Formalization of Divisibility Theory in Nuprl

Paul B. Jackson
Laboratory for Foundations of Computer Science, University of Edinburgh, King's Buildings, Edinburgh EH9 3JZ, United Kingdom

Abstract. The formalization of divisibility theory over cancellation monoids in Nuprl is described. The main theorems presented concern the existence and uniqueness of factorisations. Issues addressed include how to make formalized mathematics readable and the use of automated inference. The constructive nature of mathematics in Nuprl is also discussed.

Contents

1. Introduction

The aim of this paper is to demonstrate that readable formal mathematical developments including readable formal proofs can be produced without too excessive tedium.

The work was done in the NUPRL proof development system (C+86; Jac95a). NUPRL has been developed over the past 10 years at Cornell University primarily as a system for reasoning about functional programs and exploring the possibilities of synthesizing functional programs from proofs in constructive mathematics. It features a well-developed user interface and a large suite of proof commands to assist in the partial automation of proof.

The paper presents a development of the theory of divisibility in abelian monoids with a cancellation property. Conditions were derived under which factorisations in these monoids exist and are unique. The development tackled some basic concrete issues common in algebra: reasoning about permutations of finite sequences and with equivalence relations and congruence properties. The fundamental theorem of arithmetic stating that every natural greater than one can be uniquely factored into primes was proven as a special case of the main theorem.

When mathematics is formalised and mechanised, there are many choices that have to be made about what sequence of definitions and theorems to choose, about how exactly to state the theorems, and what proofs to go for. These choices are always influenced to a lesser or greater degree by the particular formalism used and by the kind of mechanical support available.

A conscious effort was made in the development presented here to keep the style close in spirit to that which might be found in a mathematics text-book. Rather than make choices and find proofs which were most tractable mechanically, I wanted to see how the automation in NUPRL could be improved to support well argument styles I was finding in text-books and that seemed more mathematically natural. I was also motivated by a desire to make the development as readable as possible.

The paper has been organized as follows: Section 2 provides an overview of the constructive type-theoretic formalism underlying the work. Much of this section can be skimmed on a first reading. Section 3 gives an introduction to the NUPRL system and describes aspects which are important for understanding the presentation of the NUPRL development given in this paper and the accompanying development listings.

The next three sections give the an overview of the development. Section 4 introduces how algebraic classes were defined in NUPRL's type theory. Section 5 covers the definition of permutation functions and relations which were subsequently necessary. Section 6 contains a description of the main development of the factorization theorems and includes abbreviated proofs.

Section 7 discusses the style of the development and its presentation. Section 8 highlights what automation helped in the development. Section 10 comments on the adequacy of constructive type theory for mathematics. Section 9 covers related work. Section 11 summarizes the main contributions.

2. Type Theory

2.1. Introduction

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. Nuprl uses instead a *type theory* which takes the place of both predicate calculus and set theory.

I give here some background on Nuprl's type theory. However, it is the intent of the author that much of this paper should be comprehensible to readers with little or no knowledge of type theory.

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 π_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. 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.

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). One significant difference is that Nuprl's type theory has extra types such as its *set* type (Con85) and *quotient* type (Con85). Another is that the computation language is considered to be untyped and defined prior to the type theory itself.

Allen did the basic work on giving a semantics for NUPRL's type theory (All87a; All87b). This semantics amounted to defining a relation between types and terms saying when a term had a given type and when two terms were considered equal. Howe (How91) 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. Howe has also proposed much simplified alternatives to NUPRL's current type theory (How93).

2.2. 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 paper include:

- The type `Void` with no elements, the boolean type $\mathbb{B}$ with two elements `tt` and `ff`, and the integers $\mathbb{Z}$.

- 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 $x \in A$, the type is written as $A \times B$. $\times$ is assumed to associate to the right. Dependent-product types are elsewhere known as Σ or dependent sum types.

- 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 $x \in A$, the type is written as simply $A \rightarrow B$. $\rightarrow$ is assumed to associate 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.

  Dependent-function types are elsewhere known as Π, cartesian-product or dependent-product types.

- 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.

- A polymorphic list type A 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 sequence is denoted by [].

- *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. Often if a subscript is the variable i it is dropped, and the universe $\mathbb{U}_{i+1}$ is abbreviated as $\mathbb{U}'$.

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. Often when T is obvious from context the $\in T$ is dropped.

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

2.3. Propositions

An often confusing aspect of constructive type theory is that all the logical connectives and quantifiers are defined in terms of types. Thoughout NUPRL developments logical notation is always used when types are considered as propositions, so NUPRL users and readers of NUPRL developments need not for the most part be familiar with this correspondence between propositions and types.

The definitions of the connectives and quantifiers are:

$$\begin{aligned}
\bot &=_{def} \mathsf{Void} \\
A \wedge B &=_{def} A \times B \\
A \vee B &=_{def} A + B \\
A \Rightarrow B &=_{def} A \to B \\
\forall x{:}A.\ B_x &=_{def} x{:}A \to B_x \\
\exists x{:}A.\ B_x &=_{def} x{:}A \times B_x \\
\mathbb{P}_i &=_{def} \mathbb{U}_i
\end{aligned}$$

The symbol $\bot$ denotes falsity. $\mathbb{P}_i$ denotes the type of propositions at level i, and, as with $\mathbb{U}_i$, the suffix i is often suppressed. Negation, $\neg A$, is defined as $A \Rightarrow \perp$, and bi-implication (if and only if) $A \iff B$ is defined as $(A \Rightarrow B) \wedge (B \Rightarrow A)$. 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).

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 paper, 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.

2.4. Sequents

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 and is often written as:

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

Each hyphesis 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. If H_i is a declaration $x{:}T$, it binds free occurrences of x in $H_{i+1} \ldots H_n$ and in C. A sequent is considered true if one can prove the conclusion C under the hypotheses $H_1, \ \ldots \ , H_n$.

Type-theoretically, all clauses of a sequent are types. A hypothesis thought of as a proposition declares the type of a variable which is normally never visible. A sequent is true just when there exists a computable function from the types of the hypotheses to the type of the conclusion.

2.5. 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: A consequence of this is that there is no automatic 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.

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 having several Nuprl rules have special well-formedness premisses.

In practice nearly all well-formedness proofs are automated so the user need not be concerned with them. Unfortunately checking well-formedness by proof is much slower than checking by some completely automatic type checker, and is a major source of inefficiency in the Nuprl system. Nearly all other theorem provers do their well-formedness checking entirely by completely automatic means, distinct from proof generation.

3. Mechanization

3.1. Overview

The current release of Nuprl, NUPRL V4.2 works on Unix-based workstations that run X-Windows. NUPRL is written in a combination of Common Lisp and the original Edinburgh version of the functional language ML. To run efficiently it requires a commercial implmentation of Lisp such as Allegro Common Lisp or Liquid Common Lisp from Harlequin. The release and other information about NUPRL is available from Cornell's web site at URL `http://www.cs.cornell.edu/`.

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.

