Exploring Abstract Algebra in Constructive Type Theory *

Paul Jackson †
Department of Computer Science
Cornell University
Ithaca NY 14853
USA
Tel: [+1] 607-255-1372
E-mail: jackson@cs.cornell.edu

January 20, 1995

Abstract

I describe my implementation of computational abstract algebra in the Nuprl system. I focus on my development of multivariate polynomials. I show how I use Nuprl's expressive type theory to define classes of free abelian monoids and free monoid algebras. These classes are combined to create a class of all implementations of polynomials. I discuss the issues of subtyping and computational content that came up in designing the class definitions. I give examples of relevant theory developments, tactics and proofs. I consider how Nuprl could act as an algebraic 'oracle' for a computer algebra system and the relevance of this work for abstract functional programming.

1 Introduction

1.1 Aims and Motivation

One aim of the Nuprl project is to explore the use of constructive type theory for specifying, verifying and synthesizing functional programs. To date, effort has been concentrated on developing programs over concrete data-types. I would like to see how well Nuprl could support the development of programs using abstract data types (ADT's); ADT's are widely advocated as an effective structuring principle and support for them is provided in a variety of modern programming languages. Type theory can provide an effective framework for formally reasoning about ADT's. Type theory is particularly convenient in that both ADT specifications and implementations are described in a single language [MP85, BC93].

In previous work on using Nuprl to verify hardware designs for floating-point arithmetic, I noticed that algebraically-similar kinds of reasoning were coming up frequently over different concrete datatypes such as the integers, the rationals and

*To Appear in *Proceedings of the Twelfth International Conference on Automated Deduction*, July 1994
†Supported by NASA GSRP Fellowship NGT-50786 and ONR Grant N00014-92J-1764

bit-vectors. For example, a significant part of the reasoning concerned iterated sums over monoids. It was obvious that work could be saved by first making definitions and proving theorems in an appropriately abstract setting.

A newer motivation comes from observing the development of computer algebra systems such as Mathematica, Maple and Axiom. Such systems provide a wealth of functionality, but not the rigor or the abstractness of a theorem proving environment. I want to explore possibilities for symbiotic interactions between computer algebra systems and theorem provers, and speculate on how in the long term they might be integrated. Already promising work has been done in the area [CZ92, HT93]. Here at Cornell I am currently investigating links between Nuprl and the Weyl system [Zip93]. In particular I am exploring the use of Nuprl as an algebraic oracle to Weyl.

Nuprl's type theory provides a certain range of options for making explicit various computational aspects of algebra. I would like to understand which options are worth formalizing and which ones I am missing, both from the point of view of providing assistance to computer algebra systems and of developing abstract programs. For example, most computer algebra systems are interested in algebraic structures with decidable equality relations, so I am reasoning primarily with structures with boolean-valued equality functions.

1.2 Background

Nuprl [C$^+$86, Jac92] is an interactive tactic-based theorem prover in the LCF tradition. It uses a constructive type theory similar to that of Martin-Löf [ML82]. The type theory is briefly reviewed in Section 2.

Nuprl has been developed in the last 10 years by over a dozen people. It has a well developed user interface and a large collection of tactics for such tasks as forward and backward chaining, rewriting and arithmetic reasoning. Libraries have been built up in number theory [How87], analysis [CH92], hardware verification [BV90], hardware synthesis [AL90] and meta-reasoning [How88]. Many of the tactics and theory developments mentioned in this paper are covered in greater detail in my thesis [Jac94].

1.3 Organization of this Paper

To illustrate my work, I have chosen to show a sequence of definitions leading up to the class of multivariate polynomial algebras. Section 2 contains an introduction to Nuprl's type theory that should be adequate for understanding the definitions, and the definitions themselves are presented in Section 3. Section 4 gives some practical details on my work to date and Section 5 describes applications of the work that we are actively pursuing. Finally, Section 6 summarizes my accomplishments.

1.4 Related Work

I know of several other efforts to develop abstract algebra in a theorem proving environment. Gunter[Gun89] working with HOL has proven group isomorphism theorems and shown the integers mod n to be an implementation of abstract groups. Harrison and Thèry [HT93] have looked at the interaction between HOL and Maple, where Maple performs algebraic manipulations and integrations that HOL then verifies. The IMPS people have a notion of *little theories* [FGT92] which they use for proving theorems about groups and rings. Anthony Bailey has 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]. The Mizar project seems to have done a fair amount in algebra, but hasn't yet tackled polynomials.

