the permutation relation `≡(|g|)`. Also, in order to rewrite the hypothesis that moved to the end of the hypothesis list, the rewriter checked that the ***Π***`(g)` respected the permutation relation; specifically, it checked that `g` was abelian.

### 8.1.4 Unique Factorization Monoid Existence

I present here a summarizing theorem that I proved that gives conditions for when a cancellation monoid is a UFM. First, I give a few definitions. A *uniquely satisfies up to* predicate was defined as

```
uni_sat_upto:
  a r !x:T. Q[x]  == Q[a] ∧ (∀a':T. Q[a'] ⇒ a' [r] a)
```

Given a type `T`, an equivalence relation `r` on `T`, and an element `a` of type `T`, the expression '`a r !x:T. Q[x]`' can be read as "upto `r`, `a` is the unique `x` of type `T` such that `Q[x]` holds".

An *exists unique up to* predicate was defined in terms of the 'uniquely satisfies up to' predicate as:

```
exists_uni_upto:
  (r)∃!x:T. Q[x] == ∃a:T. a r !x:T. Q[x]
```

```
1. g: IAbMonoid
2. Cancel(|g|;|g|;*)
3. ∀a,b:|g|.  Dec(a | b)
4. ps: Prime(g) List
⊢ ∀qs:Prime(g) List. Π ps ~ Π qs ⇒ ps ≡ qs upto ~
|
BY  List induction on  ps.
|
├─5. qs: Prime(g) List
|  6. e ~ Π qs
|  ⊢ [] ≡ qs upto ~
|  |
|  BY  hyp 6 says that Π qs  is a unit. This can only happen if q = []
|
└─5. p: Prime(g)
   6. ps': Prime(g) List
   7. ∀qs:Prime(g) List. Π ps' ~ Π qs ⇒ ps' ≡ qs upto ~
   8. qs: Prime(g) List
   9. p * Π ps' ~ Π qs
   ⊢ p::ps' ≡ qs upto ~
   |
   BY  Hyp 9 implies that p  divides Π qs
   |   Since p  is prime, p  divides an element i  of qs.
   |
   10. i: ℕ||qs||
   11. p | qs[i]
   |
   BY  Since p  non-unit and qs[i]  atomic, hyp 11 can be strengthened.
   |
   11. p ~ qs[i]
   |
   BY  Move hyp 9 to end of hyps to put in scope of  i.
   |   Bring  qs[i] to front of  qs in moved hyp 9 and concl.
   |
   9. i: ℕ||qs||
   10. p ~ qs[i]
   11. p * Π ps' ~ qs[i] * Π qs\[i]
   ⊢ p::ps' ≡ qs[i]::qs\[i] upto ~
   |
   BY  Decompose ::  in concl and apply cancellation hyp 2 to hyp 11.
   |
   11. Π ps' ~ Π qs\[i]
   ⊢ ps' ≡ qs\[i] upto ~
   |
   BY  Concl follows from hyp 11 using induction hyp 7.
```

Figure 8.2: Abbreviated proof of Uniqueness of Prime Factorizations

```
1. g: IAbMonoid
2. Cancel(|g|;|g|;*)
3. ∀a,b:|g|.  Dec(a | b)
4. ps: Prime(g) List
5. p: Prime(g)
6. ps': Prime(g) List
7. ∀qs:Prime(g) List. Π(g) ps' ∼{g} Π(g) qs ⇒ ps' ≡ qs upto ∼{g}
8. qs: Prime(g) List
9. p *g Π(g) ps' ∼{g} Π(g) qs
10. i: ℕ||qs||
11. p ∼{g} qs[i]
⊢ p::ps' ≡ qs upto ∼{g}
|
BY MoveToEnd 9
|  THEN (OnMCls [0;-1] (RWH (IfIsC ⌈qs⌉
|         (RevLemmaWithC [‘i’,⌈i⌉] ‘select_reject_permr‘))) ...a)
|  THEN AbReduce (-1)
|
9. i: ℕ||qs||
10. p ∼{g} qs[i]
11. p *g Π(g) ps' ∼{g} qs[i] *g Π(g) qs\[i]
⊢ p::ps' ≡ qs[i]::qs\[i] upto ∼{g}
```

Figure 8.3: Step in Proof of Uniqueness of Prime Factorizations

Given a type `T` and an equivalence relation `r` on `T`, the expression `(r)∃!x:T. Q[x]` can be read as "upto `r`, there exists a unique `x` of type `T` such that `Q[x]` holds".

A UFM predicate was defined as

```
IsUFM(g)
== ∀b:|g|. ¬(g-unit(b)) ⇒ (≡∼)∃!as:Atom{g} List. (b = Π as)
```

Here, the ≡∼ notation denotes the 'permutation and associates' relation on `g`. So this definition can be read as "`g` is a unique factorization monoid, just when every non-unit can be factored uniquely (up to permutations and associates) into atomic elements".

The previous two theorems about the existence and uniqueness of factorizations were then combined into the single theorem:

```
ufm_char:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*g)
    ⇒ WellFnd(|g|;x,y.x p| y in g )
    ⇒ (∀a:Atom{g}. IsPrime{g}(a))
    ⇒ (∀a:|g|. Dec(Reducible{g}(a)))
    ⇒ (∀a,b:|g|.  Dec(a | b in g ))
    ⇒ IsUFM(g)
```

### 8.1.5 The Fundamental Theorem of Arithmetic

The fundamental theorem of arithmetic essentially says that the monoid of the positive integers under multiplication form a UFM. With the definition of the multiplicative monoid of positive integers:

```
<ℤ+,*>
== <ℕ+, λx,y.(x =z  y), λx,y.x ≤z y, λx,y.x * y, 1, λx.x>
```

and a set of lemmas, verifying that `<ℤ+,*>` satisfied all the preconditions of the `ufm_char` lemma in the previous section, I established the theorem:

```
IsUFM(<ℤ+,*>)
```

# Chapter 9

# Finite Sets and Finite Multisets

## 9.1 Introduction

The work described in the this chapter was done with several aims in mind:

1. To explore the view of algebraic class definitions as abstract-data-type (ADT) specifications, and of inhabitants of these classes as implementations of the specifications. In particular, I wanted to explore the properties of free algebraic class definitions.

   This work served as a foundation for the much larger case study on ADT specification and implementation described in Chapter 10.

2. To investigate the abstracting properties of Nuprl's equality-quotienting type.

3. To develop a library of definitions and theorems covering constructive finite sets and multisets. By constructive here, I mean that all sets and multisets have a concrete representation, and all the primary functions and predicates are computable.

   This library was put to use in characterizing the ADT implementations developed in Chapter 10. Note that this use was quite distinct from the use described above.

The particular algebraic class studied in this chapter as an ADT specification, is that of free abelian monoids over a set. All elements of this class are isomorphic, and so in mathematics one usually talks of *the* free abelian monoid on some set $s$. However, instances of this class can be constructed in quite different ways and have different computational characteristics. The implementation described here is based on using lists of elements of $s$ with the order of elements in the lists ignored. A second implementation was produced in the work on polynomials described in Chapter 10. This implementation was based on association lists (a-lists) and assumed that the set $s$ was linearly ordered. It allowed multiset operations such as union and intersection to be computed in $O(n)$ time as oppose to in $O(n^2)$ time for the list implementation here.

## 9.2 Definition of Free Abelian Monoid Class

The abstract characterization of being a *free abelian monoid* says that a monoid $\mathcal{M} = \langle M, *, e \rangle$ is a free abelian monoid over a set $S$ if there is an mapping $\iota$ of $S$ into $M$, such that for any abelian monoid $\mathcal{M}'$ and mapping $\phi$ of $S$ into $M'$, there is a unique abelian monoid homomorphism $\hat{\phi}$ from $M$ to $M'$ which satisfies the equation $\phi = \hat{\phi} \circ \iota$. This equation can be stated pictorially by saying for each $\mathcal{M}'$ and $\phi$ there is a unique $\hat{\phi}$ such that the following diagram commutes:

$$\begin{tikzcd} S \arrow[d, "\iota"] \arrow[dr, "\phi"] & \\ M \arrow[r, "\hat{\phi}"] & M' \end{tikzcd} .$$

This property of a free abelian monoid is a paradigmatic example of a universal property. Such properties are extensively studied in algebra, universal algebra and category theory.

The class definition that I used for free abelian monoids is shown in Figure 9.1. This definition captures the requirement that there is an appropriate mapping into any abelian monoid by insisting that a function be supplied that generates this mapping. Such a function is useful for building a variety of more specific functions.

Auxiliary definitions used in the definition of `FAbMon` are:

```
MonHom(M1,M2) == {f:(|M1| → |M2|)| monoid_hom_p(M1;M2;f)}

monoid_hom_p(M1;M2;f) == fun_thru_2op(|M1|;|M2|;M1.op;M2.op;f)
                          ∧ ((f M1.id) = M2.id ∈ |M2|)

{!x:T | P[x]} == {x:T| (P[x] ∧ (∀y:T. P[y] ⇒ (y = x ∈ T)))}
```

`FAbMon` is an example of a class definition that is parameterized by an element of another class (`DSet` in this case).

To exercise this definition, I proved the uniqueness of instances of this class up to isomorphism:

```
⊢ ∀S:DSet
   ∀M,N:FAbMon(S)
    ∃f:MonHom(M.mon,N.mon).
     ∃g:MonHom(N.mon,M.mon). InvFuns(|M.mon|;|N.mon|;f;g)
```

```
Class Declaration for: FAbMon(S)

  Long Name: free_abmonoid
  Short Name: free_abmon

  Parameters:
     S : DSet

  Fields:
     mon : AbMon
     inj : |S| → |mon|
     umap : mon':AbMon
            → f':(|S| → |mon'|)
            → {!g:MonHom(mon,mon') | g o inj
                                     = f'
                                     ∈ |S| → |mon'|}

  Universe: 𝕌'
```

Figure 9.1: Class of Free Abelian Monoids

Note the constructive content of this theorem; the content is a computable function that given any free abelian monoids M and N, can come up with the monoid homomorphisms that shows that the monoids are isomorphic. This theorem is only provable constructively because the class definition for free abelian monoids requires this universal mapping function to be supplied.

This theorem indicates the genericity of any implementation of the class; if a function is definable on one implementation of the class, then the mapping functions can be used to construct an equivalent function on any other implementation. In theory, this might be a very handy feature; some functions are much more easily defined on one implementation than another. In practice, the translation back-and-forth might not be too efficient.

## 9.3 Implementation of Free Abelian Monoid

Here I show an implementation of the free abelian monoid class over a set $S$ that uses lists of elements of $S$ as the monoid carrier. One of the main purposes of this example is to illustrate the usefulness of Nuprl's quotient type.

As explained earlier, my approach to building classes in Nuprl hinges on being able to use the quotient type to build carrier types with appropriate associated equality relations. I also discuss limitations of the current quotient type and directions in which it needs to be improved.

Incidentally, this example also proves that free abelian monoids exist.

### 9.3.1 Sketch of Implementation

Ignoring typing and verification issues, the construction of an implementation based on lists was rather straightforward. In order to have a computable equality relation on the carrier of the free abelian monoid be computable, I needed the generating set to be some element `s` of the `DSet` class rather than just a type in universe $\mathbb{U}$. The monoid was formed using lists over `|s|` for the monoid carrier, list append `@` for the binary monoid operation, and the empty list `[]` for the monoid unit. For the boolean-valued function computing whether two lists were equal when considered as multisets, I used the `bpermr` function from the `list_2` theory. `bpermr{s}(as,bs)` returns `tt` just when the list `as` is a permutation of the list `bs`.

The injection of an element `a` of `|s|` into the monoid was the singleton list `a::[]`. Using the `mon_for` summation function, the universal mapping function was defined as:

```
λm,f,y.For{|s|,m} z ∈ y. f z
```

### 9.3.2 Definitions

Here are the actual definitions, their typing lemmas, and comments on how relevant properties about them were proved.

The definition and well-formedness lemma for the multiset type were:

```
*A mset              MSet{s} == as,bs:(|s| List)//(as ≡(|s|) bs)

*T mset_wf           ∀s:DSet. MSet{s} ∈ 𝕌
```

The `≡(|s|)` relation is the `permr` 'are related by permutation' relation from the `perms_2` theory and the `//` operation is Nuprl's type-quotienting operation.

The very first issue I came across in using this type was how to deal with its inclusion properties. I proved the lemma:

```
*T mset_qinc         ∀s:DSet. (|s| List) ⊆ MSet{s}
```

Here the differences between type theory and set theory really became apparent. In Nuprl's type theory, to prove `A ⊆ B` (defined as `∀x:A. x ∈ B`) one must not only show that every element of `A` is also an element of `B`, but also that if two elements are equal elements of `A` they are also equal elements of `B`. The lemma `mset_qinc` is provable partly because two lists that are considered equal as lists, are also related by a permutation (the identity permutation).

If and when proper subtyping predicates are added to Nuprl's type theory, it probably make sense to introduce two: one for when the equalities are the same but the elements might differ and the other for when the elements are the same but the equalities differ. One interesting question is whether suitable rules in Nuprl's type theory could be devised that cleanly make this distinction apparent. Having equality reasoning 'built-in' to the type theory might make things rather awkward.

I set up the `Inclusion` tactic (and hence the `Auto` tactic) to automatically look for and apply quotient inclusion lemmas such as `mset_qinc` above.

Since the computation language of Nuprl is intrinsically untyped, I found myself often unsure in the middle of a proof whether I was thinking of some term as a list or a multiset. Conceptually, I found it helpful to think of the `MSet{s}` type as more abstract than the `|s| List` type and where possible I introduced extra definitions that made this abstraction more explicit. As I explain in Section 9.3.5, I also introduced a set of rewrite lemmas for shifting around the position in a term where this abstraction is considered to be made.

The first definition I introduced was a function injecting lists into multisets:

```
*A mk_mset          mk_mset(as) == as

*T mk_mset_wf       ∀s:DSet. ∀as:|s| List. mk_mset(as) ∈ MSet{s}
```

I considered this function an 'encapsulation function'; it took lists, ordered sequences of elements over some type as input, and returned as output multisets of these elements, collections in which the order of elements was effectively hidden. Note that in Nuprl's type theory, it is not always possible to type a function that opens up multisets represented as lists, and returns the underlying lists. Nuprl requires all functions to respect the equalities of their domain and range types. An 'opening up' function would have to map lists thought of as equal multisets, to equal lists. Such a function could only be constructively defined if extra structure were assumed of the type of elements of the lists. For example, if the type were linearly ordered, then a sorting function would make a suitable 'opening up' function.

I used the `mk_mset` definition in defining the injection function from `|s|` into `DSet`:

```
*A mset_inj               mset_inj{s}(x) == mk_mset(x::[])

*T mset_inj_wf            ∀s:DSet. ∀x:|s|. mset_inj{s}(x) ∈ MSet{s}
```

I gave the `mset_inj` abstraction an extra parameter `s` to indicate the type of the multiset being injected into. This made sure that the type of `mset_inj` was inferable when needed. Perhaps this wasn't strictly necessary since the type could have been inferred from the type of `x`. However, there had been similar times in previous proofs when type inference had failed because the inferred type of `x` was say $\mathbb{Z}$ rather than `|<ℤ,λx,y.x =z y>|`. The type inference function didn't know enough to convert a raw type into an appropriate element of the `DSet` class. To be consistent, I probably should have also given `mk_mset` an extra similar parameter.

The null multiset, multiset sum and multiset equality were defined as:

```
*A null_mset                 0{s} == []

*T null_mset_wf              ∀s:DSet. 0{s} ∈ MSet{s}

*A mset_sum                  a + b == a @ b

*T mset_sum_wf               ∀s:DSet. ∀a,b:MSet{s}.  a + b ∈ MSet{s}
```

In Figure 9.2, I show the proof of `mset_sum_wf` because it illustrates the form of some the basic Nuprl rules for dealing with quotient types. The `D 3` and `D 2` tactics invoke the decomposition rule for quotient types in the hypothesis list. This rule is rather specialized in comparison with other hypothesis decomposition rules in that it requires the conclusion to be of a certain form, namely that it is an equality term. The rule turned out to be adequate for this multiset theory. The `EqTypeCD` tactic invokes the decomposition rule for the quotient type being the type of an equality in the conclusion. `RelArgCD` is a generalization of the `EqCD` tactic; it refines a goal involving an equivalence relation in the conclusion over two terms with same outermost constructor, to relations between the immediate subterms of the equivalence relation. In this case, the relations between the subterms follow immediately from the hyps `4` and `7`.