3.2. Terms

In NUPRL parlance, a *term* is a general-purpose uniform tree-shaped data-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. All propositions in Nuprl's logic are represented as terms, as are all expressions and types in its type theory. This is not the only use of terms. NUPRL also uses terms to represent the contents of all the kinds of objects in theories except proofs. In particular text in ML declarations and comments is kept as terms.

A basic notation for terms uses prefix names and has subterms surrounded by parentheses and seperated by `;`'s. For example, the term $3+4$ is `add(3;4)` in this notation. Parameters to terms are enclosed in braces and are listed before the subterms. For example, terms can take strings of characters as parameters. The string of characters `"nuprl"` is represented in basic notation as `string{nuprl:s}()`. The `:s` indicates that '`nuprl`' should be considered a parameter of string type. Variables that a term binds in a subterm are listed before that subterm and seperated from the subterm by a '`.`' (period). If there is more than one binding variable for a subterm, they are seperated by '`,`'s. For example, the term $\lambda x, y.x$ might be in basic notation `lambda2(x,y.x)`.

3.3. 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 complete control over how the fixed text and arguments of a constructor are displayed. For conciseness, display forms can be set up to optionally hide less interesting arguments to terms. Format commands in display forms control line-breaking and indentation. Parentheses can be generated based on precedences assigned to terms. Special display forms can be selected for when similar terms are nested inside one another.

Using display forms the term in basic notation

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

is usually displayed as:

```
∀ i,j:ℤ. ∃ k: ℤ. k ≥ i * j
```

Note here that 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. However it should be straightforward to use more advanced display techniques to generate displays using multiple sizes and kinds of fonts, and two dimensional layout of formulae.

All terms shown in this paper have been automatically formatted by NUPRL's display routines.

Terms are interactively edited and viewed exclusively using a structure editor. The structure 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.

There is no requirement that the displayed form of terms be mechanically parseable. This gives the user great versatility in devising concise display forms. When users are interacting with NUPRL they can resolve ambiguities by selectively suppressing display forms. Unfortunately this option isn't available when reading printed display forms so extra care is needed when NUPRL developments are presented in print.

Note that for clarity argument suppression is frequently used in the definitions and theorems presented in this paper. Definitions might be puzzling because some variable only seems to occur on one side of the definition. In these cases, there is always at least one occurrence of that variable in some suppressed subterm on the other side of the definition.

Note too that parenthesisation assumes that infix functions have higher precedence than infix atomic predicates which in turn have higher precedence than infix logical connectives. Also, binding constructs such as the logical quantifiers have scope which extends as far as possible to the right.

3.4. Theories

The NUPRL V4.2 data-base of definitions and theorems is divided into *theories*. Each theory contains a linear sequence of *library objects* or *objects* for short. The kinds of objects include:

abstraction An abstraction object defines a new term constructor or *abstraction* in terms of other previously defined abstractions and primitive terms whose meanings are fixed. 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.

display form definition A display form object controls the appearance of a primitive or abstract term.

comment Comment objects can contain arbitrary text and terms.

ML An ML object contains one or more ML declarations. 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.

theorem A theorem object has several components. These include a proposition — often called the goal or main goal — which is the statement of the theorem, and a proof script which is a collection of ML tactics which prove the proposition. In addition when a proof script is executed or a proof is initially constructed, a theorem object contains a proof tree datastructure which records intermediate stages of a proof and shows how the proof parts depend on each other.

Figure 1 shows a listing of part of a theory dealing with functions.

```
*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
```

Figure 1. Partial Listing of Theory on Functions

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 in some appropriate sense. For example, the status of a theorem object is shown as `*` just when the theorem has been proven. Other status characters include `#` for incomplete, and `-` for bad in some sense. An ML object would have `-` status if it contained a declaration that didn't parse or type-check. The capital letter gives the kind of the object. For example: `D` for display form, `C` for comment, `M` for ML, `T` for theorem.

Following the kind of an object is the object's name and the contents of an object. Frequently for conciseness the proof scripts and proof trees of theorem objects are not shown in listings.

Currently over 30 theories have been defined in NUPRL V4.2. See the NUPRL V4.2 distribution for details.

3.5. Definitions

A definition in NUPRL has at least two parts: an abstraction for the logical structure of the definition and a set of one or more display forms for controlling the visual appearance of the abstraction. If the definition is for a term in NUPRL's type theory, the definition also usually has a well-formedness lemma. Well-formedness lemmas give typing information about abstractions and are used by NUPRL's type checking tactics.

Definitions are sometimes presented in this paper by giving just the abstraction definition and sometimes by combining information from the abstraction definition with the well-formedness lemma when it's helpful to see the typing information. For example, the definition for `compose` can be presented as:

```
compose:
  ∀A,B,C:𝕌. ∀f:B → C. ∀g:A → B.
    f o g = (λx.f (g x)) ∈ A → C
```

The structure of most display forms should be obvious. In the more complicated cases a simplified left-hand-side of a display form definition is shown to make clear what the arguments are and what is the fixed text of the display form. For example, a simplified left-hand side of the `compose` display form is `<f> o <g>` — arguments to display forms always being surrounded by '<>'s.

3.6. Proofs

Proofs in NUPRL are constructed using a paradigm introduced in the LCF system (GMW79) and adopted in other systems such as HOL (GM93) and ISABELLE: in this paradigm all proofs are ultimately constructed by a fixed core set of functions which implement the rules of logical calculus of the prover, in NUPRL's case its type theory. A suite of functions are built on top of this core set to provide users with varying degrees of automated inference. However no matter how complex these are, the soundness of the prover only relies on the correctness of the implementation of the core set.

LCF introduced the functional language ML for writing these higher-level functions which it called *tactics*. NUPRL adopted both the ML language and the tactic notion from LCF. NUPRL's tactics are described in the next section.

Proofs are constructed in a refinement style: a proof editor presents the user with an unproven sequent and the user enters a tactic which can either prove the sequent or refine it into one or more easier-to-prove sequents from which it follows. The proof process starts with the user entering a conjectured theorem. this becomes the initial goal. The user then repeatedly applies tactics to this goal and subsequent subgoals until hopefully none remain. The theorem is then proven.

An unusual feature of NUPRL is that proof-tree data-structures are maintained as proofs progress that record the subgoals at all intermediate stages of proofs. Most other interactive provers only maintain the unproved subgoals at the fringe of partial proof trees. Maintaining whole proof trees makes interactive development of proofs considerably easier; at any stage it's straightforward to review the structure of a proof and go back to experiment with alternate proof strategies. Proof trees also serve to document and explain proofs.

3.7. TACTICS

There are 4 main classes of tactics that users explicitly invoke in nearly all NUPRL developments.

Decomposition For breaking apart the outermost structure of of the conclusion or hypotheses of a sequent. Tactics for instantiating quantifiers are included in this class.. Typical names have `D` or `Inst` as roots.

Structural `Assert` invokes the cut rule; it introduces intermediate stages in proofs. `Decide` and `Cases` are for doing case splits. `Thin` deletes hypotheses which are no longer needed and clutter the hypothesis list.