Clarke and Xudong have been adding theorem proving capabilities to Mathematica 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.

The Larch [GH93] group has worked on program specification and verification using ADT's in a first-order-logic setting.

2 Type Theory Preliminaries

I give here an informal overview of the types in Nuprl's type theory I have been working with:

- The *booleans* $\mathbb{B}$ and the *integers* $\mathbb{Z}$.

- A *dependent-function* (Π) 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. Each type $A \rightarrow B$ is considered as containing only the computable functions from A to B rather than all set theoretic functions.

- A *dependent-product* (Σ) 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.

- A *set* type constructor $\{\ :\ |\ \}$. If A is a type and P_x is a proposition in which x of type A occurs free, then $\{x{:}A|P_x\}$ is the type of those element x of A for which P_x is true. The type of non-negative integers $\mathbb{N}$ and positive integers $\mathbb{N}^+$ are constructed using the subset type constructor.

- *Universes* of types $\mathbb{U}_i$ for $i = 1, 2, 3 \ldots$. $\mathbb{U}_i$ includes $\mathbb{U}_j$ for all $j < i$ as base types and is closed under the type constructors listed above. In Section 3 I drop these universe level indices without risk of confusion, although the indices must be stated when proving lemmas in the Nuprl system.

Every type has a natural equality relation on it. I write $x =_T y$ or $x = y \in T$ when I want to say that x and y are members of T and are equal by the equality relation associated with T. I write $x = y$ if T is obvious from context. Functions always respect type equalities, so if function f has type $S \rightarrow T$, then $fx =_T fx'$ whenever $x =_S x'$.

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 $f\ a$. Often I use infix notation for binary function application. For example, if $* \in (T \times T) \rightarrow T$, then for $*\langle a, b\rangle$ I write $a * b$.

Logic is injected into type theory using the propositions-as-types correspondence. Each predicate-logic expression corresponds to a type with the type being inhabited iff the predicate-logic expression is provable. The proof of a logical expression specifies exactly how to construct the term that inhabits the corresponding type. 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. Nuprl's logic is well-suited to constructive mathematics, but it also can support classical styles of reasoning.

3 Polynomial Algebra Development

3.1 The Approach

In mathematics, one talks about *the* ring of polynomials $A[S]$, given some ring A and some set of indeterminates S. However, a computer algebra system might have several *different* implementations of polynomials. Mathematically they are all isomorphic, but computationally they have distinct characteristics.

I am interested here in using type theory to characterize this practice, so I define a type $\mathsf{PolyAlg}(\mathcal{S}, \mathcal{A})$ that is the class of *all* implementations of polynomials over indeterminates S and ring A. This definition gets at the general properties of polynomials and abstracts away from the features of particular implementations. At the same time, I craft the definition such that a wide variety of useful computable functions could be constructed from an arbitrary inhabitant of the class.

My approach is based on the standard abstract approach found in textbooks such as Lang [Lan84] or Bourbaki [Bou74]:

1. Monomials in S are elements of a free abelian monoid over S.

2. Polynomials over S with coefficients from A are elements of a free monoid algebra over the monoid of monomials and the ring A.

From the above two constructions I derive the ring of polynomials and such functions as the injections from A and S, and 'substitution' function for instantiating indeterminates with elements of any ring B given an homomorphism from A to B.

I require implementations of classes to provide boolean-valued functions for deciding equality. Without these functions, one is rather limited in the kinds of computable functions that can be defined. This is standard practice in most computer algebra systems. However, the class definitions I give don't rely in any essential way on these functions.

3.2 The Free Abelian Monoid of Monomials

I start by defining EqType, a class for types with associated equality functions:

$$\mathsf{EqTypeSig} \doteq S{:}\mathbb{U} \times (S^2 \to \mathbb{B})$$

$$\mathsf{EqType} \doteq \{\langle S, eq\rangle{:}\mathsf{EqTypeSig} \mid \forall a, b{:}S.\ a\ eq\ b \Longleftrightarrow a =_S b\}\ .$$

The class EqType is the set of those elements of the signature $\mathsf{EqTypeSig}$ for which the equality function agrees with the equality relation of the type. I refer to elements of this class as *equality types*. I assume in what follows that an element $\mathcal{S}$ of EqType has form $\langle S, eq\rangle$.