I introduced a new variant on `For` for summing over multisets rather than lists:

```
*A mset_for
            msFor{s,m} x ∈ a. f[x] == For{|s|,m} x ∈ a. f[x]

*T mset_for_wf
  ∀s:DSet. ∀g:IAbMonoid. ∀f:|s| → |g|. ∀a:MSet{s}.
    msFor{s,g} x ∈ a. f[x] ∈ |g|
```

The proof of the `mset_for_wf` lemma was straightforward and is shown in Figure 9.3. Here as elsewhere, I had already done all the harder work of showing functionality of various operators with respect to `permr`, so things look perhaps deceptively simple.

An interesting feature of the `mset_for_wf` lemma is that it requires `g` to be an abelian monoid. This lemma is unprovable, if `g` is merely required to be an element of the `GrpSig` class that `IAbMonoid` is derived from. This fact can easily be seen by considering the last step of the proof shown in Figure 9.3; the summation value is only invariant under permutation of the list being summed over when `g` at least has the properties of an abelian monoid.

```
⊢ ∀s:DSet. ∀a,b:MSet{s}. a + b ∈ MSet{s}
|
BY Unfold `mset_sum` 0
|  THEN (UnivCD ...a)
|
1. s: DSet
2. a: MSet{s}
3. b: MSet{s}
⊢ a @ b ∈ MSet{s}
|
BY Unfold `member` 0
|
⊢ a @ b = a @ b ∈ MSet{s}
|
BY (New [`b1';`b2'] (D 3) ...a)
|
3. b1: |s| List
4. b2: |s| List
5. b1 ≡(|s|) b2
⊢ a @ b1 = a @ b2 ∈ MSet{s}
|
BY (New [`a1';`a2'] (D 2) ...a)
|
2. b1: |s| List
3. b2: |s| List
4. b1 ≡(|s|) b2
5. a1: |s| List
6. a2: |s| List
7. a1 ≡(|s|) a2
⊢ a1 @ b1 = a2 @ b2 ∈ MSet{s}
|
BY (EqTypeCD ...a)
|
⊢ (a1 @ b1) ≡(|s|) (a2 @ b2)
|
BY (RelArgCD ...)
```

Figure 9.2: Proof of `mset_sum_wf` Lemma

```
⊢ ∀s:DSet. ∀g:IAbMonoid. ∀f:|s| → |g|. ∀a:MSet{s}.
|    msFor{s,g} x ∈ a. f[x] ∈ |g|
↓
BY (UnivCD ...a)
↓
1. s: DSet
2. g: IAbMonoid
3. f: |s| → |g|
4. a: MSet{s}
⊢ msFor{s,g} x ∈ a. f[x] ∈ |g|
↓
BY Unfolds ``member mset_for`` 0
|  THEN (D 4 ...a)
↓
4. a1: |s| List
5. a2: |s| List
6. a1 ≡(|s|) a2
⊢ (For{|s|,g} x ∈ a1. f[x]) = (For{|s|,g} x ∈ a2. f[x]) ∈ |g|
↓
BY (RWH (HypC 6) 0 ...)
```

Figure 9.3: Proof of `mset_for_wf` Lemma

### 9.3.3 Constructing a monoid of multisets

I generated lemmas showing some of the expected properties of the multiset operators:

```
*T mset_sum_comm               ∀s:DSet. Comm(MSet{s};λa,b.a + b)

*T mset_sum_assoc              ∀s:DSet. Assoc(MSet{s};λa,b.a + b)

*T assert_of_eq_mset
  ∀s:DSet. ∀a,b:MSet{s}.  ↑eq_mset{s}(a,b) ⟺ a = b ∈ MSet{s}
```

and then gave the construction of the multiset monoid:

```
*A mset_mon
  mset_mon{s}
  == <MSet{s}
     , λx,y.eq_mset{s}(x,y)
     , λx,y.tt
     , λx,y.x + y
     , 0{s}
     , λx.x>

*T mset_mon_wf                 ∀s:DSet. mset_mon{s} ∈ AbMonoid
```

The proof of mset_mon_wf was straightforward; I either referred back to one of the property lemmas as above or proved the property on the spot.

### 9.3.4 Evaluating the multiset summation function

Here I describe a rewrite conversion I put together for symbolically evaluating expressions involving msFor when it is the domain of some multiset expression: specifically when it is either a sum of two multisets, a singleton multiset or the null multiset. The lemmas involved were:

```
*T mset_for_mset_sum
  ∀s:DSet. ∀g:IAbMonoid. ∀f:|s| → |g|. ∀a,b:MSet{s}.
    msFor{s,g} x ∈ a + b. f[x]
    = (msFor{s,g} x ∈ a. f[x]) *g (msFor{s,g} x ∈ b. f[x])
    ∈ |g|

*T mset_for_mset_inj
  ∀s:DSet. ∀g:IAbMonoid. ∀f:|s| → |g|. ∀u:|s|.
    msFor{s,g} x ∈ mset_inj{s}(u). f[x] = f[u] ∈ |g|
```

Rewrite conversions for the null multiset and for normalizing expressions involving mset_for were defined by the ML object:

```
*M mset_for_eval
               let mset_for_null_msetC =
                 MacroC `mset_for_null_msetC`
                 (EvalC ``mset_for null_mset``)
                    'msFor{s,g} x ∈ 0{s}. f[x]'
                 IdC
                    'g.id'
               ;;

               let mset_for_normC =
                 TryC (SweepDnC
                         (mset_for_null_msetC
                         ORELSEC LemmaC `mset_for_mset_sum`
                         ORELSEC LemmaC `mset_for_mset_inj`))
               ;;
```

It might help here to use some general mathematical notation to describe the rules. Assuming $+$ is multiset sum, $\{u\}$ is a singleton multiset, $\emptyset$ is the empty multiset, and that msFor is summing over a multiplicative monoid, the mset_for_normC implements the rewrite rules:

$$\prod_{x \in \emptyset} f_x \quad \rightarrow \quad 1$$

$$\prod_{x \in \{u\}} f_x \quad \rightarrow \quad f_u$$
$$\prod_{x \in a+b} f_x \quad \rightarrow \quad \prod_{x \in a} f_x \times \prod_{x \in b} f_x$$

Examples of the use of `mset_for_normC` can be found in the proof of the `mset_fmon_wf` theorem given in Section 9.3.6.

### 9.3.5 Multiset elimination

Here I describe a rewrite conversion I constructed for converting an expression or proposition involving the `MSet` type and operators over multisets into one involving lists and the underlying list operations. The lemmas and `MacroC` atomic conversions involved are as follows:

```
*T all_mset_elim
  ∀s:DSet. ∀F:MSet{s} → ℙ.
    (∀a:MSet{s}. SqStable(F[a]))
    ⇒ ((∀a:MSet{s}. F[a]) ⟺ (∀a:|s| List. F[mk_mset(a)]))

*T equal_mset_elim
  ∀s:DSet. ∀as,bs:|s| List.
    mk_mset(as) = mk_mset(bs) ∈ MSet{s} ⟺ (as ≡(|s|) bs)

*M mset_elim_1
  let null_mset_elimC = SimpleMacroC ‘null_mset_elimC‘
    ⌈0{s}⌉ ⌈mk_mset([])⌉ ‘‘null_mset mk_mset‘‘;;

  let mset_inj_elimC =  SimpleMacroC ‘mset_inj_elimC‘
    ⌈mset_inj{s}(x)⌉ ⌈mk_mset(x::[])⌉ ‘‘mset_inj mk_mset‘‘;;

  let mset_sum_elimC = SimpleMacroC ‘mset_sum_elimC‘
  ⌈mk_mset(as) + mk_mset(bs)⌉ ⌈mk_mset(as @ bs)⌉
   ‘‘mset_sum mk_mset‘‘ ;;

*T mset_mon_for_elim
  ∀s:DSet. ∀T:𝕌. ∀f:T → |s| List. ∀as:T List.
    (For{T,mset_mon{s}} x ∈ as. mk_mset(f[x]))
    = mk_mset(For{T,lapp_mon(s)} x ∈ as. f[x])
    ∈ MSet{s}
```

```
*M mset_elim_2
  let mset_for_elimC = SimpleMacroC `mset_for_elimC`
    ⌈msFor{s,g} x ∈ mk_mset(as). F[x]⌉
  ⌈For{|s|,g} x ∈ as. F[x]⌉ ``mset_for mk_mset``  ;;

  let raise_mk_msetC =
    FirstC
      [null_mset_elimC;mset_inj_elimC;mset_sum_elimC;mset_for_
  elimC] ;;


  let mset_elimC =
    TryC (SweepDnC (LemmaC `all_mset_elim`))
    ANDTHENC TryC (SweepUpC raise_mk_msetC)
    ANDTHENC TryC (HigherC (LemmaC `mset_mon_for_elim`))
    ANDTHENC TryC (HigherC (LemmaC `equal_mset_elim`))
  ;;
```

The `lapp_mon(s)` monoid in `mset_mon_for_elim` is the monoid of lists over `|s|` with the empty list `[]` as the unit and `@` as the binary operator.

Note the following pattern of behavior in the various atomic rewrite rules:

1. the `LemmaC `all_mset_elim``, `null_mset_elimC` and `mset_inj_elimC` introduce `mk_mset` terms,
2. the `mset_sum_elimC` and `LemmaC mset_mon_for_elim` raise `mk_mset` terms upwards in term trees,
3. `LemmaC `equal_mset_elim`` and `mset_for_elimC` absorb `mk_mset` terms.

This pattern motivates the order in which the rewrite rules were assembled into one conversion.

The `mset_elimC` conversion was used in the proof a couple of theorems:

```
*T dist_hom_over_mset_for
  ∀s:DSet. ∀m,n:IAbMonoid. ∀f:MonHom(m,n). ∀a:MSet{s}.
  ∀g:|s| → |m|.
    f (msFor{s,m} x ∈ a. g[x])
    = msFor{s,n} x ∈ a. f g[x]
    ∈ |n|

*T mset_fact
  ∀s:DSet. ∀a:MSet{s}.
    a = msFor{s,mset_mon{s}} x ∈ a. mset_inj{s}(x) ∈ MSet{s}
```

```
⊢ ∀s:DSet. ∀m,n:IAbMonoid. ∀f:MonHom(m,n). ∀a:MSet{s}.
   ∀g:|s| → |m|.
     f (msFor{s,m} x ∈ a. g[x]) = msFor{s,n} x ∈ a. f g[x] ∈ |n|

BY (RepeatMFor 4 (D 0) ...a)

1. s: DSet
2. m: IAbMonoid
3. n: IAbMonoid
4. f: MonHom(m,n)
⊢ ∀a:MSet{s}. ∀g:|s| → |m|.
     f (msFor{s,m} x ∈ a. g[x]) = msFor{s,n} x ∈ a. f g[x] ∈ |n|

BY (RW mset_elimC 0 ...a)

⊢ ∀a:|s| List. ∀g:|s| → |m|.
     f (For{|s|,m} x ∈ a. g[x]) = (For{|s|,n} x ∈ a. f g[x]) ∈ |n|

BY (Backchain ``dist_hom_over_mon_for`` ...)
```

Figure 9.4: Proof of `dist_hom_over_mset_for` Lemma

```
⊢ ∀s:DSet. ∀a:MSet{s}.
     a = msFor{s,mset_mon{s}} x ∈ a. mset_inj{s}(x) ∈ MSet{s}

BY (D 0 ...a)

1. s: DSet
⊢ ∀a:MSet{s}
     a = msFor{s,mset_mon{s}} x ∈ a. mset_inj{s}(x) ∈ MSet{s}

BY (RW mset_elimC 0 ...a)

⊢ ∀a:|s| List. (a ≡(|s|) (For{|s|,lapp_mon(s)} x ∈ a. (x::[])))

BY (D 0 ...a)
   THEN (StrengthenRel THENM BLemma `lapp_fact` ...)
```

Figure 9.5: Proof of `mset_fact` Lemma

The proofs of these theorems are given in Figure 9.4 and Figure 9.5. These proofs used the following lemmas from the `list_2` theory:

```
*T lapp_fact
  ∀s:DSet. ∀as:|s| List.
    as = (For{|s|,lapp_mon(s)} x ∈ as. (x::[])) ∈ |s| List

*T dist_hom_over_mon_for
  ∀T:𝕌. ∀m,n:IMonoid. ∀f:MonHom(m,n). ∀a:T List. ∀g:T → |m|.
    f (For{T,m} x ∈ a. g[x]) = (For{T,n} x ∈ a. f g[x]) ∈ |n|
```

### 9.3.6 The main theorem

The definition of the free abelian monoid tuple of multiset functions, and the theorem that states that this tuple really is a free abelian monoid were

```
*A mset_fmon
  mset_fmon(s)
  == <mset_mon{s}
     , λx.mset_inj{s}(x)
     , λm,f,y.msFor{s,m} z ∈ y. f z>

*T mset_fmon_wf                ∀s:DSet. mset_fmon(s) ∈ FAbMon(s)
```

The proof of the `mset_mon_wf` theorem is shown in Figure 9.6 to Figure 9.9. I have included comments at most of the steps to explain what's going on.

## 9.4 Finite Multisets

As in the previous section, most of the functions here were first defined on lists, and then characterized as having a multiset type.

The relevant list functions had the typed definitions:

```
diff:
  ∀s:DSet. ∀as,bs:|s| List.
    (as -s bs)
    = case bs of [] => as | b::bs' => (as \s b) -s bs' esac
    ∈ |s| List

lmax:
  ∀s:DSet. ∀as,bs:|s| List.
    lmax(s;as;bs) = (as -s bs) @ bs ∈ |s| List

lmin:
  ∀s:DSet. ∀as,bs:|s| List.
    lmin(s;as;bs) = (as -s (as -s bs)) ∈ |s| List
```

```
⊢ ∀s:DSet. mset_fmon(s) ∈ FAbMon(s)
|
BY % Open definitions and let Auto rip %
|
|  (Unfolds ``mset_fmon free_abmonoid`` 0 ...)
|
|  % Only non-trivial goals to check involve the umap %
|
1. s: DSet
2. m: AbMonoid
3. f: |s| → |m|
⊢ (λy.msFor{s,m} z ∈ y. f z) ∈ {!g:MonHom(mset_mon{s},m) |
|                                   g o (λx.mset_inj{s}(x))
|                                   = f
|                                   ∈ |s| → |m|}
|
BY % Open up unique set concl type %
|
|  (MemTypeCD ...)
|
|→⊢ (λy.msFor{s,m} z ∈ y. f z) ∈ MonHom(mset_mon{s},m)
| |
| BY % Check umap is a homomorphism %
| |
| ...
|
|→⊢ (λy.msFor{s,m} z ∈ y. f z) o (λx.mset_inj{s}(x)) = f ∈ |s| → |m|
| |
| BY % Check commutativity of umap triangle %
| |
| ...
|
└→ 4. y: MonHom(mset_mon{s},m)
  5. y o (λx.mset_inj{s}(x)) = f ∈ |s| → |m|
  ⊢ y = (λy.msFor{s,m} z ∈ y. f z) ∈ MonHom(mset_mon{s},m)
  |
  BY % Check uniqueness of umap %
  |
  ...
```

Figure 9.6: Proof of mset_fmon_wf Theorem: Top Part

```
...
↓
⊢ (λy.msFor{s,m} z ∈ y. f z) ∈ MonHom(mset_mon{s},m)
↓
BY % Check umap is a homomorphism %
|
|  (MemTypeCD ...)
|  THEN AbEval ``monoid_hom_p fun_thru_2op`` 0
|  THEN (GenUnivCD ...a)
|
└→4. a1: MSet{s}
| 5. a2: MSet{s}
| ⊢ (msFor{s,m} z ∈ a1 + a2. f z)
| |   = (msFor{s,m} z ∈ a1. f z) m.op (msFor{s,m} z ∈ a2. f z)
| |   ∈ |m|
| ↓
| BY (RW mset_for_normC 0 ...)
|
└→⊢ (msFor{s,m} z ∈ 0{s}. f z) = m.id ∈ |m|
  ↓
  BY (RW mset_for_normC 0 ...)
```

Figure 9.7: Proof of `mset_fmon_wf` Theorem: *umap hom* Part

```
...
↓
⊢ (λy.msFor{s,m} z ∈ y. f z) o (λx.mset_inj{s}(x)) = f ∈ |s| → |m|
↓
BY % Check commutativity of umap triangle %
|
|  (Ext ...a) THEN AbReduce 0
↓
4. x: |s|
⊢ (msFor{s,m} z ∈ mset_inj{s}(x). f z) = f x ∈ |m|
↓
BY (RW mset_for_normC 0 ...)
```

Figure 9.8: Proof of `mset_fmon_wf` Theorem: *umap comm* Part

```
...
⊢ y = (λy.msFor{s,m} z ∈ y. f z) ∈ MonHom(mset_mon{s},m)

BY % Check uniqueness of umap %

  RenameVar ‘g’ 4   % fix name to avoid confusion %

4. g: MonHom(mset_mon{s},m)
5. g o (λx.mset_inj{s}(x)) = f ∈ |s| → |m|
⊢ g = (λy.msFor{s,m} z ∈ y. f z) ∈ MonHom(mset_mon{s},m)

BY % Open up MonHom type in concl %

  (EqTypeCD ...a)

 ⊢ g = (λy.msFor{s,m} z ∈ y. f z) ∈ |mset_mon{s}| → |m|

 BY % Show functions equal by using function extensionality rule %

   (New [‘w’] Ext ...a) THEN AbReduce 0

 6. w: |mset_mon{s}|
 ⊢ g w = (msFor{s,m} z ∈ w. f z) ∈ |m|

 BY % Eliminate f in concl using hyp 5 %

   (RWH (RevHypC 5) 0 ...a) THEN AbReduce 0

 ⊢ g w = (msFor{s,m} z ∈ w. g mset_inj{s}(z)) ∈ |m|

 BY % Pull g out past msFor and apply mset_fact lemma %

   (RWH (RevLemmaC ‘dist_hom_over_mset_for‘) 0
    THENM RWH (RevLemmaC ‘mset_fact‘) 0 ...)


 ⊢ monoid_hom_p(mset_mon{s};m;g)

 BY % concl follows trivially from hyp 4

   (AddProperties 4 ...)
```

Figure 9.9: Proof of `mset_fmon_wf` Theorem: *umap unique* Part

```
bsublist:
  ∀s:DSet. ∀as,bs:|s| List.
    bsublist(s;as;bs) = null(as -s bs)

count:
  ∀s:DSet. ∀a:|s|. ∀bs:|s| List.
    a #∈s bs = (For{|s|,<ℤ+>} x ∈ bs. b2i(x =b s a))
```

The count function provided the essential characterization of what the @, lmax, lmin and diff functions did when order was ignored. It provides the view of lists as 'functions of finite support' from the set s that the multisets are over into the naturals. Finite support means in particular here, that the functions map all but a finite number of elements in the domain to 0.