Chaining Backward-chaining tactics `BLemma`, `BHyp`, `Backchain` apply lemmas and hypotheses to the conclusion to reduce it to 0,1 or more easier to prove subgoals. Forward-chaining tactics `FLemma` and `FHyp` apply lemmas and hypotheses to hypotheses to infer new hypotheses.

Rewriting Tactics with the names involving `RW` or `Rewrite` apply equations to carry out substitutions in hypotheses and the conclusion. Particular instances of rewrite tactics are `Unfold` for opening definitions and `AbReduce` for simplification.

Hypotheses are referred to by number. `0` refers to the conclusion and negative numbers index into the hypothesis list from the right-hand end. Optional arguments to tactics and variants on the basic ones allow for fine control over their behaviour. Explicit instantiating arguments can be provided for lemmas and hyps when using them for chaining. Rewrite rules can be targeted to single positions even when they apply in several places. Both rewriting and chaining tactics employ second-order pattern matching (BD77) to smoothly handle expressions with binding structure.

Tactics are combined using tacticals: the infix `THENM` sequences tactics, as does `SeqOnM` which takes a list of tactics as arguments. Other tacticals allow for repetition of tactics or trying alternative tactics. The tactic `Auto` is run after virtually every invocation of the tactics above. It takes care of type-checking, reasoning about type inclusions, calling decision procedures for arithmetic and binary relations and trivial propositional reasoning. It is good for checking auxiliary conditions on rewrite rules and chaining lemmas. Its presence is usually indicated by a `...` following tactic text. This is a display form for the invocation.

Subgoals created by tactics are labelled with their kind (for example *main*, *well-formedness*). Some tacticals allow the steering of tactics to subgoals of different kinds. The `THENM` tactic steers its second argument to only main subgoals.

Section 8 describes some of the features of the rewrite tactics and the decision procedures in more detail.

4. Monoid Classes

Nuprl's dependent product type can be used to create types for algebraic classes of objects. The product type `GrpSig` which underlied the the monoid class had definition

```
grp_sig:
  GrpSig ==
    car:𝕌
    × eq:(car → car → 𝔹)
    × le:(car → car → 𝔹)
    × op:(car → car → car)
    × id:car
    × (car → car)
```

Elements of this type were 6-tuples of form `<car,eq,le,op,id,inv>` where the types of `eq` and components to the right depended on the component `car`. When a monoid with carrier `car`, binary operation `op`, and identity `id` was represented as such a 6-tuple, dummy values were supplied for the `eq`, `le`, and `inv` components. These components were useful in other class definitions: `inv` was used for the inverse operation of a group, `eq` for a computable equality function, and `le` for a computable inequality function . Defining a monoid as a 6-tuple as above rather than more obviously as 3-tuple simplified reasoning about subclass relationships. For example, using the `GrpSig` type as shown above, every group and every monoid with a decidable equality was also automatically a member of the monoid class.

`GrpSig` was automatically created by the class declaration shown in Figure 2. This class declaration was an ML object in the `groups_1` NUPRL theory. Various NUPRL definitions hid less-readable underlying ML syntax. The `g ∈ GrpSig` and the text in parentheses in the field section (e.g. `(|g|)`) were comments; they didn't translate to anything in the ML syntax.

The class declaration also automatically created definitions for selector functions for each of the fields of class members. For example, the definition for the selector of the `op` field was:

```
grp_op:
  ∀g:GrpSig. *g = g.2.2.2.1 ∈ |g| → |g| → |g|
```

where the postfix `.1` and `.2` notation is for the selector functions for first and second components of pairs. The display forms for the selectors of `GrpSig` components were often set up to suppress the postfix arguments `g`.

```
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
     (1g)  id : car
     (∼g)  inv : car → car

  Universe: 𝕌'
```

Figure 2. Declaration for `GrpSig` Class

Basic classes for monoids `IMonoid` and `IAbMonoid` were introduced by the definitions

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

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

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

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

IMonoid == {g:GrpSig| IsMonoid(|g|;*;1)}

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

The `I` prefix stands for *indiscrete*, a term used in constructive mathematics for sets when the equality relation on the set is not known to be computable.

Most theorems in Section 6 assume that the monoid has a cancellation property. This was expressed using the predicate

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

with both S and T set to the monoid carrier.

5. Permutations

The factorisation work required reasoning about permutations of finite sequences represented as lists. Specifically a relation was needed for saying when one list was a permutation of another. Such a relation can be defined in several ways. Constructively, some of these definitions only work when a computable equality function is available on list elements. I chose to work with a definition in terms of permutation functions that didn't require such a function.

5.1. Permutation Functions

Classically, a permutation on a set T is a bijection of type $T \to T$. Implicit in the definition of any bijection f of type $A \to B$ is the existence of an inverse function b of type $B \to A$. There is no way in general of computing b from f, even though b is a useful function, so constructively a bijection is commonly defined as a pair of functions $\langle f, b\rangle$ 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 definitions for `PermSig` type

```
PermSig(T) == (T → T) × (T → T)
```

and for selector functions with postfix notation `.f` and `.b` were introduced by the class declaration shown in Figure 3.

```
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 3. Signature Class for Permutation Functions

Other relevant definitions were

```
Id == λx.x

f o g == λx.f (g x)

InvFuns(A;B;f;g) == g o f = Id ∧ f o g = Id
```

`Sym(n)`, the symmetry group on `n` elements, was defined as:

```
Sym(n) == Perm(ℕn)
```

where ℕ was a special display form with simplified definition

```
ℕ<j>== {0..<j>⁻}
```

and

```
{i..j⁻} ==  {k:ℤ| i ≤ k < j}
```

5.2. Permutation Relation

The definition of the permutation relation `permr` on lists said that two lists `as` and `bs` were a permutation of each other if they were the same length and the forward permutation function permuted the `bs` into the `as`. The definition was:

```
permr:
  as ≡ bs
   == (|as| = |bs|) c∧ (∃p:Sym(|as|)
                              ∀i:ℕ|as|. as[(p.f i)] = bs[i])
```

where `|·|` was the length function for lists, and `as[i]` was 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))
```

Confusion of the length function with the monoid carrier selector function which has the same notation should be should be easy to resolve by considering context and the type of the argument.

The notation `c∧` was for a *conditional and* constructor. Its definition was the same as the usual ∧ of Nuprl's type theory, but it was type checked slightly differently. To prove `P c∧ Q` well-formed, it was sufficient to prove `P` well-formed, and then assume `P` true to prove `Q` well-formed. With `P ∧ Q`, `Q` had to be proved well-formed irrespective of th truth of `P`. The conditional 'and' was needed to guarantee that the index argument to the `select` functions were always in range: `select` had typing lemma

```
∀A:𝕌. ∀l:A List. ∀n:ℤ.  0 ≤ n ⇒ n < |l| ⇒ l[n] ∈ A
```

The `length` and `nth_tl` definitions above were recursive definitions. This is indicated by the `==r` rather than the `==` notation in their presentations above. General recursive function definitions are encoded in NUPRL by using the Y combinator; the computation language of NUPRL is untyped so the Y combinator is expressible in it. Recursions were shown to well-founded when the well-formedness lemmas for the definitions were proven.