The class signature for monoids, MonSig is:

$$\mathsf{MonSig} \doteq M{:}\mathbb{U} \times (M^2 \to \mathbb{B}) \times (M^2 \to M) \times M$$

and the class for abelian monoids AbMon is:

$$\begin{aligned}\mathsf{AbMon} \doteq \{\langle M, eq, *, e\rangle{:}\mathsf{MonSig}|&\\ &\forall a, b{:}M.\ a\ eq\ b \Longleftrightarrow a =_M b\\ &\wedge\ \forall a, b, c{:}M.\ (a * b) * c = a * (b * c)\\ &\wedge\ \forall a, b, c{:}M.\ a * b = b * a\\ &\wedge\ \forall a{:}M.\ (a * e) = a\\ \}\ .&\end{aligned}$$

I assume in what follows that an element $\mathcal{M}$ of AbMon has form $\langle M, eq, *, e\rangle$.

The definition of FAbMon(S), the class of free abelian monoids (with computable equalities) over the equality type $\mathcal{S}$ is:

$$\begin{aligned}\mathsf{FAbMon}(\mathcal{S}) \doteq\ & \mathcal{M}{:}\mathsf{AbMon}\\ & \times \iota{:}S \to M\\ & \times \mathcal{M}'{:}\mathsf{AbMon} \to \phi{:}(S \to M')\\ & \quad \to \{!\ \hat{\phi}{:}\mathsf{MonHom}(\mathcal{M}, \mathcal{M}') \mid \phi = \hat{\phi} \circ \iota\ \}\ .\end{aligned}$$

Here, $\mathsf{MonHom}(\mathcal{M}, \mathcal{M}')$, the set of monoid homomorphisms from M to M', is defined as:

$$\begin{aligned}\mathsf{MonHom}(\mathcal{M}, \mathcal{M}') \doteq \{\phi{:}M \to M' \mid\ & \phi\ e = e'\\ & \wedge\ \forall a, b{:}M.\ \phi\ (a * b) = \phi\ a *' \phi\ b\}\end{aligned}$$

and I use the abbreviation:

$$\{!x{:}T|P_x\} \doteq \{x{:}T|P_x \wedge \forall x'{:}T.\ P_{x'} \Rightarrow x = x'\}\ .$$

$\{!x{:}T|P_x\}$ should be be read as 'the type containing the unique x of type T such that P_x holds'.

The definition of $\mathsf{FAbMon}(\mathcal{S})$ is based on the characterization of a free abelian monoid over some set S as being an abelian monoid $\mathcal{M}$ and an injection ι of S into M, such that for any abelian monoid $\mathcal{M}'$ and mapping ϕ 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 ϕ there is a unique $\hat{\phi}$ such that the following diagram commutes:

$$\begin{tikzcd} S \arrow[d, "\iota"] \arrow[dr, "\phi"] & \\ M \arrow[r, "\hat{\phi}"] & M' \end{tikzcd} .$$

The definition of FAbMon(S) captures the requirement that there is an appropriate mapping into any abelian monoid by insisting that a function be supplied that generates this mapping. Specifically, let the triple $\langle \mathcal{M}, \iota, \Psi \rangle$ be some element of FAbMon(S). If Ψ is given as arguments an abelian monoid $\mathcal{M}'$ and a function ϕ, it must return the unique monoid homomorphism $\hat{\phi}$ from $\mathcal{M}$ to $\mathcal{M}'$ that satisfies the equation $\phi = \hat{\phi} \circ \iota$.

I could have written the definition of FAbMon(S) as:

$$\mathsf{FAbMonSig}(\mathcal{S}) \doteq \mathcal{M}{:}\mathsf{AbMon} \times S \rightarrow M$$

$$\begin{aligned} \mathsf{FAbMon}(\mathcal{S}) \doteq \{ & \langle \mathcal{M}, \iota \rangle{:}\mathsf{FAbMonSig}(\mathcal{S}) \mid \\ & \forall \mathcal{M}'{:}\mathsf{AbMon}\ \forall \phi{:}(S \rightarrow M') \\ & \quad \exists! \hat{\phi}{:}\mathsf{MonHom}(\mathcal{M}, \mathcal{M}').\ \phi = \hat{\phi} \circ \iota \ \}, \end{aligned}$$

not requiring Ψ to be explicitly supplied, but this would not have been nearly as computationally interesting.