Theorems proved included:

```
count_append:
  ∀s:DSet. ∀as,bs:|s| List. ∀c:|s|.
    c #∈s (as @ bs) = (c #∈s as) + (c #∈s bs)

count_diff:
  ∀s:DSet. ∀as,bs:|s| List. ∀c:|s|.
    c #∈s (as -s bs) = (c #∈s as) -- (c #∈s bs)

count_lmax:
  ∀s:DSet. ∀as,bs:|s| List. ∀c:|s|.
    c #∈s lmax(s;as;bs) = imax(c #∈s as;c #∈s bs)

count_lmin:
  ∀s:DSet. ∀as,bs:|s| List. ∀c:|s|.
    c #∈s lmin(s;as;bs) = imin(c #∈s as;c #∈s bs)

count_bsublist_a:
  ∀s:DSet. ∀as,bs:|s| List.
    ↑bsublist(s;as;bs) ⟺ (∀c:|s|. c #∈s as ≤ c #∈s bs)
```

Once I proved a characterization of the permr list permutation relation (see Section 7.3) in terms of the count function

```
permr_iff_eq_counts:
  ∀s:DSet. ∀as,bs:|s| List.
    (as ≡(|s|) bs) ⟺ (∀x:|s|. x #∈s as = x #∈s bs)
```

it was straightforward to make new definitions for corresponding multiset operations

```
mset_union:
  ∀s:DSet. ∀a,b:MSet{s}.  a ∪s b = lmax(s;a;b) ∈ MSet{s}
```

```
mset_inter:
  ∀s:DSet. ∀a,b:MSet{s}.  a ∩s b = lmin(s;a;b) ∈ MSet{s}

mset_diff:
  ∀s:DSet. ∀a,b:MSet{s}.  a -s b = (a -s b) ∈ MSet{s}

bsubmset:
  ∀s:DSet. ∀a,b:MSet{s}.  a ⊆b s b = bsublist(s;a;b)
```

and to make corresponding `count` theorems:

```
mset_count_sum:
  ∀s:DSet. ∀as,bs:MSet{s}. ∀c:|s|.
    c #∈ (as + bs) = (c #∈ as) + (c #∈ bs)

mset_count_union:
  ∀s:DSet. ∀as,bs:MSet{s}. ∀c:|s|.
    c #∈ (as ∪ bs) = imax(c #∈ as;c #∈ bs)

mset_count_diff:
  ∀s:DSet. ∀as,bs:MSet{s}. ∀c:|s|.
    c #∈ (as - bs) = (c #∈ as) -- (c #∈ bs)

count_bsubmset:
  ∀s:DSet. ∀a,b:MSet{s}.
    ↑(a ⊆b  b) ⟺ (∀x:|s|. x #∈ a ≤ x #∈ b)
```

Additional definitions introduced for multisets included:

```
mset_map:
  ∀s,s':DSet. ∀f:|s| → |s'|. ∀a:MSet{s}.
    msmap{s,s'}(f;a)
    = msFor{mset_mon{s'}} x ∈ a. mset_inj{s'}(f x)
```

With the `count` characterization, it was trivial to verify algebraic properties of the `union` and `inter` operations:

```
mset_inter_assoc:
  ∀s:DSet. ∀a,b,c:MSet{s}.  a ∩s (b ∩s c) = (a ∩s b) ∩s c

mset_inter_comm:
  ∀s:DSet. ∀a,b:MSet{s}.  a ∩s b = b ∩s a

mset_union_ident_l:
  ∀s:DSet. ∀a:MSet{s}.  0{s} ∪ a = a
```

With such theorems, I showed that `union` operation formed an abelian monoid:

```
mset_union_mon_wf:
  ∀s:DSet. <MSet{s},∪,0> ∈ AbMon
```

## 9.5 Finite Sets

Finite sets on some discrete set s were simply defined as a subset of the multisets on set s.

```
fset:
  FSet{s} == {a:MSet{s}| ∀x:|s|. x #∈ a ≤ 1}
```

A function was defined to reduce any finite multiset to the corresponding finite set.

```
fset_of_mset:
  ∀s:DSet. ∀a:MSet{s}.
    fset_of_mset(s;a)
    = msFor{<MSet{s},∪,0>} x ∈ a. mset_inj{s}(x)
```

With this, a mapping function for finite sets was defined:

```
fset_map:
  ∀s,s':DSet. ∀f:|s| → |s'|. ∀a:MSet{s}.
    fs-map(f, a) = fset_of_mset(s';msmap{s,s'}(f;a))
```

I proved alternative well-formedness lemmas for the multiset operations:

```
null_mset_wf_f:
  ∀s:DSet. 0{s} ∈ FSet{s}

mset_union_wf_f:
  ∀s:DSet. ∀a,b:FSet{s}.  a ∪ b ∈ FSet{s}

mset_inter_wf_f:
  ∀s:DSet. ∀a,b:FSet{s}.  a ∩s b ∈ FSet{s}

mset_diff_wf_f:
  ∀s:DSet. ∀a,b:FSet{s}.  a - b ∈ FSet{s}

mset_inj_wf_f:
  ∀s:DSet. ∀x:|s|.  mset_inj{s}(x) ∈ FSet{s}
```

A multiset member function was introduced:

```
mset_mem:
  ∀s:DSet. ∀x:|s|. ∀a:MSet{s}.  x ∈_b  a = x ∈_b  a
```

where the $\in_b$ on the right denotes the mem function on lists.
and the multiset operations were recharacterized in terms of it:

```
mset_mem_sum:
  ∀s:DSet. ∀a,b:MSet{s}. ∀u:|s|.
    u ∈_b  a + b = (u ∈_b  a) ∨_b (u ∈_b  b)
```

```
mset_mem_union:
  ∀s:DSet. ∀as,bs:MSet{s}. ∀c:|s|.
    c ∈b as ∪ bs = (c ∈b as) ∨b (c ∈b bs)

mset_mem_inter:
  ∀s:DSet. ∀as,bs:MSet{s}. ∀c:|s|.
    c ∈b as ∩s bs = c ∈b as ∧b c ∈b bs

mset_mem_diff:
  ∀s:DSet. ∀as:FSet{s}. ∀bs:MSet{s}. ∀c:|s|.
    c ∈b as - bs = c ∈b as ∧b ¬b (c ∈b bs)

mem_bsubmset:
  ∀s:DSet. ∀a:FSet{s}. ∀b:MSet{s}.
    ↑(a ⊆b b) ⟺ (∀x:|s|. ↑(x ∈b a) ⇒ ↑(x ∈b b))
```

# Chapter 10

# Polynomials

## 10.1 Introduction

The aim of this chapter is to present a case study in the verification of an implementation of functions for multivariate polynomial arithmetic. The case study demonstrates how one can mix inductive and algebraic styles of reasoning, and the rich variety of symbolic manipulations possible in a theorem proving system.

The implementation verified is very similar to the basic sparse implementation described in texts on the design of computer algebra systems such as Davenport,Siret, Tournier [DST93] and Zippel [Zip93a]. This kind of implementation is in common use in current computer algebra systems. It involves representing a monomial by an association list (a-list) with indeterminates as keys [1] and indeterminate exponents as values, and representing a polynomial by an a-list with monomials as keys and monomial coefficients as values.

I based the ADT specification on the standard abstract mathematical characterization of multivariate polynomials found in say Lang [Lan84] or Bourbaki [Bou74]. The characterization defines algebraic structures for the monomials and polynomials over a given set of indeterminates and commutative ring of coefficients:

1. monomials are a free abelian monoid on the indeterminates.

2. polynomials are a free monoid algebra on the commutative ring of coefficients and the monoid of monomials.

I present in this chapter how I demonstrated in Nuprl that implementation types and functions for monomials and polynomials have all the abstract properties one would expect from this characterization. More specifically, I constructed the classes of all monomial and polynomial implementations and then proved that my implementations inhabit these classes.

[1] sometimes called *indices*

Since a-lists are used for both the monomial and polynomial implementations, I chose to first develop a common core theory of a-lists over a set of keys and over an abelian monoid of values. I showed that, algebraically, these a-lists have a monoid copower structure.

I then specialized this construction to get a free abelian monoid implementation and generalized it to get the free monoid algebra implementation.

In reading this chapter, bear in mind that it draws heavily on definitions and theorems introduced in Chapter 6 and Chapter 9.

## 10.2 Conventions Adopted

1. All `typewriter` font text presented in this chapter is taken verbatim from Nuprl library printouts. It is identical to what the user would see if interacting with Nuprl.

2. To clearly distinguish the nullary unary and binary concrete operations on a-lists from abstract operations on arbitrary classes, I in most cases double up the principle character of the display form for the concrete operations. For example, a-list 'addition' is denoted by `++`.

3. Many of the definitions here take parameters that I have chosen not to display. This significantly increases legibility of the Nuprl text. It should always be easy for the reader to infer what these parameters are from typing considerations. On occasion without specific mention, I have revealed certain parameters because their value is particularly informative.

   During proof, the user can switch the suppression of various parameters on and off in a few seconds, so the parameter hiding rarely causes a problem.

4. Definitions in this chapter are presented as *typed definitions*. These have form

   ```
   ∀x1:T1 ... ∀xn:Tn
     lhs
     =  rhs
     ∈ T
   ```

   Such a definition defines the term `lhs` in terms of the term `rhs`. This format combines information presented separately in other chapters in abstraction definitions and well-formedness lemmas. Recursive definitions are also presented as typed definitions. In these cases, there are one or more occurrences of the `lhs` term inside the `rhs` term.

The type parameter `T` of Nuprl's standard equality relation is displayed in these type definitions though it usually isn't displayed elsewhere in this chapter.

When a definition has more than one typing lemma, I have usually chosen to base the typed definition on the lemma with a smaller type `T`; such lemmas in general provide more information about the definition (though often too, these lemmas place more restrictions on the definition's arguments).

5. Both boolean and and propositional (type-valued) predicates are presented in this case study. Most boolean-valued predicates have a '$_b$' subscript in their display. The prefix function '$\uparrow$' converts boolean values to propositional values.

## 10.3 Specification

### 10.3.1 Monoid Copower

A monoid copower is a specialization of the categorical coproduct construction in the category of abelian monoids to the case when all the monoids that the product is being taken over are the same.

The signature for the monoid copower class is introduced in Figure 10.1 and the definition of the monoid class is given in Figure 10.2. The notation `a =`

```
Class Declaration for: MCopowerSig(s;g)

  Long Name: mcopower_sig
  Short Name: mcopower

  Parameters:
     s : DSet
     g : AbMon

  Fields:
     mon : AbMon
     inj : |s| → |g| → |mon|
     umap : h:AbMon → (|s| → |g| → |h|) → |mon| → |h|

  Universe: 𝕌'
```

Figure 10.1: Signature Class for Monoid Copowers

```
mcopower:
  ∀s:DSet. ∀g:AbMon.
    MCopower(s;g)
    = {c:MCopowerSig(s;g)|
       (∀j:|s|. IsMonHom{g,c.mon}(c.inj j))
       ∧ (∀h:AbMon. ∀f:|s| → MonHom(g,h).
            c.umap h f
             = !v:|c.mon| → |h|.
                IsMonHom{c.mon,h}(v)
                ∧ (∀j:|s|. f j = v o c.inj j ∈ |g| → |h|))}
    ∈ 𝕌'
```

Figure 10.2: Class of Monoid Copowers

`!x:T. Q[x]` can be read as "`a` is the unique `x` of type `T` such that `Q[x]` holds". Its definition was

```
uni_sat:
  a = !x:T. Q[x] == Q[a] ∧ (∀a':T. Q[a'] ⇒ a' = a)
```

I chose to make the construction in two stages so that the verification of implementations two could be split into two stages; one just involving basic type-checking of the implementation and the other involving checking all its algebraic properties. In the free abelian monoid case study described in Chapter 9, I had lumped everything into a single definition. Splitting things up didn't save any work, and probably made the definition slightly more verbose, but it seemed a natural thing to do, and I think it might make for easier-to-read definitions.

### 10.3.2 Free Abelian Monoid

The class definition for this was introduced in Section 9.2. The typed definition for the mapping from the class of monoid copowers into the class of free abelian monoids was:

```
fabmon_of_nat_mcp:
  ∀s:DSet. ∀m:MCopower(s;<ℤ+>↓hgrp).
    fabmon_of_nat_mcp(m)
    = <m.mon
      , λu.m.inj u zhgrp(1)
      , λm',f.m.umap m' (λz,n.(nat(n) ·m' (f z)))>
    ∈ FAbMon(s)
```

### 10.3.3 Free Monoid Algebra

The signature for the class of free monoid algebras is given in Figure 10.3 and the class definition is given in Figure 10.4.

```
Class Declaration for: FMASig(G;A)

  Long Name: fma_sig
  Short Name: fma

  Parameters:
     G : GrpSig
     A : RngSig

  Fields:
     alg : AlgebraSig(|A|)
     inj : |G| → |alg|
     umap : N:A-Algebra → (|G| → |N|) → |alg| → |N|

  Universe: 𝕌'
```

Figure 10.3: Signature Class for Free Monoid Algebras

```
fmonalg:
  ∀g:AbMon. ∀a:CRng.
    FMonAlg(g;a)
    = {m:FMASig(g;a)|
       IsMonHom{g,m.alg↓rg↓xmn}(m.inj)
       ∧ (∀n:a-Algebra. ∀f:MonHom(g,n↓rg↓xmn).
            m.umap n f
             = !f':|m.alg| → |n|.
                IsAlgHom{a,m.alg,n}(f')
                ∧ f' o m.inj = f ∈ |g| → |n|)}
    ∈ 𝕌'
```

Figure 10.4: Class of Free Monoid Algebras

### 10.3.4 Polynomial Algebra

The polynomial algebra class was introduced by the class declaration shown in Figure 10.5.

```
Class Declaration for: PolynomAlg(S;A)

  Long Name: polynom_alg
  Short Name: polyalg

  Parameters:

     S : DSet
     A : CRng

  Fields:

     mo : FAbMon(S)            % the monomials %
     poly : FMonAlg(mo.mon;A)  % the polynomials %

  Universe: 𝕌'
```

Figure 10.5: Class of Polynomial Algebras

## 10.4 Monoid Copower Construction

### 10.4.1 Ordered A-List Type

The basic type of a-lists with keys from a set `a` and values from an abelian monoid or group `b` is `(|a| × |b|) List`. There were extra restrictions I wanted to consider on the a-lists; that

1. The set `a` would be linearly ordered and the keys in an a-list would be in strictly descending order,
2. the values in an a-list would all be non-zero.

These restrictions define a canonical form on a-lists representing monomials or polynomials. Two a-lists represent the same monomial or polynomial just when their canonical forms are equal as a-lists.

The restriction of having the keys linearly ordered is a standard one in computer algebra; linear orders are commonly assumed on indeterminates and sets of monomials.