A useful lemma involving the `permr` relation was

```
mon_reduce_functionality_wrt_permr:
  ∀g:IAbMonoid. ∀xs,ys:|g| List.
    xs ≡ ys ⇒ Π xs = Π ys
```

where the generalized list product function **Π** had the definition:

```
reduce:
  ∀A,B:𝕌. ∀f:A → B → B. ∀k:B. ∀as:A List.
    reduce(f;k;as)
    = case as of [] => k | a::as' => f a reduce(f;k;as') esac
    ∈ B

mon_reduce:
  ∀g:IMonoid. ∀as:|g| List.  Π(g) as = reduce(*g;1g;as) ∈ |g|
```

Note that the monoid argument to **Π** was often suppressed. The `mon_reduce_functionality_wrt_permr` lemma was proven using an induction lemma

```
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]}
```

which reduced the problem to one of checking that products were unchanged by transpositions.

6. Divisibility Theory

Divisibility theory is commonly first presented abstractly in integral domains though the basics of the theory can first be more concisely introduced over abelian monoids satisfying a cancellation law. The non-zero elements of an integral domain under multiplication always have such a structure.

The development presented here derived a couple of characterisations of when factorizations into atoms (irreducible elements) exist in cancellation monoids and when in a certain way they are unique. It was based closely on that of Jacobson (Jac74, pp114-120). At the end the fundamental theorem of arithmetic was derived as a corollary of one of the main theorems.

6.1. Basic Definitions

The basic definitions for divisibility theory over an abelian monoid `g` were:

```
a ∼ b == a | b ∧ b | a

b | a  == ∃c:|g|. a = b * c

unit(u) == u | 1

a p| b == a | b ∧ ¬(b | a)
```

Here, | is the *divides* relation, ∼ is the *associate* relation and p| is the *properly divides* relation. All these relations take a suppressed argument g. Basic properties were derived about these relations including that the divides relation is a preorder and the associate relation an equivalence relation.

All factorization theory over monoids is modulo associates: the basic predicates and functions respect the associate relation ∼ and predicates concerning equality lift to predicates concerning associates. For example:

```
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)
```

Definitions were introduced for reducibility, atomicity (irreducibility) and primeness. It was convenient to define both types and predicates for atomicity and primeness.

```
IsPrime(a)
== ¬unit(a)
   ∧ (∀b,c:|g|.
        a | b * c ⇒ a | b ∨ a | c )

Prime == {x:|g|| IsPrime(x)}

Reducible(a)
== ∃b,c:|g|. ¬unit(b) ∧ ¬unit(c) ∧ a = b * c

Atomic(a) == ¬unit(a) ∧ ¬Reducible(a)

Atom == {a:|g|| Atomic(a)}
```

Alternate characterizations of reducibility and atomicity were

```
mreducible_elim:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ (∀a:|g|
          Reducible(a) ⟺ (∃b:|g|. ¬(unit(b)) ∧ b p| a))
```

and

```
matomic_char:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ (∀x,y:|g|.  Dec(x | y))
    ⇒ (∀a:|g|
          Atomic(a)
          ⟺ ¬(unit(a)) ∧ (∀b:|g|. b | a ⇒ b ∼ 1 ∨ b ∼ a))
```

The predicate `Dec` is for decidability. Its definition was

```
Dec(P) ==  P ∨ ¬P
```

Classically this is a tautology. Constructively, `Dec(P)` asserts the existence of a computable function that can decide whether `P` is true or false. Further, when `P` is true, it asserts that the constructive content of `P` can be computed. The constructive content of the assertion `x | y` can be seen from the definition of the divides relation to be some `c` such that `y = x * c`.

Every prime is an atom.

```
mprime_imp_matomic:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*g)
    ⇒ (∀a:|g|. Prime(a) ⇒ Atomic(a))
```

The notions of of primeness and atomicity are not in general equivalent. 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).

6.2. Greatest Common Divisors

A predicate stating that `y` is the greatest common divisor (GCD) of `a` and `b` had the expected definition:

```
  GCD(a;b;y) ==
    y | a ∧ y | b ∧ (∀z:|g|. z | a ⇒ z | b ⇒ z | y)
```

GCDs are only unique up to associates:

```
mgcd_unique:
  ∀g:GrpSig. ∀a,b,y1,y2:|g|.
    GCD(a;b;y1) ⇒ GCD(a;b;y2) ⇒ y1 ~ y2
```

Theorems involving a function for computing GCDs were stated by explicitly assuming the existence of such a function in the contexts of the theorems. For example:

```
mgcd_comm:
  ∀g:GrpSig. ∀f:|g| → |g| → |g|.
    IsGCDFun(f) ⇒ (∀a,b:|g|.  (a,b) ~ (b,a))


mgcd_assoc:
  ∀g:IAbMonoid. ∀f:|g| → |g| → |g|.
    IsGCDFun(f) ⇒ (∀a,b,c:|g|.  ((a,b),c) ~ (a,(b,c)))
```

The definition of `IsGCDFun` was

```
IsGCDFun(f) ==  ∀x,y:|g|.  GCD(x;y;f x y)
```

The notation `(a,b)` was for the application of the GCD function `f` to arguments `a` and `b`. Its definition read

```
(a ,{f} b) ==  f a b
```

and an alternate display form was usually used which suppressed the argument `f`. Other properties proven of GCD functions included:

```
mgcd_mul:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ (∀f:|g| → |g| → |g|
          IsGCDFun(f)
          ⇒ (∀a,b,c:|g|.  c * (a,b) ~ (c * a,c * b)))


mgcd_op_coatomic:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ (∀f:|g| → |g| → |g|
          IsGCDFun(f)
          ⇒ (∀a,b,c:|g|.
                (a,b) ~ 1 ⇒ (a,c) ~ 1 ⇒ (a,b * c) ~ 1))
```

If a GCD function exists then atoms and primes amount to the same thing. This follows from

```
matom_imp_prime_with_gcds:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ (∀x,y:|g|.  Dec(x | y))
    ⇒ (∀f:|g| → |g| → |g|
         IsGCDFun(f) ⇒ (∀a:|g|. Atomic(a) ⇒ IsPrime(a)))
```

6.3. Existence Theorem

It was shown that if the 'properly divides' relation is well-founded and if reducibility is decidable then every non-unit factors into atomic elements. 'Decidable' here means constructively not only that it is possible to compute whether or not any element of the monoid is reducible, but also that in the case that the element is reducible it is possible to compute two proper factors whose product is that element. The statement of this theorem was:

```
mfact_exists:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ WellFnd(|g|;x,y.x p| y )
    ⇒ (∀c:|g|. Dec(Reducible(c)))
    ⇒ (∀b:|g|. ¬unit(b) ⇒ (∃as:Atom 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 that provides a means of doing a constructive well-founded induction in the proof of the theorem.

An abbreviated listing of the proof is shown in Figure 4. The listing starts after a trivial initial step in which outermost universal quantifiers and implications were decomposed.

NUPRL sequents are displayed here with numbered hypotheses and with turnstiles ($\vdash$) separating the hypotheses and the conclusion. At each 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 listing shows only those parts of sequents that have been changed by the refinements immediately above. The contents of a suppressed part at some proof node can always be determined by following the proof path towards the proof-tree root looking for the last point in the proof when that part changed.

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(unit(c)))
    ⇒ (∀b:|g|. ∃as:Atom List. b ∼ Π as)
```

The `mfact_exists` and `mfact_exists_a` theorems had the ∀∃ structure typical of theorems with interesting computational content: read constructively, both theorems claim that given a monoid satisfying the various preconditions, and given an arbitrary element `b` of that monoid, a factorization into atomic elements can be computed. The NUPRL system is set up to be able to actually synthesize such programs.