An example of a useful function that can easily be generated from Ψ is as follows: if $\mathcal{S}$ is an equality type of indeterminates, then $\Psi\ \langle \mathbb{Z}, eq_{\mathbb{Z}}, +, 0 \rangle\ (\lambda x.1)$, is a function for calculating the total degree of a monomial (the sum of the powers of the indeterminates).

If one applies Ψ to the monoid of monomials and the injection function of some other implementation of $\mathsf{FAbMon}(\mathcal{S})$, one gets a function that translates representations of monomials into that used by the other implementation. Similarly, one could use the Ψ of that other implementation to translate back. These translation functions are inverses of one another and hence set up a constructive isomorphism between the two implementations. (This is just an instance of the fact that in category theory, initial objects are unique up to isomorphism.) A computable function defined over one implementation can be lifted to any other implementation using these translation functions. Therefore, all implementations are equally expressive, and every implementation is in a sense 'computationally complete'. Without the Ψ functions being explicitly required of implementations I would not have this computational completeness.

The standard mathematical representation for the carrier of a free abelian monoid over S is the set of functions of finite support of type $S \rightarrow \mathbb{N}$. (A function has *finite support* if it returns 0 for all but a finite number of elements of its domain.) A naive translation of this into type theory gives the type

$$\{ f{:}(S \rightarrow \mathbb{N}) \mid f \text{ has finite support} \} .$$

This type is not good constructively since there is no way of *computing* a finite support of elements of this type and hence giving a computable definition for Ψ. To make the finite support explicit, one could use instead the type:

$$\{\langle s, f\rangle{:}\mathsf{FinSet}(S) \times (S \to \mathbb{N}) \mid f \text{ has finite support } s\}$$

or the type $x, y{:}\mathsf{A\text{-}List}(S, \mathbb{N}^{+})//x$ permutation of y, where

$$\mathsf{A\text{-}List}(T, T') \quad \doteq \quad \{z{:}(T \times T')\mathsf{List} \mid \text{all elements of } z \text{ have distinct left sides}\}.$$

The quotient type is needed here because my class definition for free abelian monoids expects the equality associated with the carrier type of implementations to be the monoid equality; for example, it requires that all functions taking elements of the monoid carrier as arguments to respect this equality. However, one can have two a-lists that represent the same monomial but that are distinct according to the standard equality on lists. If type $\mathsf{FinSet(S)}$ of finite sets of elements of S is implemented using lists, one would similarly have to use a quotient type to hide the list order.

Nuprl's quotient type is an important tool for abstraction because it hides structural detail. Consider verifying the correctness of functions in a class implementation that use quotiented alists as described above. Nuprl's type theory forces one to show that the functions' behaviors are independent of the order of pairs in elements of the carrier, and so in this case the order is hidden.

3.3 The Polynomial Algebra

The definition of the class $\mathsf{FMonAlg}(\mathcal{G}, \mathcal{A})$ of free monoid algebras over the abelian monoid $\mathcal{G}$ and the ring $\mathcal{A}$ is similar in structure to that of the free abelian monoid:

$$\begin{aligned}
\mathsf{FMonAlg}(\mathcal{G}, \mathcal{A}) \quad \doteq \quad & \mathcal{B}{:}\mathcal{A}\text{-}\mathsf{Algebra} \\
& \times\ \iota{:}\mathsf{MonHom}(\mathcal{G}, \mathcal{B}\downarrow_{mmon}) \\
& \times\ \mathcal{B}'{:}\mathcal{A}\text{-}\mathsf{Algebra} \\
& \quad \to \phi{:}\mathsf{MonHom}(\mathcal{G}, \mathcal{B}'\downarrow_{mmon}) \\
& \quad \to \{!\ \hat{\phi}{:}\mathcal{A}\text{-}\mathsf{AlgHom}(\mathcal{B}, \mathcal{B}') \mid \phi = \hat{\phi} \circ \iota\ \}
\end{aligned}$$

where $\mathcal{A}$-$\mathsf{Algebra}$ is the class of algebras over the ring $\mathcal{A}$, $\downarrow_{mmon}$ is a forgetful class morphism that projects out the multiplicative monoid of an algebra and $\mathcal{A}$-$\mathsf{AlgHom}(\mathcal{B}, \mathcal{B}')$ is the type of $\mathcal{A}$-Algebra homomorphisms from $\mathcal{B}$ to $\mathcal{B}'$. As with the free monoid algebra, I require the universal projection function to be an explicit part of implementations.