This canonical form is preserved by all the functions on a-lists that I defined.

Since I desired a constructive implementation of a-lists, I needed to explicitly generate an boolean-valued equality function for a-lists. The structure of this function exactly mirrors the structure of the underlying type `(|a| × |b|) List`, so I designed a collection of 'discrete set' constructors that paralleled the type constructors of Nuprl. Their typed definitions were:

```
eq_pair:
  ∀s,t:DSet. ∀a,b:|s| × |t|.
    a =_b b = (a.1 =_b b.1) ∧_b (a.2 =_b b.2) ∈ 𝔹
```

```
set_prod:
  ∀s,t:DSet.  s × t = mk_dset(|s| × |t|;λa,b.a =_b b) ∈ DSet
```

```
eq_list:
  ∀s:DSet. ∀as,bs:|s| List.
    as =_b bs
    = case as of
        [] => null(bs)
        a::as' => case bs of
                    [] => ff
                    b::bs' => (a =_b b) ∧_b (as' =_b bs')
                  esac
      esac
    ∈ 𝔹
```

```
dset_list:
  ∀s:DSet. (s List) = mk_dset(|s| List;λx,y.x =_b y) ∈ DSet
```

```
dset_set:
  ∀s:DSet. ∀Q:|s| → ℙ.
    {x:s| Q[x]} = mk_dset({x:|s|| Q[x]} ;=_b ) ∈ DSet
```

Using these definitions, I defined the discrete set of ordered a-lists as:

```
oalist:
  ∀a:LOSet. ∀b:AbMon.
    oalist(a;b)
    = {ps:(a × b↓set) List|
        ↑sd_ordered(map(λx.x.1;ps)) ∧ ¬↑mem(e,map(λx.x.2;ps))}
    ∈ DSet
```

Here, `mem` is the list membership function and `sd_ordered` is a predicate on lists over an ordered type, stating that the elements of the list are in strictly descending order.

```
mem:
  ∀s:DSet. ∀a:|s|. ∀bs:|s| List.
    mem(a,bs) = (For{<𝔹,∨_b >} x ∈ bs. x =_b  a) ∈ 𝔹

sd_ordered:
  ∀s:DSet. ∀as:|s| List.
    sd_ordered(as)
    = case as of
        [] => tt
        a::bs => before(a;bs) ∧_b sd_ordered(bs)
      esac
    ∈ 𝔹

before:
  ∀a:DSet. ∀ps:|a| List. ∀u:|a|.
    before(u;ps) = null(ps) ∨_b (hd(ps) <_b  u) ∈ 𝔹
```

I chose to make the `mem` function boolean-valued since it is a useful function to be able to compute. The `sd_ordered` function could have equally well been a propositional-valued predicate. An extra awkwardness working in Nuprl's constructive framework is that one constantly has to make choices as to whether to represent predicates as propositional or boolean-valued, and often one ends up doing a lot of inter-converting.

`LOSet` is a sub-class of `DSet`. It is the type of linearly-ordered discrete sets with computable inequality relations.

### 10.4.2 Ordered A-List Functions

The primary functions I introduced that assume the a-list values are drawn from an abelian monoid, are described by the type definitions:

```
oal_nil:
  ∀a:LOSet. ∀b:AbMon.  []a,b = [] ∈ |oalist(a;b)|

oal_inj:
  ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀v:|b|.
    inj(k,v)
    = if v =_b  e then [] else <k, v>::[] fi
    ∈ |oalist(a;b)|
```

```
oal_merge:
  ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oalist(a;b)|.
    ps ++ qs
    = if null(ps) then qs
      if null(qs) then ps
      if hd(qs).1 <_b hd(ps).1 then hd(ps)::(tl(ps) ++ qs)
      if hd(ps).1 <_b hd(qs).1 then hd(qs)::(ps ++ tl(qs))
      if (hd(ps).2 * hd(qs).2) =_b e then tl(ps) ++ tl(qs)
      else <hd(ps).1, hd(ps).2 * hd(qs).2>::(tl(ps) ++ tl(qs))
      fi
    ∈ |oalist(a;b)|

oal_lk:
  ∀s:LOSet. ∀g:AbMon. ∀ps:|oalist(s;g)|.
    ¬(ps = []s,g ∈ |oalist(s;g)|) ⇒ lk(ps) = hd(ps).1 ∈ |s|

oal_lv:
  ∀s:LOSet. ∀g:AbMon. ∀ps:|oalist(s;g)|.
    ¬(ps = []s,g ∈ |oalist(s;g)|) ⇒ lv(ps) = hd(ps).2 ∈ |g|

oal_null:
  ∀s:LOSet. ∀g:AbMon. ∀ps:|oalist(s;g)|.
    null(ps) = null(ps) ∈ 𝔹
```

Note that in the `oal_null` definition, the function displayed on the left as `null` is the new definition and the function displayed on the right as `null` is a previously defined function. The difference is that the new definition has some extra unshown parameters that help with type-checking it.

Function definitions I introduced that are suitable when the a-list values come from an abelian group are:

```
oal_neg:
  ∀a:LOSet. ∀b:AbMon. ∀ps:|oalist(a;b)|.
    --ps = map(λkv.<kv.1, ~ kv.2>;ps) ∈ |oalist(a;b)|

oal_bpos:
  ∀s:LOSet. ∀g:AbGrp. ∀ps:|oalist(s;g)|.
    pos(ps) = ¬_b null(ps) ∧_b (e <_b lv(ps)) ∈ 𝔹

oal_blt:
  ∀s:LOSet. ∀g:AbGrp. ∀ps,qs:|oalist(s;g)|.
    ps <<_b qs = pos(qs ++ --ps) ∈ 𝔹

oal_ble:
  ∀s:LOSet. ∀g:AbGrp. ∀ps,qs:|oalist(s;g)|.
    ps ≤≤_b qs = (ps =_b qs) ∨_b (ps <<_b qs) ∈ 𝔹
```

The order relations introduced here define a standard lexicographic ordering on a-lists.

By waiting to define the order function on the a-lists until I was assuming that the values were drawn from a linearly ordered group rather than a linearly ordered monoid, I was able to define the function in terms of a-list subtraction. Note that when the oalist type is specialized to that of monomials, the structure instantiating the type of the a-list values, namely the naturals under addition, is a monoid and not a group. However, still I needed the order function over monomials in order to just *define* the type of polynomials (recall that when a-lists are used for polynomials, the monomials are the keys and the keys must be over an ordered set).

Note that typings ascribed to functions are sometimes more permissive than the environments in which they are typically used. For example, all the above functions, with values from a group, are only used when the $<$ relation on the group is assumed to be linear order and the group operation is assumed to be monotone.

### 10.4.3 Basic Properties of Functions

The algebraic properties of a-lists are most clearly revealed when they are considered to be functions of finite support. In fact, the standard 'concrete' construction of monomials and polynomials in algebra text-books starts from this point. The two functions that enable this view of a-lists are:

```
oal_dom:
  ∀a:LOSet. ∀b:AbMon. ∀ps:|oalist(a;b)|.
    dom(ps) = mk_mset(map(λz.z.1;ps)) ∈ FSet{a}

lookup:
  ∀a:PosetSig. ∀B:𝕌. ∀z:B. ∀k:|a|. ∀xs:(|a| × B) List.
    xs[k]{a,z}
    = case xs of
        [] => z
        b::bs => let <bk,bv> = b
                 in
                 if bk =_b a k then bv else bs[k]{a,z} fi
      esac
    ∈ B
```

Note that the `lookup` function returns the default value `z` when the key being looked up in an a-list doesn't match any of the keys in the list. The a-lists described in this chapter always have values drawn from either monoids or rings, and this default value is either the monoid identity or the ring zero respectively. The truth of many of the theorems that are presented in this chapter relies on the default values being what they are. Since this default-value argument to the `lookup` function can be easily inferred, I normally suppress it. I also suppress the domain set argument (`a` in the definition) since this too can be easily inferred.

Trivially, since every a-list only has a finite number of keys, the `lookup` function must be a function of finite support.

As is shown later, it was essential to be able to compute the exact support of any a-list. The `oal_dom` function does this. Note that it returns a *finite set* as defined in Section 9.5, not a list. In this chapter, all functions that make use of the result of this domain function are typed to expect a finite set, or more generally, a finite multiset.

The following lemmas spell out the relationship between a-lists and the `lookup` and `domain` functions.

```
lookup_non_zero:
  ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps:|oalist(a;b)|.
    ¬(ps[k] = e ∈ |b|) ⟺ ↑msMem(k,dom(ps))

lookups_same_a:
  ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oalist(a;b)|.
    (∀u:|a|. ps[u] = qs[u] ∈ |b|) ⇒ ps = qs ∈ |oalist(a;b)|
```

The lemma `lookup_non_zero` says that the `lookup` function is surjective onto the set of functions of finite support. The lemma `lookups_same_a` says that `lookup` function is injective into the set of functions of finite support. Therefore the `lookup` function is a bijection between the oalists and the functions of finite support.

The next step was to characterize the value of the `lookup` and `oal_dom` functions when applied to the a-list operations of merging, injection and negation. defined above.

```
oal_dom_merge:
  ∀a:LOSet. ∀b:AbMon. ∀ps,qs:|oalist(a;b)|.
    ↑(dom(ps ++ qs) ⊆_b  dom(ps) ∪ dom(qs))

lookup_merge:
  ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀ps,qs:|oalist(a;b)|.
    (ps ++ qs)[k] = ps[k] * qs[k]

oal_dom_inj:
  ∀a:LOSet. ∀b:AbMon. ∀k:|a|. ∀v:|b|.
    dom(inj(k,v))
    = if v =_b  e then 0a else inj(k) fi
    ∈ FSet{a}

lookup_oal_inj:
  ∀a:LOSet. ∀b:AbMon. ∀k,k':|a|. ∀v:|b|.
    inj(k,v)[k'] = when k =_b  k' . v
```

```
oal_dom_neg:
  ∀a:LOSet. ∀b:AbGrp. ∀ps:|oalist(a;b)|.
    dom(--ps) = dom(ps) ∈ FSet{a}

lookup_oal_neg:
  ∀a:DSet. ∀b:IGroup. ∀k:|a|. ∀ps:(|a| × |b|) List.
    (--ps)[k] = ~ ps[k] ∈ |b|
```

Using these lemmas it was then trivial to show that the functions `oal_nil`, `oal_merge` and `oal_neg` defined an abelian group structure on oalists whenever the a-list values were over a group, and similarly how they defined an abelian monoid when the values were over a monoid.

```
oal_mon:
  ∀a:LOSet. ∀b:AbMon.
    oal_mon(a;b)
    = <|oalist(a;b)|, =b , =b , λx,y.x ++ y, []a,b, λx.x>
    ∈ AbMon

oal_grp:
  ∀s:LOSet. ∀g:AbGrp.
    oal_grp(s;g)
    = <|oalist(s;g)|
      , =b
      , λx,y.x ≤≤b y
      , λx,y.x ++ y
      , []s,g
      , λx.--x>
    ∈ AbGrp
```

### 10.4.4 Linear Monotonic Order on A-Lists

The `oal_blt` relation ($<<_b$) is a lexicographic order. When the a-lists are representing monomials, it corresponds to the lexicographic monomial ordering, common in computer algebra. For the purposes of this case-study, it was not necessary to show that this order is well-founded; indeed in general here is it not, because I don't adopt any assumption about the well-foundedness of a-list keys.

To show that `oal_blt` is a lexicographic order, I created a standard definition of a lexicographic order

```
oal_lt:
  ∀s:LOSet. ∀g:OCMon. ∀ps,qs:|oalist(s;g)|.
    (ps << qs)
    = (∃k:|s|
        (∀k':|s|. k <s k' ⇒ ps[k'] = qs[k'])
        ∧ ps[k] < qs[k])
    ∈ ℙ
```

and proved that `oal_blt` was logically equivalent to this definition.

```
assert_of_oal_blt:
  ∀s:LOSet. ∀g:OCGrp. ∀ps,qs:|oalist(s;g)|.
    ↑(ps <<_b qs) ⟺ ps << qs
```

It was straightforward to verify that `oal_lt` and hence `oal_blt` was irreflexive and transitive.

```
oal_lt_irrefl:
  ∀s:LOSet. ∀g:OCMon.  Irrefl(|oal(s;g)|;ps,qs.ps << qs)

oal_lt_trans:
  ∀s:LOSet. ∀g:OCMon.  Trans(|oal(s;g)|;ps,qs.ps << qs)
```

To show that `oal_blt` was a linear order it was simplest to first prove a trichotomy-like property for the `oal_bpos` definition.

```
oal_bpos_trichot:
  ∀s:LOSet. ∀g:OCGrp. ∀rs:|oalist(s;g)|.
    ↑pos(rs) ∨ rs = []s,g ∈ |oalist(s;g)| ∨ ↑pos(--rs)

oal_lt_trichot:
  ∀s:LOSet. ∀g:OCGrp. ∀ps,qs:|oalist(s;g)|.
    ps <{s,g} qs ∨ ps = qs ∈ |oalist(s;g)| ∨ qs <{s,g} ps
```

For later verification, I needed that the `oal_merge` function was monotonic with respect to the `oal_lt` relation:

```
oal_merge_preserves_lt:
  ∀s:LOSet. ∀g:OCMon. ∀ps,qs,rs:|oalist(s;g)|.
    qs << rs ⇒ ps ++ qs << ps ++ rs
```

In retrospect, it probably would have been slightly easier to derive all the properties of `oal_blt` from equivalent properties of `oal_bpos`.

All the above properties on the `oal_blt` relation induce corresponding properties on `oal_ble`, its reflexive closure. Once these properties, were verified, I was able to prove a second lemma for `oal_grp`:

```
oal_grp_wf2:
  ∀s:LOSet. ∀g:OGrp.  oal_grp(s;g) ∈ OGrp
```

As it stood, this lemma was too specialized to apply to monomials, since it required the a-list values to be drawn from a group, whereas with monomials, the values come from a monoid. To fix this, I proved the following lemma, which simply says that any `g:GrpSig` that can be isomorphically embedded into an `OCMon`, is itself an `OCMon`.

```
inj_into_ocmon:
  ∀g:GrpSig
    (∃h:OCMon
      ∃f:|g| → |h|
       IsMonHomInj(g;h;f)
       ∧ RelsIso(|g|;|h|;x,y.↑(x =b y);x,y.↑(x =b y);f)
       ∧ RelsIso(|g|;|h|;x,y.↑(x g.le y);x,y.↑(x h.le y);f))
    ⇒ g ∈ OCMon
```

where the typed definition for `RelsIso` is

```
rels_iso:
  ∀T,T':𝕌. ∀R:T → T → ℙ. ∀R':T' → T' → ℙ. ∀f:T → T'.
    RelsIso(T;T';x,y.R[x;y];x,y.R'[x;y];f)
    = (∀x,y:T.  R[x;y] ⟺ R'[f x;f y])
    ∈ ℙ
```

I then applied the half group construction described in Section 6.6 to the linearly-ordered group of values of ordered a-lists to yield a corresponding construction on the a-lists themselves.

```
oal_hgp:
  ∀s:LOSet. ∀g:OGrp.
    oal_hgp(s;g)
    = <|oalist(s;g↓hgrp)|
      , =b
      , λx,y.x ≤≤b y
      , λx,y.x ++ y
      , []s,g↓hgrp
      , λx.x>
    ∈ OCMon
```

Now, since the naturals are a half group of the group of integers under addition, I could instantiate this theorem with the group of integers as the 'extended' a-list values domain, and get that a-lists with the group of naturals under addition as values form an ordered cancellation monoid.

### 10.4.5 Monoid Copower Assembly

The universal mapping function is defined in terms of the multiset summation function `msFor` introduced in Section 9.3.2:

```
oal_umap:
  ∀s:LOSet. ∀g,h:AbMon. ∀f:|s| → |g| → |h|.
    umap(h,f)
    = (λps:|oalist(s;g)|. msFor{h} k ∈ dom(ps). f k ps[k])
    ∈ |oalist(s;g)| → |h|
```

Once characterizing lemmas about the `oal_inj` and `oal_umap` definitions were proven, assembly of the monoid copower was trivial.

```
oal_inj_mon_hom:
  ∀a:LOSet. ∀b:AbMon. ∀k:|a|.
    IsMonHom{b,oal_mon(a;b)}(λv.inj(k,v))

oal_umap_char_a:
  ∀s:LOSet. ∀g,h:AbMon. ∀f:|s| → MonHom(g,h).
    umap(h,f)
     = !v:|oalist(s;g)| → |h|.
        IsMonHom{oal_mon(s;g),h}(v)
        ∧ (∀j:|s|. f j = v o (λw.inj(j,w)) ∈ |g| → |h|)

oal_omcp:
  ∀s:LOSet. ∀g:OGrp.
    oal_omcp{s,g}
    = <oal_hgp(s;g), λk,v.inj(k,v), λh,f.umap(h,f)>
    ∈ MCopower(s;g↓hgrp)
```

## 10.5 Free Monoid Algebra Construction

Here, I assumed more structure on the a-list keys and values. Specifically, I assumed that the keys come from a monoid and the values from a commutative ring. Remember that from here on, the default value returned by the a-list function `lookup` (`as[k]`) is the zero of the values ring of the a-lists.