6.4. Uniqueness Theorem

The main uniqueness theorem 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. 'Essentially' here means up to permutations and associates. The statement was

```
unique_mfact:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*)
    ⇒ (∀a,b:|g|.  Dec(a | b))
    ⇒ (∀ps,qs:Prime List.  Π ps ∼ Π qs ⇒ ps ≡ qs upto ∼)
```

Here, the notation '`ps ≡ qs upto ∼`' is for a 'permutation and associates' relation. It should be read as saying "`ps` is equal to `qs` up to permutations and associates." Its definition was:

```
as ≡ bs upto ∼ == as ≡ bs upto x,y.x ∼ y
```

where

```
as ≡ bs upto x,y.R[x; y]
== (|as| = |bs|) c∧ (∃p:Sym(|as|)
                         ∀i:ℕ|as|. R[as[(p.f i)]; bs[i]])
```

```
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. ¬(unit(b))
⊢ ∃as:Atom List. b = Π as
|
BY  Induction on b  using hyp 3.
|
5. j: |g|
6. ∀k:|g|. k p| j ⇒ ¬(unit(k)) ⇒ (∃as:Atom List. k = Π as)
7. ¬(unit(j))
⊢ ∃as:Atom 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. ¬(unit(b))
|   11. ¬(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 List
|   16. b = Π as1
|   17. as2: Atom 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 4. Abbreviated Proof of Existence of Atomic Factorizations

An abbreviated proof of the theorem is shown in Figure 5. The full proof printout is less than 3 pages long and can be found in Section A.2 of Appendix A.

```
1. g: IAbMonoid
2. Cancel(|g|;|g|;*)
3. ∀a,b:|g|.  Dec(a | b)
4. ps: Prime List
⊢ ∀qs:Prime List. Π ps ∼ Π qs ⇒ ps ≡ qs upto ∼
|
BY  List induction on  ps.
|
└→5. qs: Prime 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
  6. ps': Prime List
  7. ∀qs:Prime List. Π ps' ∼ Π qs ⇒ ps' ≡ qs upto ∼
  8. qs: Prime 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 5. Abbreviated proof of Uniqueness of Prime Factorizations

Constructively, this theorem asserts the existence of a function which when given two factorisations of some element will compute exactly how one is a permutation of the other and in addition will give the units by which each pair of associated elements differ.

6.5. Unique Factorization Monoid Existence

Previous theorems about the existence and uniqueness of factorizations were combined into the single theorem:

```
ufm_char:
  ∀g:IAbMonoid
    Cancel(|g|;|g|;*g)
    ⇒ WellFnd(|g|;x,y.x p| y in g )
    ⇒ (∀a:Atom. IsPrime(a))
    ⇒ (∀a:|g|. Dec(Reducible(a)))
    ⇒ (∀a,b:|g|.  Dec(a | b in g ))
    ⇒ IsUFM(g)
```

The definitions involved in the `IsUFM` predicate were as follows. Firstly a *uniquely satisfies up to* predicate (`uni_sat_upto`) was defined as

```
:
  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]`' should be read as "upto `r`, `a` is the unique `x` of type `T` such that `Q[x]` holds". The notation `a' [r] a` was an infix version of the second-order-variable application `r[a';a]`.

An *exists unique up to* predicate (`exists_uni_upto`) was defined in terms of the 'uniquely satisfies up to' predicate as:

```
(r)∃!x:T. Q[x] == ∃a:T. a r !x:T. Q[x]
```

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".

The `IsUFM` predicate was then defined as

```
IsUFM(g)
== ∀b:|g|. ¬(unit(b)) ⇒ (≡∼)∃!as:Atom 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".

An alternate characterization of when UFMs exist was derived from the lemma `matom_imp_prime_with_gcds` shown in Section 6.2.

Need to prove lemma and insert here

6.6. 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, the theorem:

```
IsUFM(<ℤ+,*>)
```

was established.

From this theorem a program was extracted for computing factorizations. Here are a couple of examples of execution of this program.

Generate examples and insert here

7. Discussion of Development Style

7.1. Style of Definitions and Theorems

A major choice involved the definition of the permutation relation on finite sequences. There are many alternatives. Two that were useful in other NUPRL developments (Jac95a) were a recursive function definition:

```
bpermr:
  as ≡b bs
  ==r case as of
        [] => null(bs)
        a::as' => a ∈b bs ∧b as' ≡b bs \ a
      esac
```

where the function `bs \ a` removes one occurrence of `a` from the list `bs` (if there are any occurrences at all) and a characterization

```
permr_iff_eq_counts:
  ∀s:DSet. ∀as,bs:|s| List.
    as ≡ bs ⟺ (∀x:|s|. x #∈ as = x #∈ bs)
```

where the function call `x #∈as` returns the number of occurrences of `x` in `as`.

The advantage of the first was that it permitted the computation of whether two lists were permutation equivalent. Boyer and Moore used a similar definition when they proved the fundamental theorem of arithmetic (BM79). A great advantage to them of this definition was that its recursive structure suggested good induction schemes. There are also no auxiliary constructs which require any complicated type checking with this definition. A minor disadvantage from the constructive viewpoint was that it required the equality of list elements to be decidable.

The second expresses the permutation relation in terms of a simpler permutation-invariant function. If finite sequences are used to model finite sets of multisets, this turns out to be a useful form.

The definition I chose for this development — which started with the notion of permutation functions — does seem though to be a more natural and direct from a mathematical viewpoint. This definition turned out to be more troublesome from the mechanization point of view. Firstly, the definition put greater demand on the arithmetic capabilities of NUPRL's type-checking tactic, and extensions had to be made to the tactic to assist matters (see Section 8). Secondly, there were some definite awkwardnesses dealing with functions whose domain of totalness was non-trivial. For example one function for restricting a permutation from `Sym(n)` to `Sym(n-1)` had typing lemma:

```
restrict_perm_wf:
  ∀n:ℕ. ∀p:Sym(n + 1).
    p.f n = n ⇒ restrict_perm(p;n) ∈ Sym(n)
```

Manual guidance had to be given to proofs involving this function to solve the `p.f n = n` antecedent.

Some choices were influenced by mechanical considerations: for example, the use of lists to model finite sequences rather than arrays where the lengths would be always carried around explicitly and elements would always be explicitly indexed. This desire to avoid index notation is of course common in mathematical practice; it's found in preferences for matrix notation in linear algebra for instance.