I use this class to form polynomial algebras by supplying a free abelian monoid of monomials for $\mathcal{G}$ and considering $\mathcal{A}$ as the ring of coefficients. The addition and multiplication operations of the free monoid algebra are then the standard corresponding polynomial operations. A standard mathematical implementation of the carrier of a free monoid algebra is as the set of functions of finite support of type $G \to A$. One constructive implementation is:

$$x, y{:}\mathsf{A\text{-}List}(G, A^{-0})//x \text{ permutation of } y\ .$$

where A^{-0} is the type of the non-zero elements of the ring carrier A. If $\mathcal{G}$ is a monoid of monomials, then this implementation is considering polynomials as a-lists of monomials and their coefficients with the order of the a-lists quotiented out.

Finally, by defining a polynomial algebra implementation to be a pair of the implementation of monomials and the implementation of polynomials over the monomials, I arrive at a definition of an class for polynomial algebras:

$$\begin{aligned}\mathsf{PolyAlg}(\mathcal{S},\mathcal{A}) \;&\doteq\; \mathcal{M}{:}\mathsf{FAbMon}(\mathcal{S})\\ &\quad\times \mathsf{FMonAlg}(\mathsf{mon}(\mathcal{M}),\mathcal{A})\end{aligned}$$

where $\mathcal{S}$ is an equality type for the indeterminates, $\mathcal{A}$ is the coefficient ring and $\mathsf{mon}(\mathcal{M})$ is a function that selects out the monoid part of the triple $\mathcal{M}$.

I can define functions that project out from implementations of $\mathsf{PolyAlg}(\mathcal{S},\mathcal{A})$ the ring of polynomials and functions related to this ring. For example, injections from A and S, and a universal summation/evaluation function over polynomials that given a homomorphism from coefficient ring $\mathcal{A}$ to some other ring $\mathcal{B}$ and an assignment of values in B to the polynomial indeterminates, creates a function that maps polynomials over $\mathcal{A}$ into B.

3.4 Choices in Making Definitions

In algebra, subtyping relationships between algebraic classes are ubiquitous. With the simplest form of subtyping, *set subtyping*, the underlying signatures are the same; every member of the class of gaussian monoids is a member of the class of monoids. A richer form of subtyping, *forgetful subtyping*, involves forgetting components of signatures; every member of the class of groups can be considered a member of the class of monoids if one forgets the inverse operation. Algebraic notation without forgetful subtyping quickly becomes extremely cluttered with trivial forgetful class morphisms. I can implement set subtyping with Nuprl's set type, but unfortunately, the rigid nature of Σ types prevents me from implementing forgetful subtyping. An ad-hoc partial solution that I've adopted for now is to minimize the number of class signatures; in my implementation of the abelian monoid class, I actually use the group signature class rather than a monoid signature class. When specifying an inhabitant of the abelian-monoid class I then have to supply a dummy inverse function.

There are numerous proposals [Wir90] for forms of dependent record types that support forgetful subtyping. I am very interested in trying to adapt one of them to Nuprl's type theory.

Many algebraic definitions have computational content when considered constructively. The free class definitions in Section 3 have these universal projection functions. I have experimented with defining the permutation relation on lists such that the computational content of the proposition 'list x is a permutation of list y' when the proposition is true is a permutation function that permutes list x into list y. The content of a relation expressing membership of an element x of a commutative ring A in a finitely-generated ideal $\langle a_1 \ldots a_n\rangle$ could be a list of elements $c_1 \ldots c_n$ of A such such that $x = c_1 a_1 + \ldots c_n a_n$.

One challenge in writing class definitions is in deciding what computational content to explicitly bring out and what to leave implicit. Consider the discussion in Section 3.2 of alternative definitions for $\mathsf{FAbMon}(\mathcal{S})$. Definitions become very tedious if one takes a conservative approach and makes explicit all computational content. If the equality relation on the carrier type of members of algebraic classes has computational content, then every predicate involving equality such as $\forall x, y.\ x + y \Rightarrow y + x$ or $\forall x, y.\ x = y \Rightarrow y = x$ also has computational content. When defining classes in Nuprl, I would have to include such predicates in my class signatures rather than the right-hand-side of set types, and I would lose all set subtyping properties.