### 10.5.1 Lifting of Definitions and Theorems

For convenience, I first lifted many of the definitions and theorems from the theory of a-lists developed previously. The convenience was chiefly cosmetic; it enabled me to avoid having an abundance of forgetful functors floating around and saved on typing.

The type-checking tactics can cope with these functors, though they can slow it down. An example definition that wasn't lifted was the `lookup` function.

The lifting was very straightforward; I wrote a simple ML function that automatically lifted all the necessary theorems and entering the new definitions was very quick. However, this exercise did indicate the desirability of a better handling of class subtyping where forgetful functors, if needed at all, are always inserted automatically.

Here is a sampling of the lifted definitions with parameter suppression disabled so that the forgetful functors are visible.

```
omralist:
  ∀g:OCMon. ∀r:CRng.
    omralist(g;r) = oalist(g↓set;r↓+gp) ∈ DSet

omral_dom:
  ∀g:OCMon. ∀r:CRng. ∀ps:|omralist(g;r)|.
    dom{g,r}(ps) = dom{g↓set,r↓+gp}(ps) ∈ FSet{g↓set}

omral_plus:
  ∀g:OCMon. ∀r:CRng. ∀ps,qs:|omralist(g;r)|.
    ps ++g,r qs = ps ++g↓set,r↓+gp qs ∈ |omralist(g;r)|

omral_zero:
  ∀g:OCMon. ∀r:CRng.
    00g,r = []g↓set,r↓+gp ∈ |omralist(g;r)|

omral_minus:
  ∀g:OCMon. ∀r:CRng. ∀ps:|omralist(g;r)|.
    --g,r ps = --g↓set,r↓+gp ps ∈ |omralist(g;r)|

omral_inj:
  ∀g:OCMon. ∀r:CRng. ∀k:|g|. ∀v:|r|.
    inj{g,r}(k,v) = inj{g↓set,r↓+gp}(k,v) ∈ |omralist(g;r)|
```

### 10.5.2 Multiplicative Functions on Ordered A-Lists

The multiplicative functions that I defined to create an algebra structure on a-lists were:

```
omral_scale:
  ∀g:OCMon. ∀r:CRng. ∀k:|g|. ∀v:|r|. ∀ps:|omralist(g;r)|.
    <k,v>* ps
    = case ps of
        [] => []
        p::ps' => if (v * p.2) =b 0
                  then <k,v>* ps'
                  else <k * p.1, v * p.2>::(<k,v>* ps')
                  fi
      esac
    ∈ |omralist(g;r)|

omral_times:
  ∀g:OCMon. ∀r:CRng. ∀ps,qs:|omralist(g;r)|.
    ps ** qs
    = case ps of
        [] => []
        p::ps' => <p.1,p.2>* qs ++ (ps' ** qs)
      esac
    ∈ |omralist(g;r)|
```

```
omral_one:
  ∀g:OCMon. ∀r:CRng.  11g,r = inj(e,1r) ∈ |omralist(g;r)|

omral_action:
  ∀g:OCMon. ∀r:CRng. ∀v:|r|. ∀ps:|omralist(g;r)|.
    v ·· ps = <e,v>* ps ∈ |omralist(g;r)|
```

As before, the key to deriving these functions' algebraic properties was to consider the a-alists as functions of finite support. The theorems I proved included

```
omral_dom_scale:
  ∀g:OCMon. ∀r:CRng. ∀k:|g|. ∀v:|r|. ∀ps:|omralist(g;r)|.
    ↑(dom(<k,v>* ps) ⊆_b  fs-map(λk'.k' * k, dom(ps)))

lookup_omral_scale_d:
  ∀g:OCMon. ∀r:CRng. ∀z,k:|g|. ∀v:|r|. ∀ps:|omralist(g;r)|.
    (<k,v>* ps)[z]
    = (Σr y ∈ dom(ps). when (k * y) =_b  z. v * ps[y])
    ∈ |r|

omral_times_dom:
  ∀g:OCMon. ∀r:CRng. ∀ps,qs:|omralist(g;r)|.
    ↑(dom(ps ** qs) ⊆_b  dom(ps) ×g dom(qs))

lookup_omral_times_a:
  ∀g:OCMon. ∀r:CRng. ∀ps,qs:|omralist(g;r)|. ∀z:|g|.
    (ps ** qs)[z]
    = (Σr x ∈ dom(ps)
        Σr y ∈ dom(qs). when (x * y) =_b  z. ps[x] * qs[y])
    ∈ |r|

omral_dom_action:
  ∀g:OCMon. ∀r:CRng. ∀v:|r|. ∀ps:|omralist(g;r)|.
    ↑(dom(v ·· ps) ⊆_b  dom(ps))

lookup_omral_action:
  ∀g:OCMon. ∀r:CRng. ∀k:|g|. ∀v:|r|. ∀ps:|omralist(g;r)|.
    (v ·· ps)[k] = v * ps[k] ∈ |r|
```

### 10.5.3 Properties of Multiplicative Functions

Using the above characterization I proved:

```
omral_times_assoc_a:
  ∀g:OCMon. ∀a:CRng. ∀ps,qs,rs:|omralist(g;a)|.
    ps ** (qs ** rs) = (ps ** qs) ** rs
```

```
omral_times_comm_a:
  ∀g:OCMon. ∀a:CRng. ∀ps,qs:|omralist(g;a)|.
    ps ** qs = qs ** ps

omral_bilinear_a:
  ∀g:OCMon. ∀a:CRng. ∀ps,qs,rs:|omral(g;a)|.
    ps ** (qs ++ rs) = (ps ** qs) ++ (ps ** rs)
    ∧ (qs ++ rs) ** ps = (qs ** ps) ++ (rs ** ps)

omral_times_ident_l:
  ∀g:OCMon. ∀r:CRng. ∀ps:|omralist(g;r)|.
    11 ** ps = ps

omral_times_ident_r:
  ∀g:OCMon. ∀r:CRng. ∀ps:|omralist(g;r)|.
    ps ** 11 = ps

omral_action_one:
  ∀g:OCMon. ∀r:CRng. ∀ps:|omralist(g;r)|.
    1 ·· ps = ps

omral_action_times:
  ∀g:OCMon. ∀r:CRng. ∀v,w:|r|. ∀ps:|omralist(g;r)|.
    (v * w) ·· ps = v ·· (w ·· ps)

omral_action_times_r1:
  ∀g:OCMon. ∀r:CRng. ∀v:|r|. ∀ps,qs:|omralist(g;r)|.
    v ·· (ps ** qs) = (v ·· ps) ** qs

omral_action_times_r2:
  ∀g:OCMon. ∀r:CRng. ∀v:|r|. ∀ps,qs:|omralist(g;r)|.
    v ·· (ps ** qs) = ps ** (v ·· qs)

omral_action_plus_l:
  ∀g:OCMon. ∀r:CRng. ∀v,w:|r|. ∀ps:|omralist(g;r)|.
    (v + w) ·· ps = (v ·· ps) ++ (w ·· ps)

omral_action_plus_r:
  ∀g:OCMon. ∀r:CRng. ∀v:|r|. ∀ps,qs:|omralist(g;r)|.
    v ·· (ps ++ qs) = (v ·· ps) ++ (v ·· qs)
```

and hence was able to assemble the algebra structure:

```
omral_alg:
  ∀g:OCMon. ∀r:CRng.
    omral_alg(g;r)
    = <|omralist(g;r)|
      , =b
      , λx,y.tt
      , λx,y.x ++ y
      , 00
      , λx.--x
      , λx,y.x ** y
      , 11
      , λx,y.(inr · )
      , λa,x.a ·· x>
    ∈ AlgebraSig(|r|)
```

### 10.5.4 Assembly of Free Monoid Algebra

For the free monoid algebra, I needed the definition for the universal mapping function

```
omral_alg_umap:
  ∀g:OCMon. ∀a:CRng. ∀n:a-Algebra. ∀f:|g| → |n|.
    alg_umap{g,a}(n,f)
    = (λps:|omral(g;a)|. Σ{g↓oset,n↓rg} k ∈ dom{g,a}(ps).
                            ps[k]{g↓oset,0a} ·n (f k))
    ∈ |omral(g;a)| → |n|
```

and with several characterizing theorems about this and the `omral_inj` function:

```
omral_action_inj:
  ∀g:OCMon. ∀r:CRng. ∀k:|g|. ∀v,v':|r|.
    v ·· inj(k,v') = inj(k,v * v') ∈ |omralist(g;r)|

omral_inj_mon_op:
  ∀g:OCMon. ∀r:CRng. ∀k,k':|g|.
    inj(k * k',1) = inj(k,1) ** inj(k',1) ∈ |omralist(g;r)|

omral_alg_umap_is_hom:
  ∀g:OCMon. ∀a:CRng. ∀n:a-Algebra. ∀f:MonHom(g,n↓rg↓xmn).
    IsAlgHom{a,omral_alg(g;a),n}(alg_umap(n,f))

omral_alg_umap_tri_comm:
  ∀g:OCMon. ∀a:CRng. ∀n:a-Algebra. ∀f:|g| → |n|.
    alg_umap(n,f) o (λk.inj(k,1)) = f
```

```
omral_alg_umap_unique:
  ∀g:OCMon. ∀a:CRng. ∀n:a-Algebra. ∀f:|g| → |n|.
  ∀f':a-AlgebraHom(omral_alg(g;a);n).
    f' o (λk:|g|. inj(k,1)) = f ⇒ f' = alg_umap(n,f)
```

I demonstrated that a-lists over ordered monoids as keys and commutative rings as values are a free monoid algebra.

```
omral_fma:
  ∀g:OCMon. ∀a:CRng.
    omral_fma(g;a)
    = <omral_alg(g;a), λk.inj(k,1), λn,f.alg_umap{g,a,n,f}>
    ∈ FMASig(g;a)
```

## 10.6 Polynomial Algebra Assembly

This was very straightforward, given all the previous work. Here are the typed definitions:

```
oal_fabmon:
  ∀s:LOSet
    oal_fabmon(s)
    = fabmon_of_nat_mcp(oal_omcp{s,<ℤ+>})
    ∈ FAbMon(s)

oal_polyalg:
  ∀s:LOSet. ∀a:CRng.
    oal_polyalg(s;a)
    = <oal_fabmon(s), omral_fma(oal_fabmon(s).mon;a)>
    ∈ PolynomAlg(s;a)
```

## 10.7 Proof Examples

Here I have selected a few proof fragments to indicate the styles of reasoning used in building this theory.

### 10.7.1 Algebraic Manipulation of Sums

The theorem I am studying here is

```
omral_times_assoc_a:
  ∀g:OCMon. ∀a:CRng. ∀ps,qs,rs:|omral(g;a)|.
    ps ** (qs ** rs) = (ps ** qs) ** rs
```

In Figure 10.6, quantifiers are stripped and attention is focussed on the value of the product expressions at the a-list key u. The context in hypotheses 1-6 remains unchanged for the rest of the proof, so I have elided it in subsequent displays. Also, to simplify the presentation I have elided the right-hand side of the conclusion equality and the tactics that affect it. However, since many tactics work on both the left and right-hand sides simultaneously, the full proof involves only a third more tactic invocations than are shown here.

```
⊢ ∀g:OCMon. ∀a:CRng. ∀ps,qs,rs:|omral(g;a)|.
     ps ** (qs ** rs) = (ps ** qs) ** rs

BY (RepD
    THENM BLemma `omral_lookups_same_a`
    THENM RepD ...a)

1. g: OCMon
2. a: CRng
3. ps: |omral(g;a)|
4. qs: |omral(g;a)|
5. rs: |omral(g;a)|
6. u: |g|
⊢ (ps ** (qs ** rs))[u] = ((ps ** qs) ** rs)[u]
```

Figure 10.6: Focussing Attention at u.

The goal of the proof is to put the left and right-hand sides in some common form. Consider the left-hand side of the equality in omral_times_assoc_a, expanded using the lemma lookup_omral_times_a:

```
Σx ∈ dom(ps).
 Σy ∈ dom(qs ** rs).
  when (x * y) =_b u.
    ps[x]
    * (Σx1 ∈ dom(qs).
        Σy1 ∈ dom(rs). when (x1 * y1) =_b y. qs[x1] * rs[y1])
```

The strategy is to bring the summation over y and the expression when (x1 * y1) $=_b$ y together and then cancel them using the following lemma for summation over a single value:

```
rng_fset_for_when_eq:
  ∀s:DSet. ∀r:Ring. ∀f:|s| → |r|. ∀e:|s|. ∀as:FSet{s}.
    ↑(e ∈_b as) ⇒ (Σx ∈ as. when x =_b e. f[x]) = f[e]
```

Note the precondition of this lemma `↑(e ∈_b as)`. This precondition is not always satisfied because given `x1 ∈ dom(qs)` and `y1 ∈ dom(rs)`, it is not always true that `x1 * y1 ∈ dom(qs ** rs)` (cancellation may occur). Fortunately, the relation

```
omral_times_dom:
  ∀g:OCMon. ∀r:CRng. ∀ps,qs:|omral(g;r)|.
    ↑(dom(ps ** qs) ⊆_b dom(ps) × dom(qs))
```

holds and the value of the summation over `y` doesn't change when the summation range is widened from `dom(qs ** rs)` to `dom(qs) × dom(qs)`, since its argument is zero in the intervening range. This strategy was formalized as follows.

In Figure 10.7, the lookup of the outer is expanded, but not the inner product. In Figure 10.8, the widening of the domain of summation of `y` is carried out.

```
⊢ (ps ** (qs ** rs))[u] = ...

BY (RWO "lookup_omral_times_a" 0 ...a)

⊢ (Σx ∈ dom(ps).
      Σy ∈ dom(qs ** rs).
       when (x * y) =_b u. ps[x] * (qs ** rs)[y])
   = ...
```

Figure 10.7: Expanding Outer Lookup

Note that this is an example of mononicity reasoning being done by the rewrite package. The first sub-goal is to check the condition that the summation value really is zero on the range being widened over. It is generated because the rewrite package selected the functionality lemma for the summation term

```
rng_mssum_functionality_wrt_bsubmset:
  ∀s:DSet. ∀r:Ring. ∀f,f':|s| → |r|. ∀p,q:MSet{s}.
    (∀x:|s|. ↑(x ∈_b q - p) ⇒ f'[x] = 0)
    ⇒ ↑(p ⊆_b q)
    ⇒ (∀x:|s|. ↑(x ∈_b p) ⇒ f[x] = f'[x])
    ⇒ (Σx ∈ p. f[x]) = (Σx ∈ q. f'[x])
```

when justifying the rewrite. Note how the functionality lemma takes care of introducing the necessary assumptions 7 and 9 in the subgoal.

```
⊢ (Σx ∈ dom(ps).
     Σy ∈ dom(qs ** rs).
      when (x * y) =_b u. ps[x] * (qs ** rs)[y])
   = ...

BY (RWO "omral_times_dom" 0 ...a)

 7. x: |(g↓set)|
 8. ↑(x ∈_b dom(ps))
 9. y: |(g↓set)|
 10. ↑(y ∈_b (dom(qs) ×g dom(rs)) - dom(qs ** rs))
 ⊢ when (x * y) =_b u. ps[x] * (qs ** rs)[y] = 0

 ⊢ (Σx ∈ dom(ps).
      Σy ∈ dom(qs) ×g dom(rs).
       when (x * y) =_b u. ps[x] * (qs ** rs)[y])
    = ...
```

Figure 10.8: Expanding Domain of Summation over y

```
7. x: |(g↓oset)|
8. ↑(x ∈_b dom(ps))
9. y: |(g↓oset)|
10. ↑(y ∈_b (dom(qs) ×g dom(rs)) - dom(qs ** rs))
⊢ when (x * y) =_b u. ps[x] * (qs ** rs)[y] = 0

BY (RWN 2 (LemmaC ‘lookup_omral_eq_zero‘) 0
    THENM RW RngNormC 0
    THENM RWH (LemmaC ‘rng_when_of_zero‘) 0 ...)

⊢ ¬↑(y ∈_b dom(qs ** rs))

BY (RWH (LemmaC ‘mset_mem_diff‘) 10
    THENM RW bool_to_propC 10 ...)
```

Figure 10.9: Proof of Summation Arg being 0 in Widening Range

The proof of this subgoal is shown in Figure 10.9. In the first step, I noted that the lookup on the a-list `qs ** rs` is `0`, whereupon the whole expression on the left-hand side of the equality simplifies to `0`. The resulting goal of `0 = 0` is not shown because it is trivially proved by the `Auto` tactic. The goal

```
¬↑(y ∈_b  dom(qs ** rs))
```

was a side condition of the `lookup_omral_eq_zero` lemma; its statement and the statement of the `rng_when_of_zero` lemma were:

```
lookup_omral_eq_zero:
  ∀g:OCMon. ∀r:CRng. ∀k:|g|. ∀ps:|omral(g;r)|.
    ¬↑(k ∈_b  dom(ps)) ⇒ ps[k] = 0

rng_when_of_zero:
  ∀r:Ring. ∀b:𝔹.  when b. 0 = 0
```

The latter lemma could easily have been folded into the ring normalization conversion `RngNormC`. The tactic `RWN` $n$ $c$ $i$ applies conversion $c$ to clause $i$, but only does the conversion in the $n$th position in pre-order that $c$ is enabled. Note that the lemma `lookup_omral_eq_zero` could not have been applied if back in the step of the proof shown in Figure 10.7, the lemma `lookup_omral_times_a` had been repeatedly applied where-ever it could have made progress.