7.2. Style of Proofs

The NUPRL proofs presented in Section 6 and elsewhere demonstrate that using a sequent calculus and appropriate tactics it is possible to create proofs which are a good approximation of detailed proofs one might expect in mathematical texts: the state in intermediate steps of proofs is clear and inferences are largely ordered in a forward direction when reading down the proofs (facts lower down follow from facts higher up). Importantly though, backward inferences are also possible.

Applications of induction rules are usually most clearly presented in backward form.

By way of contrast note that in natural-deduction presentation styles (for instance used by MIZAR (Rud92)) inferences are restricted to being only in a forward direction, and that automation concerns often result in systems which have a strong preference for backward inference (NQTHM (BM88) or LEGO (Pol90) for example.

One continuing stumbling block to proof comprehension is understanding what tactics do. The tactic language is terse to ease input and the grammar is somewhat stilted because it a subset of the ML language. Hopefully the brief introduction to tactics in Section 3.7 and study of their effect should be of some help to readers following the proofs in this development. Even if natural language versions were permitted as input or could be automatically generated, there still would be difficulty: the tactics describe the operational mechanics of proofs but don't describe the higher-level motivation for proof steps.

Another problem is NUPRL tactics are still working on a finer granularity than anyone who wanted a detailed proof would be probably interested in. The solution I adopted in presenting the proofs in this paper was to shrink groups of usually between 2 and 5 tactic steps down to higher-level natural language comments. This is not just an ad-hoc solution for a paper presentation. Work is currently underway in NUPRL to support this step grouping and annotating so such abbreviated proofs would be permanently part NUPRL databases. Users viewing proofs would see the abbreviated proofs by default and could ask when desired to see not only the group of tactics that underly the comment, but also the proof subtree. This grouping could be repeated so proof creators could provide several readings of proofs at different levels of detail. Proof step grouping would also be a great aid when developing proofs because it would ease review of where a proof has got to. In recent work I've done in PVS — a system which also maintains proof trees — I would have found such a feature very useful, despite the fact that PVS tactics are often more powerful than NUPRL and make larger steps.

Sometimes when using higher-level tactics it's not clear how or why the tactic generated the subgoals it did. With NUPRL's `Auto` tactic, one can turn on a tracing facility which gives a rather verbose transcript of its actions each time it is run. A cleaner alternative exploits the ability to attach proof subtrees to each proof node: nearly any tactic can be set up so that whenever it is called, whether by a user directly or by another tactic, it creates a single proof node containing both the text of the tactic call the proof subtree created by the tactic being executed. If the lower-level tactics called by some high-level tactic such as `Auto` are set up in this way, then the actions of runs of the high-level tactic can be viewed when desired at an appropriate level of detail.

This scheme relies on tactics being able to generate textual representations (or more precisely, representations in NUPRL's term language) of themselves and their arguments. This rules out for instance applying the scheme to tactics which take functional arguments. In practice, this probably isn't a significant limitation.

Care is needed in controlling which subtrees are retained to avoid the memory requirements for theories become excessive. For example, users can avoid asking for every proof in a large development to be simultaneously expanded out to the finest level of detail.

An experimental version of this scheme is available in the NUPRL V4.2 distribution, though no use has been made of it yet in any of the NUPRL V4.2 theories.

8. Automation

Here I highlight some of the more distinctive aspects of the automation of the proofs in this development. For more information on NUPRLs tactics, consult my thesis (Jac95a) and the NUPRL V4.2 reference manual (Jac95b).

8.1. Type Checking

The use in NUPRLs type theory of an untyped programming language and of a rich set of type constructors gives users great latitude in the kinds of definitions they can introduce. In practice however users limit themselves to types and definitions which can be type-checked automatically by the type-checking tactic.

In this development I made widespread use of definitions which required their arguments to be in types which were subranges of integers. For example, consider the typing lemmas

```
txpose_perm_wf:
  ∀n:ℕ. ∀i,j:ℕn.  txpose_perm(i;j) ∈ Sym(n)


select_wf:
  ∀A:𝕌. ∀l:A List. ∀n:ℤ.  0 ≤ n ⇒ n < |l| ⇒ l[n] ∈ A
```

Also the functions in `Sym(n)` required their arguments to be in the type `ℕn`.

To automate type-checking of instances of such definitions, I had to add a procedure for solving sets of arithmetic inequalities. The particular procedure happened to be based around the 'Sup-Inf' algorithm (Sho77), though there are other competing algorithms which need investigating further. This algorithm is designed for determining the validity of sets of inequalities involving linear arithmetic expressions over the rationals or reals. It also works reasonably well in practice over the integers.

The procedure needed to be extended to deal with the arithmetic properties of non-linear and non-arithmetic functions. For example, when type-checking applications of a permutation function from `Sym(n)`, it exploited the fact that range values were in the type $\mathbb{N}$n. When type-checking the select function several lemmas were referenced such as

```
map_length:
  ∀A,B:𝕌. ∀f:A → B. ∀as:A List.  |map(f;as)| = |as|
length_append:
  ∀T:𝕌. ∀as,bs:T List.  |as @ bs| = |as| + |bs|
length_nth_tl:
  ∀A:𝕌. ∀as:A List. ∀n:{0...|as|}.
    |nth_tl(n;as)| = |as| - n
```

which gave the arithmetic properties of the list length function on common list constructor functions.

Though these extensions worked reasonably well, they significantly slowed down the type-checking tactic which was never fast to start with. Where convenient, definitions were introduced with more relaxed requirements on their arguments than one could imagine. For example in the typing lemma:

```
reject_wf:
  ∀A:𝕌. ∀l:A List. ∀n:ℤ.  l\[n] ∈ A List
```

the argument **n** is required to have type $\mathbb{Z}$ rather than $\mathbb{N}$n.

In general when working with a type theory with subtypes there is a tradeoff between creating definitions more stringent typing properties which place more demands on type-checking programs and embellishing definitions with complicating bounds-checking code on arguments. There is also the issue of whether a result to return when arguments are out-of-bounds is conveniently available.

8.2. Rewriting

NUPRL has a rewrite package for applying equational propositions contained in hypotheses and previously-proven lemmas.

Users can apply rewrite equations singly or in groups or write programs to selectively apply equation sets. For example, programs `MonNormC` and **AbMonNormC** for putting expressions over monoids and abelian monoids into normal form were frequently used.

The package builds proofs that the applications of rewrite equations are valid by exploiting its knowledge of the equality-respecting properties of the term constructors it finds surrounding expressions being rewritten. These justifications of rewrites are simplest when the rewrite equations only involve the built-in equality relation of NUPRL's type theory. However the package also supports rewriting with other user-defined equivalence relations.

This ability was exploited extensively in this development to rewrite with equations which involved the associated relation ($\sim$) and list permutation relation ($\equiv$). For example, Figure 6 shows the step of the uniqueness where the element `qs[i]` of the occurrence of list `qs` in hypothesis 11 and the conclusions was brought to the front of the list.