In Nuprl's type theory, the equality relation naturally associated with a type corresponds to a unit type when true and a void type when false; its computational content is always trivial. I have used these natural equality relations in all my class definitions; I get set subtyping properties and get the benefit of the built-in support in Nuprl's type theory for these relations. An example of the built-in support is that elements of function types always respect these equality relations. This design decision seems in accord with current practice in constructive algebra [MRR88] and in the computer algebra system Axiom [JS92] where much attention has been paid to constructivity.

4 Practical Work

4.1 Theory Development

All the class definitions described in Section 3 have been entered into the Nuprl system, but as yet I haven't proved much using them. I would like eventually to tackle reasoning about algorithms in computational algebra and these algorithms assume much more detailed structure than my definitions provide. For example, it is fundamental that every polynomial can be uniquely expressed as a sum of monomials. The standard way of proving this involves first constructing a canonical implementation of the polynomial algebra class involving a representation of functions of finite support. I also need to reason about orderings on monomials.

My theory development efforts to date have been mostly directed at building up a set of foundational theories that are sufficient to support reasoning about implementations of polynomials. Relevant theories I have developed include:

1. Permutations: I have shown that the set of permutation functions on a type is a group and that every permutation on a finite type can be expressed as a composition of swaps. I have defined a permutation relation on lists and shown that iterated abelian operations over lists respect this relation. This theory of permutations supports the definition of finite sets and multisets.

2. Gaussian Monoids: I have proven that every non-unit can be uniquely factorized into primes up to associates and permutations.

3. Quotient algebras: I have developed the basic theory of normal subgroups and ideals, have defined quotienting operations on groups and rings and have started on proving isomorphism theorems.

One ongoing part of my work is developing a discipline for using Nuprl's quotient types. Quotient types are needed for the carrier type in implementations of my algebraic classes, as explained in Section 3. I am also relying on them to form quotient algebras.

4.2 Tactic Development

Tactics are functions in the programming language ML for executing proof development strategies. Some examples are given in the next section. I identify three kinds of tactics that I have developed that have been particularly useful in my algebraic proofs to date.

1. Tactics for arithmetic reasoning. I have implemented a procedure for solving linear arithmetic problems and have augmented it to automatically take advantage of the linear properties of non-linear arithmetic expressions (for example the list length function). These tactics proved themselves invaluable when reasoning about permutation functions (bijections on $\mathbb{N}n \to \mathbb{N}n$) and the 'select' function for picking the ith element from a list.

2. Tactics for rewriting with respect to equivalence relations. These tactics automatically index into Nuprl's library to look up lemmas about which relations functions respect. Example equivalence relations that I have come across in algebra are the permutation relation on lists, the 'associated' relation that comes up in divisibility theory, and the 'equal mod an ideal' relation that comes up in ring theory.

3. Tactics for automating the proof of type inclusion relations. Sometimes these tactics call on other tactics such as the arithmetic reasoning tactics when integer subrange types were involved.

4.3 An Example Proof

To illustrate the level and style of reasoning in Nuprl I show in Figure 1 a proof that free abelian monoids over a fixed set are unique up to isomorphism. Figure 2 gives the two lemmas that are explicitly referenced in the proof.

One proves a theorem in Nuprl by first entering it as a goal to be proved and then repeatedly invoking tactics to break goals into simpler subgoals. In Figure 1, the initial goal is indicated by ① and the tactics immediately follow each `BY`.

The logical formulas are shown exactly as they appeared when the theorem was interactively proved. Goals and tactics are entered using a structured editor, so there is no need for the notation to be completely unambiguous. For example, the `|·|` is the carrier projection function for both elements of `EqType` and `AbMonoid`. Most of the notation should be self-explanatory. The `.inj` and `.umap` postfix operators project out the second and third components of elements of the `FAbMon{i}(S)` class. The display of the projection operator for the first component has been suppressed to improve readability since it is obvious where it is used. (Visual notation can be changed in Nuprl in a matter of seconds, so it is easy if desired to make it visible.) The `{i}` parameters specify the level of the carrier universe terms in the class definitions and are implicitly universally-quantified over.