The antecedent of the `lookup_omral_eq_zero` lemma was solved by applying the lemma

```
mset_mem_diff:
  ∀s:DSet. ∀as:FSet{s}. ∀bs:MSet{s}. ∀c:|s|.
    c ∈_b  as - bs = (c ∈_b  as) ∧_b ¬_b (c ∈_b  bs)
```

and converting the boolean-valued hypothesis to a proposition using the conversion `bool_to_propC`.

In Figure 10.10, the next 3 steps of the proof are shown. Here, the inner multiplication was expanded and the summation over `y` was brought adjacent to the `when` expression containing `y` as one of subjects of the equality.

The kind of rewriting required in the last step (with the `HereDnC` conversional) is tricky. The lemmas used are:

```
rng_mssum_swap:
  ∀r:Ring. ∀s,s':DSet. ∀f:|s| → |s'| → |r|. ∀a:MSet{s}.
  ∀b:MSet{s'}.
    (Σx ∈ a. Σy ∈ b. f[x;y]) = (Σy ∈ b. Σx ∈ a. f[x;y])

rng_when_swap:
  ∀r:Ring. ∀b,b':𝔹. ∀p:|r|.
    when b. when b'. p = when b'. when b. p
```

```
⊢ (Σx ∈ dom(ps).
    Σy ∈ dom(qs) ×g dom(rs).
     when (x * y) =b u. ps[x] * (qs ** rs)[y])
  = ...

BY (RWW "lookup_omral_times_a" 0 ...a)

⊢ (Σx ∈ dom(ps).
    Σy ∈ dom(qs) ×g dom(rs).
     when (x * y) =b u.
       ps[x]
       * (Σx ∈ dom(qs).
           Σy1 ∈ dom(rs).
            when (x * y1) =b y. qs[x] * rs[y1]))
  = ...

BY % float up sigmas and whens %
  (RWW "rng_times_mssum_l
        rng_times_mssum_r
        rng_mssum_when_swap<
        rng_times_when_l
        rng_times_when_r" 0 ...a)

⊢ (Σx ∈ dom(ps).
    Σy ∈ dom(qs) ×g dom(rs).
     Σx1 ∈ dom(qs).
      Σy1 ∈ dom(rs).
       when (x * y) =b u.
         when (x1 * y1) =b y. ps[x] * (qs[x1] * rs[y1]))
  = ...

BY (RWN 2 (HereDnC (PolyC "rng_mssum_swap rng_when_swap")) 0
  ...a)

⊢ (Σx ∈ dom(ps).
    Σx1 ∈ dom(qs).
     Σy1 ∈ dom(rs).
      Σy ∈ dom(qs) ×g dom(rs).
       when (x1 * y1) =b y.
         when (x * y) =b u. ps[x] * (qs[x1] * rs[y1]))
  = ...
```

Figure 10.10: Bringing $\Sigma$ and `when` over `y` Together

The first lemma needed to be applied repeatedly, but in a controlled way to have the right effect. The conversion `HereDnC` $c$ when applied to a term $t$, tries applying $c$ at the top of $t$. If it succeeds, it then tries applying $c$ on all subterms of $t$ by applying $c$ once at each node of $t$ and then at the children of that node. Its definition in ML was:

```
let HereDnC c = c ANDTHENC TryC (SubC (SweepDnC c)) ;;
```

I only used this conversion a couple of times in this case study, but its ease of definition demonstrates the versatility of the conversional approach to structuring rewrite rules. Of course, in the long run such rewriting should be directed by higher level reasoning. For instance, one could envisage directing a rewrite tactic to 'bring the $\Sigma$ and `when` involving `y` together'. The tactic might then create a suitable cost function to direct the rewrite equations.

The cancellation of the $\Sigma$ and `when` is shown in Figure 10.11. Note again how a functionality lemma, in this case:

```
rng_mssum_functionality_wrt_equal:
  ∀s:DSet. ∀r:Ring. ∀f,f':|s| → |r|. ∀a,a':MSet{s}.
    a = a'
    ⇒ (∀x:|s|. ↑(x ∈_b  a) ⇒ f[x] = f'[x])
    ⇒ (Σx ∈ a. f[x]) = (Σx ∈ a'. f'[x])
```

takes care of introducing the necessary assumptions for the subgoal created by the antecedent of `rng_fset_for_when_eq`.

The initial proof step in Figure 10.11 is necessary to set things up right for a match to be generated against the `rng_fset_for_when_eq` lemma. The effect of the `grp_eq_sym` lemma is clear, but the effect of the conversion `dset_of_monC` is hidden. The problem is that the expression

```
(x1 * y1) =_b  y
```

at the top of Figure 10.11 and the expression

```
x =_b  e
```

in lemma `rng_fset_for_when_eq` really involve different equalities. If hidden parameters are made visible, giving respectively

```
(x1 * y1) =_b g y
```

where `g` is an element of the `OCMon` class and

```
x =_b s  e
```

where `s` is an element of the `DSet` class, the difference is more apparent. One `=` is the equality function on the `OCMon` class and the other is the equality function on the `DSet` class. The conversion `dset_of_monC` adds in an appropriate forgetful functor from `OCMon` to `DSet`. More specifically, it rewrites:

```
⊢ (Σx ∈ dom(ps).
     Σx1 ∈ dom(qs).
      Σy1 ∈ dom(rs).
       Σy ∈ dom(qs) × dom(rs).
        when (x1 * y1) =b  y.
          when (x * y) =b  u. ps[x] * (qs[x1] * rs[y1]))
   = ...

BY (RWN 1 (LemmaC `grp_eq_sym`) 0
    THENM RWH dset_of_monC 0 ...a)

⊢ (Σx ∈ dom(ps).
     Σx1 ∈ dom(qs).
      Σy1 ∈ dom(rs).
       Σy ∈ dom(qs) × dom(rs).
        when y =b  (x1 * y1).
          when (x * y) =b  u. ps[x] * (qs[x1] * rs[y1]))
   = ...

BY (RWO "rng_fset_for_when_eq" 0 ...a)

  7. x: |(g↓set)|
  8. ↑(x ∈b  dom(ps))
  9. x1: |(g↓set)|
  10. ↑(x1 ∈b  dom(qs))
  11. y1: |(g↓set)|
  12. ↑(y1 ∈b  dom(rs))
  ⊢ ↑(x1 * y1 ∈b  dom(qs) × dom(rs))

  BY (BLemma `prod_in_mset_prod` ...)

  ⊢ (Σx ∈ dom(ps).
       Σx1 ∈ dom(qs).
        Σy1 ∈ dom(rs).
         when (x * (x1 * y1)) =b  u.
           ps[x] * (qs[x1] * rs[y1]))
     = ...
```

Figure 10.11: Cancelling the $\Sigma$ and `when` over `y`

```
(x1 * y1) =b g y
```

to

```
(x1 * y1) =b (g↓set) y
```

The final step part of the proof, now with the work on the right-hand side included, is shown in Figure 10.12.

```
⊢ (Σx ∈ dom(ps).
     Σx1 ∈ dom(qs).
      Σy1 ∈ dom(rs).
       when (x * (x1 * y1)) =b  u. ps[x] * (qs[x1] * rs[y1]))
   = (Σy ∈ dom(rs).
       Σx1 ∈ dom(ps).
        Σy1 ∈ dom(qs).
         when ((x1 * y1) * y) =b  u.
           (ps[x1] * qs[y1]) * rs[y])

BY (RWN 3 (HereDnC (LemmaC ‘rng_mssum_swap‘)) 0
    THENM RW MonNormC 0
    THENM RW RngNormC 0 ...)
```

Figure 10.12: Making Both Sides Equal

### 10.7.2 Exploiting Algebraic Properties of Concrete Functions

Once it has been shown that a set of functions form some algebraic structure, it is then desirable to apply theorems about that algebraic structure to the set of functions. One way in Nuprl of doing this is to instantiate blocks of theorems about the algebraic structure with the instance in hand. Another is to temporarily rephrase the way in which the concrete functions are represented so that their algebraic structure is trivially recognized.

An example of this latter way is shown in Figure 10.13 which is the core of the proof of the theorem `oal_lt_trichot` from the theorem `oal_bpos_trichot`. Here the conversion `oal_grpC` rephrases the instances of functions that make up the `oal_grp` definition as projections from `oal_grp`. The group theorems

```
grp_inv_diff:
  ∀g:IGroup. ∀a,b:|g|.  ~ (a * (~ b)) = b * (~ a) ∈ |g|
```

```
1. s: LOSet
2. g: OGrp
3. ps: |oal(s;g)|
4. qs: |oal(s;g)|
⊢ ↑pos(qs ++ --ps) ∨ ps = qs ∈ |oal(s;g)| ∨ ↑pos(ps ++ --qs)

BY RWD oal_grpC 0

⊢ ↑pos(qs * (∼ ps))
   ∨ ps = qs ∈ |oal_grp(s;g)|
   ∨ ↑pos(ps * (∼ qs))

BY (RWN 2 (PolyC "grp_inv_diff<") 0
    THENM RWO "grp_eq_shift_right" 0 ...a)

⊢ ↑pos(qs * (∼ ps))
   ∨ qs * (∼ ps) = e ∈ |oal_grp(s;g)|
   ∨ ↑pos(∼ (qs * (∼ ps)))

BY RWD rem_oal_grpC 0

⊢ ↑pos(qs ++ --ps)
   ∨ 00 = qs ++ --ps ∈ |oal(s;g)|
   ∨ ↑pos(--(qs ++ --ps))
```

Figure 10.13: Viewing Concrete Functions Algebraically

```
grp_eq_shift_right:
  ∀g:IGroup. ∀a,b:|g|.  a = b ∈ |g| ⟺ e = b * (∼ a) ∈ |g|
```

are applied in the central tactic and finally `rem_oal_grpC` removes the abstract group. The `oal_grpC` and `rem_oal_grpC` conversions are simply defined using the `MacroC` direct-computation conversion.

```
let oal_grpC,rem_oal_grpC =
  let cprs =
        map (\t,t'. DoubleMacroC `oal_grpC` IdC t (ForceReduceC `5`) t')
  [⌈|oalist(s;g)|⌉,⌈|oal_grp(s;g)|⌉
  ;⌈ps ++ qs⌉,⌈ps * qs⌉
  ;⌈-- ps⌉,⌈∼ ps⌉
  ;⌈nil{s;g}⌉,⌈e⌉
  ]
  in
    FirstC (map fst cprs),FirstC (map snd cprs)
;;
```

### 10.7.3 Monotonicity Reasoning

A good example of monotonicity reasoning is shown in Figure 10.14. This proof step is from the proof of theorem `oal_times_dom`. The lemma being explicitly invoked is:

```
omral_plus_dom:
  ∀g:OCMon. ∀r:CRng. ∀ps,qs:|omral(g;r)|.
    ↑(dom(ps ++ qs) ⊆_b  dom(ps) ∪ dom(qs))
```

and the monotonicity lemmas that are automatically applied by the rewrite package include:

```
mset_mem_functionality_wrt_bsubmset:
  ∀s:DSet. ∀a:FSet{s}. ∀b:MSet{s}. ∀u:|s|.
    ↑(a ⊆_b  b) ⇒ ↑((u ∈_b  a) ⇒_b  (u ∈_b  b))

assert_functionality_wrt_bimplies:
  ∀u,v:𝔹.  ↑(u ⇒_b  v) ⇒ {↑u ⇒ ↑v}
```

```
↓
1. g: OCMon
2. r: CRng
3. qs: |omral(g;r)|
4. x: |(g↓set)|
5. ps: |omral(g;r)|
6. ...
7. x1: |g|
8. y: |r|
9. ...
10. ...
11. ↑(x ∈b  dom(<x1,y>* qs ++ (ps ** qs)))
⊢ ...
↓
BY (RWH (LemmaC ‘omral_plus_dom‘) 11 ...a)
↓
11. ↑(x ∈b  dom(<x1,y>* qs) ∪ dom(ps ** qs))
↓
```

Figure 10.14: Monotonic Reasoning

# Chapter 11

# Conclusions

## 11.1 Summary of Achievements

This thesis has presented work in building proof tools and developing formal theories that has transformed the Nuprl system into a substantially richer and more ergonomic environment for the production of formal mathematics. The case study on polynomial arithmetic in Chapter 10 demonstrates this success of this transformation.

More specifically, that case study demonstrates Nuprl's new capabilities for reasoning with abstract algebraic operations and for reasoning about concrete computations. Further, it shows how the capabilities can be intimately intertwined and I think it stands as a paradigm for the ADT (abstract data type) approach to program specification and implementation.

## 11.2 Future Directions

There are several directions that I think are worthy of further exploration.

- I myself am very keen to apply this work to formal reasoning about digital hardware and software systems where significant algebraic dexterity is required. For example, digital-signal-processing, fast-fourier-transform, and hybrid control systems. I see these applications as providing focussed and worthwhile challenges for theorem proving technology. This is an area of interest that I pursued earlier in my thesis research [Jac91, Jac92] and that provided the initial motivation for much of the work described in this thesis.

- One of the most significant problems with Nuprl is that of performance. The main cause of this problem is the highly redundant checking of well-formedness by proof. Proof-caching schemes have been experimented with in Nuprl a little, and they have helped, but no-one has got close yet to the optimal performance for such a scheme. By optimal, I am thinking here of when the total of the sizes of all the terms type-checked in a proof is a small constant factor times the sum of the initial size of the proposition being proven, and the new term fragments that are added by proof steps. I'm sure too that the well-formedness problem is masking other performance problems. Performance improvement of is critical if users are going to be able push through larger examples than those discussed here without having their patience exhausted.

- I think promising work could be done in further exploring the use of Nuprl to support program development using the Abstract Data Type paradigm. Better mechanisms for inheritance of structure between class definitions and subtyping between classes are definitely needed.

  As indicated in Section 5.4, there is still some room for progress to be made within Nuprl's current type theory, and Jason Hickey, a graduate student at Cornell, has started on work looking at helpful extensions to the type theory to better support inheritance and subtyping [Hic94].

  It would be interesting to see if this work could be tied in to current research in type systems for object-oriented programming [GM94], where similar issues crop up.

- The work described in this thesis has many connections with that going on in the implementation and use of computer algebra systems.

  - As shown in this thesis, we are beginning to be able to formally verify the correctness of the core code of a computer algebra system.
  - Rich kinds of algebraic manipulations that are awkward to carry out in current computer algebra systems, are becoming straightforward in theorem proving systems. For example, consider the rewrite rules demonstrated in Section 10.7 for rearranging sums. These rules have side conditions requiring non-trivial proof to solve.
  - Since the algebraic vocabulary of Nuprl is becoming more significant and more similar to that found in some computer algebra systems, possibilities are opening up of a theorem-proving environment providing useful logical-inference services to a computer algebra system.
  - A computer algebra system could be used to enhance a theorem proving environment by having the computer algebra system carry out complicated potentially-unsound computations that the prover then verifies. As mentioned in Section 1.5.2, Harrison and Thèry experimented with this idea in HOL and Maple.

- The type systems adopted by theorem-proving systems such as Nuprl and computer algebra systems such as Axiom [JS92] or Weyl [Zip93b] have a lot of similarities. Could a theorem-prover and a computer algebra system share a system of types? This clearly would simplify interfacing the two kinds of systems, since then they would be working in the same mathematical language.

In the short term, I am sure that some success is to be had in encouraging the linking of existing theorem provers and computer algebra systems. Here at Cornell, Zippel and I am currently engaged in investigating possibilities for interactions between Nuprl and Weyl [Jac94a].