The lemma referred to was

```
select_reject_permr:
  ∀T:𝕌. ∀as:T List. ∀i:ℕ|as|.  ((as[i]::as\[i]) ≡(T) as)
```

```
1. g: IAbMonoid
2. Cancel(|g|;|g|;*)
3. ∀a,b:|g|.  Dec(a | b)
4. ps: Prime List
5. p: Prime
6. ps': Prime List
7. ∀qs:Prime List. Π ps' ∼ Π qs ⇒ ps' ≡ qs upto ∼
8. qs: Prime List
9. i: ℕ|qs|
10. p ∼ qs[i]
11. p * Π ps' ∼ Π qs
⊢ p::ps' ≡ qs upto ∼
|
BY (OnMCls [0;-1] (RWH
|     (IfIsC ⌜qs⌝ (RevLemmaWithC ['i',⌜i⌝] 'select_reject_permr'))) ...a)
|
11. p * Π ps' ∼ Π qs[i]::qs\[i]
⊢ p::ps' ≡ qs[i]::qs\[i] upto ∼
```

Figure 6. Step in Proof of Uniqueness of Prime Factorizations

The automatic justification for the conclusion rewrite included references to the lemmas

```
permr_massoc_weakening:
  ∀g:IAbMonoid. ∀as,bs:|g| List.  as ≡ bs ⇒ as ≡ bs upto ∼
and

permr_massoc_functionality:
  ∀g:IAbMonoid. ∀as,as',bs,bs':|g| List.
    as ≡ bs upto ∼
    ⇒ as' ≡ bs' upto ∼
    ⇒ (as ≡ as' upto ∼ ⟺ bs ≡ bs' upto ∼)
```

and for the hypothesis the lemma

```
mon_reduce_functionality_wrt_permr:
  ∀g:IAbMonoid. ∀xs,ys:|g| List.  xs ≡ ys ⇒ Π xs = Π ys
```

When the package doesn't find a functionality lemma that applies exactly (as when the conclusion was rewritten in this example) it chooses the lemma which requires the least amount of weakening of the relation being rewritten with respect to.

8.3. Relational Reasoning

NUPRL has a tactic `RelRST` which attempts to solve goals by exploiting common properties of binary relations such as reflexivity, transitivity, symmetry, linearity, antisymmetry and irreflexivity. The heart of this tactic is a routine that builds a directed graph based on the binary relations found in a sequent and finds shortest paths in the graph. Extensions allow it to cope with strict order relations and relations of differing strengths (for example, the ∼ and ∼≡ relation in this development. Several uses were made of this tactic to eliminate tedious steps of reasoning. For example, it solved the goal:

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

in the theorem `mdivides_functionality_wrt_massoc`.

This is a rather weak example. If all applications in this development are as trivial, is it worth devoting a section to this tactic?

9. Related Work

The restriction of the factorisation theorem to the naturals where it becomes the fundamental theorem of arithmetic has been proven in several systems, NQTHM (BM79, pp311–328) by Boyer and Moore, and NUPRL (How86) by Howe for example.

The NQTHM development is most similar in that a permutation relation is defined and and similar existence and uniqueness theorems are proven. The details of the expression of the definitions and theorems there was quite different because of working in a quantifier-free logic. As discussed in Section 7.1, the recursive definition they chose for the permutation relation gave strong hints to NQTHM on how to carry out inductions.

Should this next paragraph be relegated to the theory listing?

Interestingly, Boyer and Moore chose to prove the analog of the lemma `matom_imp_prime_with_gcds` using a general proof which applies in any cancellation monoid, though there exist shorter proofs (HW78, p21) which exploit the more specific property which naturals enjoy : namely that the gcd (a, b) can be expressed as a linear sum of a and b.

Howe's previous NUPRL development avoided reasoning about permutations by requiring that factorisations always be given in a normal form where distinct factors were listed in ascending order together with their exponents. Neither development had to work with reasoning about associates.

Having readable proofs is a serious concern of the designers of the MIZAR system (Rud92). MIZAR proofs are laid out in linearized natural-deduction style. As noted in Section 7.2, arguments are constrained to being presented in a forward inference style. When creating proofs the user commonly gives the text of each new hypothesis explicitly. While this can be a tedious exercise, it makes the formal input text for proofs a readable record of the proofs. A declarative proof style helps to keep the proof command language simple. Often a user states a new hypothesis and lists the previous hypotheses and lemmas it should be deduced from. MIZAR then justifies the new hypothesis using model elimination and equality reasoning procedures.

A current field of investigation is proof planning (Bun88) where automated decisions are made on the higher level structures of proofs. Here, as with earlier work in NQTHM, motivation for proof steps is available and readable high-level explanations can be more automatically generated than is possible currently in NUPRL.

One other system that explicitly maintains proof trees is PVS. Users a graphical display of the proof tree is generated while proofs progress.

For space reasons the tree only shows the proof commands at the tree nodes, users must mouse-click on a node to see the subgoal appear in an alternate window. When proofs grow large, these trees become hard to read and some grouping mechanism as discussed in this paper would be of help.

On the issue of automation it has been interesting to note the adoption in PVS of a hybrid type-checking discipline where the simpler type-checking is carried by a dedicated procedure and then more complicated typing properties are dealt with by tactics. They are then able to exploit the advantages of having subtypes including arithmetic subtypes without as severe a performance penalty.

The ISABELLE prover (Pau94) has a rewrite package capable of handling general equivalence relations, though this has been little used and it is not clear how well it works in practice when working with a mixture of different kinds of relations (one can only specify one congruence lemma per term constructor). Boyer, Moore and Kaufmann in the ACL2 prover, the successor to NQTHM, have also introduced support for equivalence relations, though it is not known how much use they have made of this to date.

10. Adequacy of Constructive Type Theory

This definitely needs more work. How about cutting it down to a few lines or leaving it out altogether?

Its not clear at all what future rôles constructive type theories will play in formal proof.

Constructive mathematics by its very nature has a host of concerns which are absent in nearly all classical mathematics. In some inherently infinitary fields of mathematics such as analysis or topology deep thought has to go into whether and how topics can be constructivised. When there is success in these fields, the new mathematics often is quite distinctive. The proof theory part of logic seems to be one of the easier to constructivize partly because of its more finitary nature. Category theory also seems to be an easier field, perhaps due to its very abstractness. Though the development in this paper isn't too different, algebra in general seems to be a very challenging field for constructive mathematics.

A particular appeal of constructive type theories is that they permit the realisation of the computational possibilities in constructive mathematics. However it seems very unlikely at present that the possibility of being able to synthesize programs in constructive type theories is going to have any practical significance in the foreseeable future, though it does seem that ideas for program synthesis can be inspired by how the synthesis works in constructive type theories.

Even with fields of mathematics with strong computational subfields such as algebra or numerical analysis, current opinion is that it is far more fruitful to keep with classical mathematics and reason about algorithms explicitly.

Other drawbacks to the pursuit of formalizing constructive as oppose to classical mathematics include that there is much much less source material that can be straightforwardly formalized, there are far fewer people who might be interested or capable of doing the work and the audience for such formalized mathematics is also going to be far far more restricted.