At ②, univeral quantifiers were stripped. The `...` and `...a` indicate applications of the `Auto` tactic that does certain obvious steps of inference and solves *well-formedness* subgoals. Nuprl's type theory is sufficiently rich that the well-formedness of expressions is in general undecidable and so well-formedness is checked by proof. At ③, the existential quantifiers were instantiated by the given terms and the definition `Invfuns` was split open into its two component parts. At ④, lemma

```
free_abmon_unique:

①  ⊢ ∀S:EqType{i}
   |   ∀M:FAbMon{i}(S)
   |    ∀N:FAbMon{i}(S)
   |     ∃f:MonHom(M,N). ∃g:MonHom(N,M). InvFuns(|M|,|N|,f,g)
   |
②  BY (UnivCD...a)
   |
   1. S: EqType{i}
   2. M: FAbMon{i}(S)
   3. N: FAbMon{i}(S)
   ⊢ ∃f:MonHom(M,N). ∃g:MonHom(N,M). InvFuns(|M|,|N|,f,g)
   |
③  BY (InstConcl  ['M.umap N N.inj';'N.umap M M.inj']
   |    THENM D 0...a)
   |\
   | ⊢ (N.umap M M.inj o M.umap N N.inj) = Id{|M|} ∈ (|M| → |M|)
   | |
④  | BY (BLemma `free_abmon_endomorph_is_id`...a)
   | |
   | ⊢ ((N.umap M M.inj o M.umap N N.inj) o M.inj) = M.inj ∈ (|S| → |M|)
   | |
⑤  | BY (RW CompIdNormC 0
   |      THENM RewriteWith [] ``free_abmon_umap_properties_1`` 0 ...)
   |
   ⊢ (M.umap N N.inj o N.umap M M.inj) = Id{|N|} ∈ (|N| → |N|)
   |
⑥  BY ...
```

Figure 1: Proof of uniqueness of free abelian monoid up to isomorphism

```
free_abmon_umap_properties_1:

  ∀S:EqType{i}
   ∀M:FAbMon{i}(S)
    ∀N:AbMonoid{i}
     ∀p:|S| → |N|. ((M.umap N p o M.inj) = p ∈ (|S| → |N|))

free_abmon_endomorph_is_id:

  ∀S:EqType{i}
   ∀M:FAbMon{i}(S)
    ∀f:MonHom(M,M)
     ((f o M.inj) = M.inj ∈ (|S| → |M|))
     ⇒ (f = Id{|M|} ∈ (|M| → |M|))
```

Figure 2: Supporting lemmas for free-abelian-monoid uniqueness proof

`free_abmon_endomorph_is_id` was backchained through. In order to verify the instantiation of `f` in this lemma, the `Auto` tactic automatically picked up on a lemma stating that the composition of two homomorphisms is a homomorphism.

At ⑤, `RW CompIdNormC 0` normalized the function compositions (the `o`'s) in the conclusion, making them right-associated. `CompIdNormC` is a rewriting function for normalizing expressions over the monoid $\langle T \to T, \mathtt{o}, \mathtt{Id}(T)\rangle$ for any type T. Its ML definition is:

```
let CompIdNormC = MkMonoidNormC `comp_id_monoid_wf` ;;
```

where `comp_id_monoid_wf` is the name of a lemma that states that for all types T, $\langle T \to T, \mathtt{o}, \mathtt{Id}(T)\rangle$ is a monoid . I have written functions for creating rewriting function similar to `MkMonoidNormC` for algebras including groups, abelian groups and rings. The `RewriteWith` tactic repeatedly rewrites with the indicated lemmas and hypotheses. The tactics at ⑤ completed this branch of the proof. The other branch of the proof, ⑥, was proven with tactics identical to those at ④ and ⑤.

5 Applications

5.1 Interaction with Computer Algebra Systems

I am exploring ways in which Nuprl could usefully interact with the Weyl computer algebra system being developed here at Cornell [Zip93]. One scenario is for Nuprl to behave as an *algebraic oracle* for the computer algebra system. In the course of calculations, Weyl creates new instances of algebraic structures and sometimes would like to decide which algorithm to use based on properties of these structures. However the properties might require some theorem proving work to make them apparent. This is where Nuprl comes in.

A simple trial example I am looking at concerns reasoning about rings over integers, rationals and polynomials that have been quotiented by finitely generated ideals. A query might proceed as follows:

1. Weyl asks Nuprl Q1:"is the quotiented ring $\mathbb{Z}/(1009)$ an integral domain?"

2. Nuprl looks up a lemma about quotienting rings by ideals and reduces Q1 to Q2:"is the principle ideal (1009) a prime ideal?"