In the long run, I see both technologies as developing so as to become more compatible with one another. However, it may be unwise to try to construct some monolithic hybrid system. Rather, it is probably best to create an environment in which these tools can interact in ways that are both rich and varied, yet also completely formal. This topic is currently being pursued by the Logic Group at Stanford [GK94], and there is considerable interest in it at Cornell.

- As shown in this thesis, the Nuprl system can produce quite readable proofs. I think with a modest further increase in automation, possibly just achieved with the existing tactics that I have developed, it should be possible to produce both readable and completely formal expositions of small bodies of mathematics or computer science; for example, of a chapter or two of a book on number theory [HW78], 'concrete mathematics' [GKP89] or functional programming [AS85, Pau91].

  An interface between the World-Wide-Web [BCL+94] and Nuprl is currently undergoing test. It would be very exciting if say an undergraduate in mathematics or computer science could access such formal expositions over the Web and follow them if they had some familiarity with the topic, but little or no prior preparation in languages or systems for formal proof.

## 11.3 Dependence on Nuprl's Type Theory

Nearly all the ideas behind the proof tools described in this thesis do not depend in any way on the fact that Nuprl's type theory is constructive, or indeed, on that Nuprl uses a type theory at all. All the ideas in the rewrite package, the relational-reasoning tactic, or the arithmetic tactic, could just as easily be made to work in any interactive theorem prover. Of course, integrating decision procedures into theorem provers where proofs are mostly generated automatically is a much different (and often much harder) problem, as is shown in Boyer and Moore's work in NQTHM [BM88b].

Much of the work in abstract algebra did make heavy use of Nuprl's dependent function and dependent product types ($\boldsymbol{\Pi}$ and $\boldsymbol{\Sigma}$ types), so I think it would be difficult to duplicate this work in say the present HOL system [GM93], which has a much simpler type theory.

On the other hand, in a classical set-theoretic framework such as Mizar's, the concepts of a dependent function and a dependent product are defined rather than primitive notions (though the type system in Mizar does have these concepts built into it for convenience). Set-theoretic frameworks are known to be adequate for virtually all of mathematics, so one rarely runs into problems in these frameworks of not knowing at all how to formalize some concept. Of course, there might be much debate about *which* method of formalization is best.

When I was in the mode of creating and proving properties about functional programs, I did find it very elegant that Nuprl's constructive type theory forced me to make the functions computable; if I was creating a sorting function, I was forced into supplying the sorting function with a computable order function as an argument, as I would have had to in any other functional programming language. If one is defining such functions in a classical type theory such as HOL's, such a discipline is only enforced by the user and not by the system. Indeed, when it comes to passing functions for computing equality to other functions, a user of HOL might find it convenient to ignore the fact that 'real' functions in a programming language require such an argument. Note that the insistence on functions being computable is not just a practice found in constructive type theories. It also occurs in simpler logics such as the 'computational logic' of NQTHM [BM88a].

Other times though, I found the computability restrictions of Nuprl petty. For example, the split between the type of propositions and the type of booleans lead to — from a classical point of view — much unaesthetic duplication of definitions and theorems. Also in the work on permutation functions in Section 7.2, constructivity considerations forced me to make many definitions and prove many theorems twice that only would have to be done once in a classical system. I felt that formal mathematics is tedious enough as it is, without all these extra distinctions thrown in.

One solution to this dilemma involves building a theory of the semantics of a programming language in some classical framework. By default, functions are not necessarily computable. Then, when one wants to establish the computability of some function, one has to explicitly give an expression in the programming language that has that function as its denotation in the semantics. The advantage of this approach is that it smoothly opens the door to reasoning about the computational complexity of functions. Theories of programming languages have been developed in several theorem provers such as HOL and NQTHM.

Constructive type theory offers no features for proving theorems about complexity, unless one considers adding in some reflection mechanism. A fair amount of work has been done in this direction in Nuprl [Kno87, How88a, ACHA90, CH90]. However, reflection in a constructive type theory requires constructing a theory of not only a functional programming language, but also a theory of sequents, rules and proofs; this is a much more challenging task, especially in a type theory as structurally complicated at Nuprl's.

Another promising solution to this dilemma involves trying to combine set theory with a functional computation language [HS94]. Functions in this language are computable just when they are constructed in certain well-defined ways.

## 11.4 Appropriateness of Nuprl's Type Theory

It is elegant to be working in a foundational theory where the objects you are often reasoning about are computable functions that you can actually execute. However, I think that Nuprl's present type theory is an inadequate foundation for abstract algebra, be it classical or constructive; the discussion in Chapter 5 explains the difficulties with adequatly formulating such basic notions as the collection of all subtypes of a type.

Part of the problems with Nuprl's type theory are certainly not intrinsic to the idea of a constructive type theory; things would have been less awkward if the standard equality on types had been extensional rather than intensional, if equality-respecting hadn't been so wrapped up in the semantics of Nuprl sequents and if the type theory had included a subtyping predicate. Fortunately, I think that much of the theory development and all the proof tool development would remain intact if a type theory were adopted with these or equivalent changes. Howe has put much thought into how such changes might be achieved [How93, HS94].

However, part of the problems are inherent in the constructivist agenda of seeing potential computational content in every logical proposition. The constructivist makes many subtle distinctions that a classical mathematician ignores. Then, to make the presentation of the mathematics readable, the constructive mathematician hides a lot of these distinctions so they are implicit rather than explicit in the notation. I didn't know how in Nuprl's type theory to systematically formalize this hiding of detail for constructive algebra, but I did get the definite sense that mechanical formalizations of constructive type theories can play an important role in helping constructive mathematicians make their ideas precise.

The approach I adopted to constructive algebra was predominantly an explicit one; that is, I assumed that constructions would always be made explicit and not be assumed implicit in the computational content of propositions. This approach was close in spirit to that taken in Scratchpad and Axiom [JS92, DT92, DGT92]. If inhabitants of algebraic classes were to have a decidable equality, then a function would be explicitly required to compute that equality in the signature. If in a group it was desired that that every element have a computable inverse, then an inverse function had to explicitly appear in the group signature.

In a way, I was taking advantage of how Nuprl's type theory limited me. The fact that Nuprl the boolean type and the type of propositions in are distinct, forced me into giving an explicit treatment of equality in the algebraic classes I set up, and forced me into developing the beginnings of a theory of discrete types (I called them discrete sets or `DSet`'s, see Section 6.3 and Section 10.4.1).

Note that in Chapter 8, I did explore taking a näive view of a completely implicit constructive approach; I was careful never to ignore the computational content of any proposition. The results were that I had one theorem that from a constructive point-of-view, specified a program for computing the rich computational content of an equivalence relation '$a$ is equal to $b$ up to permutations and associates', where $a$ and $b$ were lists of elements of a cancellation monoid(see Section 8.1.3). It was questionable whether anyone would care about this computational content. However, another theorem I proved specified a program for computing factorizations in cancellation monoids, a more useful computation to want.

# Appendix A

# Divisibility Theory

This appendix reproduces in full the proofs abbreviated in Figure 8.1 and Figure 8.2 of Chapter 8. The numbers interspersed in the vertical bars of the proof branches serve to help trace branches when proof printouts such as these run over several pages.

## A.1 Existence Theorem

```
⊢ ∀g:IAbMonoid
│     Cancel(|g|;|g|;*)
│     ⇒ WellFnd(|g|;x,y.x p| y)
│     ⇒ (∀c:|g|. Dec(Reducible(c)))
│     ⇒ (∀b:|g|. ¬(g-unit(b)) ⇒ (∃as:Atom{g} List. b = Π as))
↓
BY (UnivCD ...a)
↓
1. g: IAbMonoid
2. Cancel(|g|;|g|;*)
3. WellFnd(|g|;x,y.x p| y)
4. ∀c:|g|. Dec(Reducible(c))
5. b: |g|
6. ¬(g-unit(b))
⊢ ∃as:Atom{g} List. b = Π as
↓
BY (WFndHypInd 3 5 THENM D 0 ...a)
↓
5. j: |g|
6. ∀k:|g|. k p| j ⇒ ¬(g-unit(k)) ⇒ (∃as:Atom{g} List. k = Π as)
7. ¬(g-unit(j))
⊢ ∃as:Atom{g} List. j = Π as
↓
BY (Decide ⌈Reducible(j)⌉ ...a)
│
├→8. Reducible(j)
│
```

```
1 BY UnfoldTopAb 8 THEN ExistHD 8

  8. b: |g|
  9. c: |g|
  10. ¬(g-unit(b))
  11. ¬(g-unit(c))
  12. j = b * c

1 BY (SwapEquands 12
      THEN FLemma `non_munit_diff_imp_mpdivides` [12] ...a)

  12. b * c = j
  13. b p| j

1 BY (RWH (LemmaC `abmonoid_comm`) 12
      THENM FLemma `non_munit_diff_imp_mpdivides` [12] ...a)

  12. c * b = j
  14. c p| j

1 BY (FHyp 6 [13] THENM FHyp 6 [14] ...a)

  15. ∃as:Atom{g} List. b = Π as
  16. ∃as:Atom{g} List. c = Π as

1 BY New [`as2'] (D 16) THEN New [`as1'] (D 15)

  15. as1: Atom{g} List
  16. b = Π as1
  17. as2: Atom{g} List
  18. c = Π as2

1 BY (With ⌜as1 @ as2⌝ (D 0)
       THENM RewriteWith [] ``mon_reduce_append`` 0 ...a)

  ⊢ j = Π as1 * Π as2

1 BY (RWH (RevHypC 18 ORELSEC RevHypC 16) 0
       THENM RW AbMonNormC 12 ...)

  8. ¬Reducible(j)

  BY (With ⌜j::[]⌝ (D 0) THENM AbReduce 0 ...a)

    ⊢ j ∈ Atom{g}

  1 BY (MemTypeCD ...)

    ⊢ Atomic(j)
```

```
1 BY (Unfold ‘matomic‘ 0 ...)

  ⊢ j = j * e

  BY (RW MonNormC 0 ...)
```

## A.2 Uniqueness Theorem

```
⊢ ∀g:IAbMonoid
     Cancel(|g|;|g|;*)
     ⇒ (∀a,b:|g|.  Dec(a | b))
     ⇒ (∀ps,qs:Prime(g) List.  Π ps ∼ Π qs ⇒ ps ≡ qs upto ∼)

BY (RepeatMFor 4 (D 0) ...a)

1. g: IAbMonoid
2. Cancel(|g|;|g|;*)
3. ∀a,b:|g|.  Dec(a | b)
4. ps: Prime(g) List
⊢ ∀qs:Prime(g) List. Π ps ∼ Π qs ⇒ ps ≡ qs upto ∼

BY (New [‘p’;‘ps\’’] (ListInd 4)
    THEN OnAll AbReduce ...)

  5. qs: Prime(g) List
  6. e ∼ Π qs
  ⊢ [] ≡ qs upto ∼

1 BY D 6 THEN Thin 6  THEN FoldTop ‘munit‘ 6

  6. g-unit(Π qs)

1 BY MoveToConcl 6
     THEN New [‘q’;‘qs\’’] (D 5)
     THEN (D 0 ...a)
     THEN OnAll AbReduce

    6. g-unit(e)
    ⊢ [] ≡ [] upto ∼

1 2 BY (StrengthenRel ...)

    6. q: Prime(g)
    7. qs’: Prime(g) List
    8. g-unit(q * Π qs’)
    ⊢ [] ≡ q::qs’ upto ∼

1   BY Assert ⌈False⌉ THENM Trivial
       THEN D 6 THEN D 7

    6. q: |g|
    7. ¬(g-unit(q))
    8. ∀b,c:|g|.  q | b * c ⇒ q | b ∨ q | c
    9. qs’: Prime(g) List
    10. g-unit(q * Π qs’)
```

```
|   ⊢ False
|   ↓
1   BY (FLemma ‘munit_of_op‘ [10] ...)
|
└→5. p: Prime(g)
  6. ps’: Prime(g) List
  7. ∀qs:Prime(g) List. Π ps’ ∼ Π qs ⇒ ps’ ≡ qs upto ∼
  8. qs: Prime(g) List
  9. p * Π ps’ ∼ Π qs
  ⊢ p::ps’ ≡ qs upto ∼
  ↓
  BY Assert ⌈p | Π qs⌉
  |
  └→⊢ p | Π qs
  |  ↓
  1 BY OnCls [9;9] D
  |  |
  |  9. c: |g|
  |  10. Π qs = (p * Π ps’) * c
  |  11. Π qs | p * Π ps’
  |  ↓
  1 BY (With ⌈Π ps’ * c⌉ (D 0)
  |      THENM RW MonNormC 10 ...)
  |
  └→10. p | Π qs
     ↓
     BY (FLemma ‘mprime_divs_list_el‘ [-1] ...a)
     |  THENM (Thin (-2) THEN D (-1))
     |
     └→⊢ IsPrime(p)
     |  ↓
     1 BY (D 5 THEN NoteConclSqStable ...)
     |
     └→10. i: ℕ||qs||
        11. p | qs[i]
        ↓
        BY Assert ⌈p ∼ qs[i]⌉
        |  THENM Thin 11
        |
        └→ ⊢ p ∼ qs[i]
        |  ↓
        1 BY (Backchain ‘‘mdivisor_of_atom_is_assoc
        |  |      mprime_imp_matomic‘‘ ...)
        |  |
        |  └→⊢ ¬(g-unit(p))
        |  |  ↓
        1 2 BY (D 5 THEN Unhide THENM D 6 ...)
        |  |
        |  └→⊢ IsPrime(qs[i])
        ↓     ↓
```

```
1   BY (Assert ⌈qs[i] ∈ Prime(g)⌉ THENM MemTypeHD (-1) ...a
       )

    12. qs[i] = qs[i]
    [13]. IsPrime(qs[i])

1   BY (NoteConclSqStable ...)

  11. p ∼ qs[i]

  BY MoveToEnd 9
     THEN (OnMCls [0;-1] (RWH
             (IfIsC ⌈qs⌉ (RevLemmaWithC [`i',⌈i⌉] `select_r
     eject_permr`))) ...a)
     THEN AbReduce (-1)

  9. i: ℕ||qs||
  10. p ∼ qs[i]
  11. p * Π ps' ∼ qs[i] * Π qs\[i]
  ⊢ p::ps' ≡ qs[i]::qs\[i] upto ∼

  BY (SeqOnM
       [RWH (HypC 10) 11;FLemma `massoc_cancel` [11];Thin (
     -2)
       ;RelArgCD] ...)

  11. Π ps' ∼ Π qs\[i]
  ⊢ ps' ≡ qs\[i] upto ∼

  BY (BHyp 7 ...)
```

# Bibliography


[ACHA90] Stuart F. Allen, Robert L. Constable, Douglas J. Howe, and William B. Aitken. The semantics of reflected proof. In *Proceedings of the Fifth Annual IEEE Symposium on Logic and Computer Science*, pages 95–107. IEEE Computer Society, June 1990.

[ACN90] L. Augustsson, T. Coquand, and B. Nordstrom. A short description of another logical framework. In *Proc. of First Annual Workshop on Logical Framworks*, pages 38–41, Sophia-Antipolis, France, 1990.

[Acz93] Peter Aczel. Galois: a theory development project. manuscript, University of Manchester, 1993.

[AL92] Mark Aagaard and Miriam Leeser. Verifying a logic synthesis tool in Nuprl: A case study in software verification. In Gregor Bochmann and David Probst, editors, *Fourth International Workshop on Computer-Aided Verification*, volume 663 of *Lecture Notes in Computer Science*, pages 69–81. Springer-Verlag, June 1992.

[All87a] Stuart F. Allen. A non-type theoretic definition of Martin-Löf's types. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 215–221. IEEE, 1987.

[All87b] Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. Ph.D. dissertation, Cornell University, Ithaca, NY, 1987. TR 87-866.

[All94] Stuart F. Allen. formalization of a semantics for imperative programming languages. To appear in a Cornell University, Department of Computer Science Technical Report, 1994.

[AP90] James A. Altucher and Prakash Panangaden. A mechanically assisted constructive proof in category theory. In *Proceedings of the 10th International Conference on Automated Deduction*. Springer-Verlag, July 1990. (LNCS).

[AS85] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, 1985.

[Bai93] Anthony Bailey. Representing algebra in LEGO. Masters dissertation, University of Edinburgh, November 1993.

[Bar93] Gilles Barthe. Formalizing galois theory. manuscript, University of Manchester, March 1993.

[Bar94] Gilles Barthe. Mathematical concepts in type theory. manuscript, University of Nijmegen, March 1994.

[Bas89] David Basin. *Building Problem Solving Environments in Constructive Type Theory*. Ph.D. dissertation, Cornell University, Ithaca, NY, 1989.