One response of researchers in mechanizing constructive type theories to criticism of constructivism has been to explore smooth classical extensions of their formalisms and systems. Also when looking at the major application areas of hardware and software verification where again things are reasonably finitary, constructivism doesn't seem to matter too much. The success of NQTHM with its 'computational logic' is worth bearing in mind here, though the explicit computational definitions are far easier to work with than the computations implicit in type theories.

However it seems unlikely that constructive-type-theory systems with classical extensions or that are tuned for applications work will ever be as easy to understand and work with as systems designed from the outset with classical formalisms.

Fortunately many automation concerns are orthogonal to the issue of the underlying formalism being constructive or classical so system development work in constructive-type-theory systems can have a wider impact.

11. Conclusions

This work has demonstrated how highly-readable formal derivations of significant theorems can be interactively produced. This readability is of benefit both to the developers of the mathematics and subsequent viewers.

The factors leading to this readability are quite general and could be exploited in a variety of other theorem proving systems. They include

- the use of a sequent calculus and a proof-tree data-structure. The sequent calculus supports both forward and backward inference styles and the proof-tree data-structure helps for interactively exploring proofs and generating readable proof listings.

– having tactics which flexibly support common reasoning styles (rewriting, chaining) and provide convenient automation for more tedious operations in proofs. The more novel examples in this development of this are the support for rewriting with equivalence relations and the automation of reasoning about arithmetic inequalities and general equivalence and order relations.

– the use of a structure editor and a display form mechanism to support concise editable representations of expressions.

Grouping tactics under hand created comments seems a good working solution to the problems of the granularity of the tactic steps being finer than readers might wish and the tactic language being obscure to the uninitiated. As discussed in Section 7.2, the grouping of proofs steps, both by hand and automatically, permits hypertext presentation of proofs where both developers and viewers can shift back and forth between multiple levels of detail.

This and other work has demonstrated the adequacy of Nuprl's type theory for some elementary abstract algebra. While there is room for further progress using similar type theories, it is thought that there are inherent difficulties in the constructive approach which seriously limit its usefulness for formalizing mathematics.

References

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.

Stuart F. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language.* PhD thesis, Cornell University, Ithaca, NY, 1987. TR 87-866.

Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Trans. Prog. Lang. Sys.*, 7(1):113–136, 1985.

R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

Robert S. Boyer and J. Strother Moore. *A Computational Logic.* ACM Monograph Series. Academic Press, New York, 1979.

Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook.* Academic Press, New York, 1988.

Alan Bundy. The use of explicit plans to guide inductive proofs. In *Proceedings of the Ninth International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 111–120. Springer-Verlag, 1988.

Robert Constable et al. *Implementing Mathematics with The Nuprl Development System.* Prentice-Hall, NJ, 1986.

H. B. Curry and R. Feys. *Combinatory Logic*, volume I. North-Holland, Amsterdam, 1958.

Thierry Coquand and Gerard Huet. The Calculus of Constructions. *Information and Computation*, 76:95–120, 1988.

Robert L. Constable. Constructive mathematics and automatic program writers. In *Proceedings of the IFIP Congress*, pages 229–233, Ljubljana, 1971.

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.

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.

M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

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.

Douglas J. Howe. Implementing number theory, an experiment with nuprl. In *Proceedings of the Eighth International Conference on Automated Deduction*, volume 230 of *Lecture Notes in Computer Science*, pages 404–415. Springer-Verlag, 1986.

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.

Douglas J. Howe. Reasoning about functional programs in Nuprl. *Functional Programming, Concurrency, Simulation and Automated Reasoning,* Lecture Notes in Computer Science, 1993.

G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Clarendon, fifth edition, 1978.

Nathan Jacobson. *Basic Algebra*, volume I. Freeman, 1974.

Paul B. Jackson. *Enhancing the Nuprl Proof Development System and Applying it to Computational Abstract Algebra*. PhD thesis, Cornell University, January 1995. Available as Cornell Department of Computer Science Technical Report TR95-1509, April 1995.

Paul B. Jackson. Nuprl V4.2 users guide and reference manual. Technical report, Cornell University, Department of Computer Science, 1995.

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.

Lawrence C. Paulson. *Isabelle : a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, 1994. with contributions by Tobias Nipkow.

R. Pollack. Lego user's guide. Technical report, University of Edinburgh, 1990.

P. Rudnicki. An overview of the Mizar project. In *1992 Workshop on Types for Proofs and Programs*, Bastad, 1992. Chalmers University of Technology.

Dana S. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275, Berlin, 1970. Spinger-Verlag.

Robert Shostak. On the SUP-INF Method for Proving Presburger Formulas. *JACM*, 24(4):351–360, 1977.

Appendix

A. Divisibility Theory

This appendix reproduces in full the proofs abbreviated in Figure 4 and Figure 5 of Section 6. 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|. ¬(unit(b)) ⇒ (∃as:Atom 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. ¬(unit(b))
⊢ ∃as:Atom List. b = Π as
|
BY (WFndHypInd 3 5 THENM D 0 ...a)
|
5. j: |g|
6. ∀k:|g|. k p| j ⇒ ¬(unit(k)) ⇒ (∃as:Atom List. k = Π as)
7. ¬(unit(j))
⊢ ∃as:Atom 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. ¬(unit(b))
| 11. ¬(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 List. b = Π as
16. ∃as:Atom List. c = Π as

1 BY New [`as2'] (D 16) THEN New [`as1'] (D 15)

15. as1: Atom List
16. b = Π as1
17. as2: Atom 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

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 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 List
⊢ ∀qs:Prime List. Π ps ∼ Π qs ⇒ ps ≡ qs upto ∼

BY (New [‘p’;‘ps\’’] (ListInd 4)
    THEN OnAll AbReduce ...)

  5. qs: Prime List
  6. e ∼ Π qs
  ⊢ [] ≡ qs upto ∼

1 BY D 6 THEN Thin 6  THEN FoldTop ‘munit‘ 6

  6. unit(Π qs)

1 BY MoveToConcl 6
     THEN New [‘q’;‘qs\’’] (D 5)
     THEN (D 0 ...a)
     THEN OnAll AbReduce

    6. unit(e)
    ⊢ [] ≡ [] upto ∼

1 2 BY (StrengthenRel ...)

    6. q: Prime
    7. qs’: Prime List
    8. unit(q * Π qs’)
    ⊢ [] ≡ q::qs’ upto ∼

1   BY Assert ⌜False⌝ THENM Trivial
       THEN D 6 THEN D 7

    6. q: |g|
    7. ¬(unit(q))
    8. ∀b,c:|g|.  q | b * c ⇒ q | b ∨ q | c
    9. qs’: Prime List
    10. unit(q * Π qs’)
    ⊢ False

1   BY (FLemma ‘munit_of_op‘ [10] ...)

  5. p: Prime
```

```
6. ps': Prime List
7. ∀qs:Prime List. Π ps' ∼ Π qs ⇒ ps' ≡ qs upto ∼
8. qs: Prime 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`` ...)

        ⊢ ¬(unit(p))

    1 2 BY (D 5 THEN Unhide THENM D 6 ...)

        ⊢ IsPrime(qs[i])

    1   BY (Assert ⌈qs[i] ∈ Prime⌉ 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 ...)
```