3. Nuprl looks up a second lemma about when principle ideals over the integers generated by positive numbers are prime ideals and reduces Q2 to Q3:"is the natural number 1009 prime?"

4. Weyl happens to have an efficient primality testing algorithm implemented, so Nuprl asks Weyl Q3.

5. Weyl answers "yes" to Q3, and so Nuprl replies "yes" to Weyl's Q1.

Currently, Nuprl answers such queries by exhaustively forward and backward chaining through a small database of lemmas. As I increase the size of the database, I will be interested in exploring more efficient inference strategies for answering queries.

5.2 Algebraic Programming

I can use Nuprl's Σ type and set type to construct ADT class definitions in a way similar to that which I used for the abelian monoid class in Section 3.2. I can treat classes as an abstract specifications of submodules of a functional program and verify the correctness of the program given an arbitrary implementations of these classes. I also can verify the correctness of particular implementations of the classes. Nuprl's type theory is therefore a good uniform framework for formally developing structured functional programs using ADT's.

Moreover, I have shown in this paper that Nuprl's type theory is adequate for defining free or initial classes. Final classes should be no more difficult to construct. The importance of this is that in the ADT community there has been a debate going on about the relative merits of loose, initial and final algebraic specifications [Wir90]. In Nuprl, all three paradigms can be explored.

6 Conclusions

I have presented here some of the first steps I have taken in implementing abstract algebra in an interactive theorem proving setting. To my knowledge, this is the first attempt at developing a general theory of polynomial algebras in either a classical or a constructive theorem proving environment. The definitions used in developing the polynomial algebras illustrate the expressiveness of Nuprl's type theory, making full use of its Π,Σ, set and quotient types.

As I described in Section 5, the work has promising applications in the formal development of software, as well as in the development of new more powerful and more rigorous computer algebra systems.

7 Acknowledgements

My formulation of the class definitions for the polynomial algebra benefited from many discussions with Robert Constable. The interface work between Nuprl and Weyl is being done in close collaboration with Weyl's architect, Richard Zippel. I thank the anonymous referees for their comments on an original draft of this paper.

References

[AL90] Mark Aagaard and Miriam Leeser. The implementation and proof of a boolean simplification system. Technical Report EE–CEG–90–2, Cornell School of Electrical Engineering, March 1990. In the*Oxford Workshop on Designing Correct Circuits*, September, 1990.

[Bai93] Anthony Bailey. Representing algebra in LEGO. Master's thesis, University of Edinburgh, November 1993.

[BC93] David A. Basin and Robert L. Constable. Metalogical frameworks. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*. Cambridge University Press, 1993.

[Bou74] Nicolas Bourbaki. *Algebra, Part I.* Elements of Mathematics. Addison-Wesley, 1974.

[BV90] David A. Basin and Peter Del Vecchio. Verification of combinational logic in Nuprl. In M. E. Leeser and G. M. Brown, editors, *Hardware Specification, Verification, and Synthesis: Mathematical Aspects*, pages 333–357. Springer Verlag, 1990. LNCS 408.

[C+86] Robert Constable et al. *Implementing Mathematics with The Nuprl Development System.* Prentice-Hall, NJ, 1986.

[CH92] Jawahar Chirimar and Douglas J. Howe. Implementing constructive real analysis: Preliminary report. In *Constructivity in Computer Science*, volume 613 of *Lecture Notes in Computer Science*, pages 165–178. Springer-Verlag, 1992.

[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.

[FGT92] 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.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.

[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.

[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.

[How88] D. Howe. *Automating Reasoning in an Implementation of Constructive Type Theory.* PhD thesis, Cornell University, Ithaca, NY, April 1988.

[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.

[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.

[Jac94] Paul B. Jackson. *Enhancing the Nuprl Theorem Prover and Applying it to Constructive Algebra.* PhD thesis, Cornell University, 1994. Forthcoming.

[JS92] Richard D. Jenks and Robert S. Sutor. *AXIOM: the Scientific Computation System.* Springer-Verlag, 1992.

[Lan84] Serge Lang. *Algebra.* Addison-Wesley, 2nd edition, 1984.

[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.

[MP85] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery, SIGACT, SIGPLAN, 1985.

[MRR88] Ray Mines, Fred Richman, and Wim Ruitenburg. *A Course in constructive Algebra.* Universitext. Springer-Verlag, 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.

[Zip93] Richard 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.