[Bat79] Joseph L. Bates. *A Logic For Correct Program Development*. Ph.D. dissertation, Cornell University, 1979.

[BB85] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Springer-Verlag, 1985.

[BC85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Prog. Lang. Sys.*, 7(1):113–136, 1985.

[BC93] David A. Basin and Robert L. Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*. Cambridge University Press, 1993.

[BCL+94] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

[BD89] David A. Basin and Peter Delvecchio. Verification of combinational logic in Nuprl. In *Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Ithaca, New York, 1989.

[BG94] Leo Bachmair and Harald Ganzinger. Ordered chaining for total orderings. In A. Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artifical Intelligence, pages 435–450. Springer Verlag, June 1994.

[Bis67] Errett Bishop. *Foundations of Constructive Analysis*. McGraw-Hill, New York, 1967.

[Bis70] Errett Bishop. Mathematics as a Numerical Language. In *Intuitionism and Proof Theory*, pages 53–71. North-Holland, NY, 1970.

[Ble75] W. W. Bledsoe. A new method for proving certain Presburger formulas. In *4th International Joint Conference on Artificial Intelligence*, pages 15–21, Tiblsi, 1975.

[BM79] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. ACM Monograph Series. Academic Press, New York, 1979.

[BM84] Robert Boyer and J. Strother Moore. Proof checking the RSA public key encription algorithm. *American Mathematical Monthly*, 91(3):181–189, 1984.

[BM88a] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, New York, 1988.

[BM88b] Robert S. Boyer and J. Strother Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. In *Machine Intelligence 11*. Oxford Universtiy Press, 1988.

[Bou74] Nicolas Bourbaki. *Algebra, Part I.* Elements of Mathematics. Addison-Wesley, 1974.

[BvHH+89] Alan Bundy, Frank van Harmelen, Jane Hesketh, Alan Smaill, and Andrew Stevens. A rational reconstruction and extension of recursion analysis. In *International Joint Conference on Artificial Intelligence*, pages 359–365, Detroit, Michigan, 1989.

[BvHSI90] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M.E. Stickel, editor, *10th International Conference on Automated Design,* Lecture Notes in Artificial Intelligence, Vol. 449, pages 132–146. Springer-Verlag, 1990.

[BWi93] Thomas Becker, Volker Weispfenning, and in cooperation with Heinz Kredel. *Gröbner Bases: a computational approach to commutative algebra*. Graduate Texts in Mathematics. Springer-Verlag, 1993.

[C+86] Robert Constable et al. *Implementing Mathematics with The Nuprl Development System*. Prentice-Hall, NJ, 1986.

[CB83] Robert L. Constable and Joseph L. Bates. The nearly ultimate pearl. Technical Report TR 83-551, Cornell University, Ithaca, NY, January 1983.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

[CGG+91] Bruce W. Char, Keith O. Geddes, Gaston H. Gonnet, Benton L. Leong, Michael B. Monagan, and Stephen M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, 1991.

[CH85] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In B. Buchberger, editor, *EUROCAL '85: European Conference on Computer Algebra*, pages 151–184. Springer-Verlag, 1985.

[CH88] Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

[CH90] Robert L. Constable and Douglas J. Howe. Implementing metamathematics as an approach to automatic theorem proving. In R.B. Banerji, editor, *Formal Techniques in Artificial Intelligence: A Source Book*, pages 45–76. Elsevier Science Publishers (North-Holland), 1990.

[Che92] W.Z. Chen. Tactic-based theorem proving and knowledge-based forward chaining. In D. Kapur, editor, *Eleventh International Conference on Automated Deduction,* Lecture Notes in A.I., Vol. 607, pages 552–566. Springer-Verlag, June 1992.

[Chu40] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:55–68, 1940.

[Chv83] Vašek Chvátal. *Linear Programming.* Freeman, 1983.

[CJE82] Robert L. Constable, Scott D. Johnson, and Carl D. Eichenlaub. *Introduction to the PL/CV2 Programming Logic*, volume 135 of *Lecture Notes in Computer Science.* Springer-Verlag, New York, 1982.

[CLO92] David Cox, John Little, and Donal O'Shea. *Ideals, Varieties, and algorithms: an introduction to computational algebraic geometry and commutative algebra.* Undergraduate Texts in Mathematics. Springer-Verlag, 1992.

[CM85] Robert L. Constable and Nax P. Mendler. Recursive definitions in type theory. In Rohit Parikh, editor, *Logics of Programs*, pages 61–78, Berlin, 1985. Springer-Verlag.

[Con71] Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233, Ljubljana, 1971.

[Con85a] Robert L. Constable. Constructive mathematics as a programming logic I: some principles of theory. In *Annals of Mathematics, Vol. 24*. Elsevier Science Publishers, B.V. (North-Holland), 1985. Reprinted from *Topics in the Theory of Computation*, Selected Papers of the Intl. Conf. on "Foundations of Computation Theory," FCT '83.

[Con85b] Robert L. Constable. The semantics of evidence. Technical Report TR 85–684, Cornell University, Department of Computer Science, Ithaca, New York, May 1985.

[CS87] Robert L. Constable and Scott F. Smith. Partial objects in constructive type theory. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*. IEEE, 1987.

[CZ92] Edmund Clarke and Xudong Zhao. Analytica - a theorem prover in Mathematica. In D. Kapur, editor, *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artifical Intelligence*, pages 761–765. Springer-Verlag, 1992.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.

[DFH+91] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, and B. Werner. *The System Coq, V5.6, User's Guide*. INRIA, September 1991. To appear as an INRIA technical report.

[DGT92] J. H. Davenport, P. Gianni, and B. M. Trager. Scratchpad's view of algebra II: A categorical view of factorization. *Numerical Algorithms Group*, 1992.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 6. Elsevier, 1990.

[DST93] J.H. Davenport, Y. Siret, and E. Tournier. *Computer Algebra — Systems and Algorithms for Algebraic Computation*. Academic Press, London, second edition, 1993.

[DT92] J. H. Davenport and B. M. Trager. Scratchpad's view of algebra I: Basic commutative algebra. *Numerical Algorithms Group*, 1992.

[Dum77] Michael Dummett. *Elements of Intuitionism*. Oxford Logic Series. Clarendon Press, Oxford, 1977.

[Edw89] H. M. Edwards. *Kronecker's views on the foundations of Mathematics.* Courant Institute of Mathematical Sciences, New York University, New York, 1989.

[FGT92a] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. IMPS: System description. In D. Kapur, editor, *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artifical Intelligence*, pages 701–705. Springer-Verlag, 1992.

[FGT92b] William M. Farmer, Joshua D. Guttman, and F. Javier Thayer. Little theories. In D. Kapur, editor, *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artifical Intelligence*, pages 567–581. Springer-Verlag, 1992.

[For93] Max B. Forester. Formalizing constructive real analysis. Technical Report TR93-1382, Computer Science Dept., Cornell University, Ithaca, NY, 1993.

[FS55] A. Frölich and J.E. Shepherdson. Effective procedures in field theory. *Philosophical Transactions of the Royal Society, Series A*, 284(950):407–432, 1955.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[Gir71] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Second Scandinavian Logic Symposium*, pages 63–92, Amsterdam, 1971. North-Holland.

[GK94] Michael R. Genesereth and Steven P. Ketchpel. Software agents. *Communications of the ACM*, 37(7):48–53,147, July 1994.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science.* Addison-Wesley, 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.

[GM94] Carl A. Gunter and John C. Mitchell, editors. *Theoretical aspects of object-oriented programming: types, semantics, and language design.* Foundations of Computing. MIT Press, 1994.

[GMW79] Michael J. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[Gor94] M.J.C. Gordon. Merging hol with set theory: preliminary experiments. Technical Report 353, University of Cambridge Computer Laboratory, 1994.

[Gro91] The HOL Group. HOL tutorial manual. Available on World-Wide Web at `http://lal.cs.byu.edu/lal/holdoc/tutorial.html`, 1991.

[GSHH91] Joseph Goguen, Andrew Stevens, Hendrik Hilberdink, and Keith Hobley. 2OBJ: A metalogical framework based on equational logic. manuscript, Programming Research Group, Oxford University, 1991.

[Gun89] Elsa L. Gunter. Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Department of computer and Information Science, University of Pennsylvania, 1989.

[Har92] John Harrison. The HOL reals library. Available on World-Wide Web at `http://lal.cs.byu.edu/lal/holdoc/library/reals.dvi`, July 1992.

[HB70] D. Hilbert and P. Bernays. *Grundlagen der Mathematik*, volume II. Springer-Verlag, second edition, 1970.

[HB92] Warren A Hunt, Jr. and Bishop C. Brock. A formal HDL and its use in the FM9001 verification. *Philosophical Transactions of the Royal Society, Series A*, 339(1652):35–47, 15 April 1992.

[Hey66] A. Heyting. *Intuitionism, An Introduction*. North-Holland, Amsterdam, 1966.

[Hic94] Jason Hickey. Highly dependent function types. Nuprl internal document, September 1994.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11:31–55, 1978.

[How87] Douglas J. Howe. Implementing number theory: An experiment with Nuprl. In *Eighth Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 404–415. Springer-Verlag, July 1987.

[How88a] Douglas J. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* Ph.D. dissertation, Cornell University, Ithaca, NY, April 1988.

[How88b] Douglas J. Howe. Computational metatheory in Nuprl. *CADE-9*, pages 238–257, May 1988.

[How89] Douglas J. Howe. Equality in lazy computation systems. *Proc. Fourth Symp. Logic in Computer Science, IEEE*, pages 198–203, June 1989.

[How91a] Douglas J. Howe. On computational open-endedness in Martin-Löf's type theory. In *Proceedings of Sixth Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.

[How91b] Douglas J. Howe. Universe polymorphism. Internal Nuprl V4 Document, 1991.

[How93] Douglas J. Howe. Reasoning about functional programs in Nuprl. *Functional Programming, Concurrency, Simulation and Automated Reasoning,* Lecture Notes in Computer Science, 1993.

[HS94] Douglas J. Howe and Scott D. Stoller. An operational approach to combining classical set theory and functional programming languages. In G. Goos and J. Hartmanis, editors, *Lecture Notes in Computer Science*, pages 36–55, New York, April 1994. International Symposium TACS '94, Springer-Verlag. Theoretical Aspects of Computer Software.

[HT93] John Harrison and Laurent Thèry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In *Proceedings of the HOL '93 Workshop on Higher Order Logic Theorem Proving and its Applications*, 1993.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, Massachusetts, 1979.

[Hue75] Gerard P. Huet. A unification algorithm for the typed lambda-calculus. *Theoretical Computer Science*, 1(1):27–58, 1975.

[HW78] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers.* Clarendon, fifth edition, 1978.

[Jac74] Nathan Jacobson. *Basic Algebra*, volume I. Freeman, 1974.

[Jac91] Paul B. Jackson. Developing a toolkit for floating-point hardware in the nuprl proof development system. In *Proceedings of the Advanced Research Workshop on Correct Hardware Design Methodologies.* Elsevier, 1991.

[Jac92] Paul B. Jackson. Nuprl and its use in circuit design. In R.T. Boute V. Stavridou, T.F.Melham, editor, *Proceedings of the 1992 International Conference on Theorem Provers in Circuit Design*, IFIP Transactions A-10. North-Holland, 1992.

[Jac94a] Paul B. Jackson. Exploring abstract algebra in constructive type theory. In A. Bundy, editor, *12th Conference on Automated Deduction*, Lecture Notes in Artifical Intelligence. Springer, June 1994.

[Jac94b] Paul B. Jackson. Nuprl V4.1 users guide and reference manual. Technical report, Cornell University, Department of Computer Science, 1994.

[Jac94c] Paul B. Jackson. Solving universe-level constraints in Nuprl. To appear in a Nuprl internal document, December 1994.

[JS92] Richard D. Jenks and Robert S. Sutor. *AXIOM: the Scientific Computation System.* Springer-Verlag, 1992.

[Jut77] L. S. Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH system.* Ph.D. dissertation, Eindhoven University, 1977.

[Kno87] Todd B. Knoblock. *Metamathematical Extensibility in Type Theory.* Ph.D. dissertation, Cornell University, 1987.

[LA93] Jordi Levy and Jaume Agusti. Bi-rewriting, a term rewriting technique for monotonic order relations. In Claude Kirchner, editor, *5th International Conference on Rewriting Techniques and Applications*, volume 690 of *LNCS*, pages 17–31. Springer-Verlag, 1993.

[Lan84] Serge Lang. *Algebra.* Addison-Wesley, 2nd edition, 1984.

[Lee92] Miriam E. Leeser. Using Nuprl for the verification and synthesis of hardware. *Philosophical Transactions of the Royal Society, Series A*, 339(1652):49–68, 15 April 1992.

[Luo89] Z. Luo. ECC, an extended calculus of construction. In *Proc. of Fourth Symp. on Logic in Comp. Sci.*, pages 385–395, Pacific Grove, CA, June 1989.

[Mat94] R. Matuszewski. *Formalized Mathematics: A Computer Assisted Approach*, volume 1-4. Université Catholique de Louvain - Fondation Philippe le Hodey, Brussels, 1990-94.

[McA89] David A. McAllester. *ONTIC: A Knowledge Representation System for Mathematics.* MIT Press, Cambridge, 1989.

[Men88] P.F. Mendler. *Inductive Definition in Type Theory.* Ph.D. dissertation, Cornell University, Ithaca, NY, 1988.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, 1991.

[Mis93] Bhubaneswar Mishra. *Algorithmic Algebra.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[ML82] Per Martin-Löf. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, pages 153–175, Amsterdam, 1982. North Holland.

[MN79] G. Metakides and A. Nerode. Effective content of field theory. *Annals of Mathematical Logic*, 17:289–320, 1979.

[MRR88] Ray Mines, Fred Richman, and Wim Ruitenburg. *A Course in constructive Algebra.* Universitext. Springer-Verlag, 1988.

[MTH91] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 1991.

[Mur90] Chet Murthy. *Extracting Constructive Content from Classical Proofs.* Ph.D. dissertation, Cornell University, 1990.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artifical Intelligence*, pages 748–752. Springer-Verlag, 1992.

[Pau83a] Lawrence C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[Pau83b] Lawrence C. Paulson. Tactics and tacticals in Cambridge LCF. Technical Report Report 39, University of Cambridge, Computer Lab., 1983.

[Pau85] Lawrence C. Paulson. Interactive theorem proving with cambridge LCF. Technical Report Report 80, University of Cambridge, Computer Lab., 1985.

[Pau87] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, Cambridge, 1987.

[Pau90] Lawrence C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–385. Academic Press, 1990.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[Pol90] R. Pollack. Lego user's guide. Technical report, University of Edinburgh, 1990.

[PPM89] F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Mathematical Foundations of Program Semantics, 5th International Conference,* Lecture Notes in Computer Science, Vol. 442, pages 209–228. Springer-Verlag, 1989.

[Pra71] Dag Prawitz. Ideas and results in proof theory. In *Studies in Logic and the Foundations of Mathematics*, pages 235–307. North-Holland, Amsterdam, 1971.

[Rud92] P. Rudnicki. An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs*, Bastad, 1992. Chalmers University of Technology.

[Rus08] Bertrand Russell. Mathematical logic as based on a theory of types. *Am. J. Math.*, 30:222–62, 1908.

[Sco70] Dana S. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275, Berlin, 1970. Spinger-Verlag.

[Sha86] Natarajan Shankar. *Proof Checking Metamathematics.* Ph.D. dissertation, University of Texas at Austin, 1986.

[Sho67] Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Massachusetts, 1967.

[Sho77] Robert Shostak. On the SUP-INF Method for Proving Presburger Formulas. *JACM*, 24(4):351–360, 1977.

[Smi89] Scott F. Smith. *Partial Objects in Type Theory.* Ph.D. dissertation, Cornell University, Ithaca, NY, 1989.

[Smu68] Raymond Smullyan. *First Order Logic.* Springer-Verlag, 1968.

[TvD88] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, An Introduction, Vol. I,II.* North-Holland, Amsterdam, 1988.

[Wir90] Martin Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, chapter 13. Elsevier, 1990.

[WOEB84] L. Wos, R. Overbeek, L. Ewing, and J. Boyle. *Automated Reasoning.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Wol91] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, second edition, 1991.

[WR27] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1, 2, 3. Cambridge University Press, 2nd edition, 1925-27.

[Zip93a] Richard E. Zippel. *Effective Polynomial Computation.* The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers, 1993.

[Zip93b] Richard E. Zippel. The Weyl computer algebra substrate. In Alfonso Miola, editor, *Design and Implementation of Symbolic Computation Systems*, volume 722 of *Lecture Notes in Computer Science*, pages 303–318. Springer Verlag, 1993.