Algebraic Processors

Algebraic Processors

By
Pouya Larjani, B.Sc.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE (2007) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: Algebraic Processors

AUTHOR: Pouya Larjani, B.Sc. (University of Toronto)

SUPERVISOR: Dr. William M. Farmer

NUMBER OF PAGES: iv, 54

Abstract

Algebraic simplification is the task of reducing an algebraic expression to a simpler form without changing the meaning of the expression. Simplification is generally a difficult task and may have different meanings according to what the subject considers as "simple". This thesis starts off by reverse-engineering the concept of algebraic processors in the IMPS interactive mathematical proof system — which is responsible for handling all the algebraic simplification tasks — and discusses its algorithm and usage in detail. Then it explores the idea of algebraic processors as generic programs that can be configured for any type of algebraic structure to simplify expressions of that type by first formalizing the theory of algebraic processors of IMPS and then extending it to provide solutions for related topics. Algebraic processors can be defined for any user-defined algebra, as long as it conforms to the structure defined in this paper. The processors are defined as external units that can communicate with other mechanized mathematics systems in a trustable fashion and provide a program and a proof of correctness for any requests of simplification. Finally, some related processors such as one for simplification in partial orders and equivalence classes are outlined with some discussion of possible future expansions.

Contents

Chapter 1

Introduction

1.1 Algebraic Simplification

Algebraic simplification [4, 17] is an important part of every mechanized mathematics system responsible for manipulating algebraic expressions in order to put them in a reduced simpler form (if possible).

There are other kinds of simplifiers that can be embedded inside mechanized mathematics systems (such as one for simplifying logical formulas), but the focus of this thesis is only on simplifying algebraic expressions. An expression is a syntactic entity that can be constructed using the language of the current theory, and is sometimes called a "term"[1] in this paper.

An expression E is said to be simplified into an expression S if $E \equiv S$, i.e. E is semantically equivalent to S, and the length of S is shorter than the length of E [4]. The "length" of an expression here is simply defined as the textual length of the description of the expression; though it is important to mention that in some cases a "simpler" form of an expression is chosen in this paper that appears to be amongst expressions of equal size, but is chosen as the simplified form in order to choose a unique canonical form for equivalent terms. This will be demonstrated later on in Chapter 4.

Traditionally, algebraic simplification refers to the process of first expanding polynomials, followed by collecting like terms, and factoring. A computer algebra system that worked only with constants, polynomials, vectors and matrices over the field of some built-in number system would have implemented a simplifier as a program specialized for manipulating expressions of the given algebras and producing simplified expressions semantically equivalent to the original expressions.

[1] This is not always the case, as in some parts of the literature a term is an expression that is not of boolean type. See [1].

For example, consider a naïve system working on the algebra of polynomials over the integers, $\mathbb{Z}[x, y]$. Given the expression $(x+y)^2 - 2xy$, the system could first start with a re-write rule[2] saying that $(a+b)^2 = a^2 + b^2 + 2ab$ (expanding the polynomial where a matches with x and b matches with y), followed by another re-write rule for $a - a = 0$ (matching a with $2xy$), and finally one for $a + 0 = a$ (matching a with $x^2 + y^2$) to finally simplify the expression above to $x^2 + y^2$.

With more sophisticated algebraic systems (such as [8, 14, 18, 22]) where users are able to define their own number systems and algebras, the problem of algebraic simplification becomes substantially more difficult since the program may not have any special routines for simplifying the custom user-defined objects. In such systems, the user is required to derive their own rules for simplifying algebraic expressions and either applying them manually every time they are required or extending the simplifier so that it can take advantage of these new theorems.

Visiting another example, assume the user has now defined a new field $\mathbb{Z}_3$, the field of integers mod 3, and now is working over the polynomials $\mathbb{Z}_3[x, y]$ to simplify the expression $(x+y)^2+xy$, which simplifies to x^2+y^2+3xy by the above process. In this new field, the user realizes that there is an additional rule that can improve the simplification greatly — the fact that $3a = 0$ for any a — but the system's simplifier either has to completely ignore this new information, or to allow the user to install additional simplification rules.

As it turns out, modifying the simplifier to consider this will significantly reduce its power by making it restricted only to the field of integers mod 3 (in other cases, the simplified answer is most likely wrong [25]). The best solution at this point would be to carry the simplification rules *with* the theory, and have the simplifier consider the additional simplification rules specified with the current theory. This way the simplifier will be able to simplify expressions from any theory without having the need to be modified before each simplification request.

In this example, the theory of $\mathbb{Z}_3$ should carry the extra simplification re-write rule for $3a = 0$. In fact, the theory will need something more powerful, to reduce the numeric terms outside of the range $0, 1, 2$ to the equivalent number mod 3.

[2]In this paper, the term "re-write rule" has the traditional meaning where it is a simple syntax manipulator without any special logic and program extensions. A complete description of re-write rules and their extensions can be found in [1, Ch. 3]

1.2 Objectives of the Thesis

This thesis will explore the concept of an *algebraic processor* [8] as a unit inside a mechanized mathematics system that can perform the task of algebraic simplification of expressions over any algebra. The goal is to have a clear definition of how such a processor can be created that can accept any algebraic structure and perform algebraic simplification tasks on it according to the options and the configuration of the structure. The algebraic processor must also be able to provide a proof of correctness for the simplified expression.

1.3 Organization

In this thesis, a method of defining and using custom algebraic simplifiers will be introduced and analyzed. First, a general survey of applications of such processors is given in Section 1.4, followed by some background information and sources for the material used in the definition in Chapter 2. The algebraic processor unit of the IMPS interactive theorem proving system [8, 11] is introduced and analyzed in Chapter 3 and compared with some other well-known mechanized mathematics systems. Chapter 4 contains the full description, structure, and algorithm of the algebraic processor — which will expand on the idea of algebraic processors of IMPS and formalize its theory. Some further topics are briefly introduced in the final chapter that are directly related to algebraic processing.

1.4 Applications

Due to the nature of algebraic simplification, the applications of such systems are mostly limited to *computer algebra* (CA) and *automated theorem proving* (ATP)[3] systems.

The user of a computer algebra system will want to see his or her result in a simplified form after running a calculation through the system, since running a complex algebraic algorithm on data can produce a final result of many small fragments that are put together by its program. For example, if the result of a computation is $(1 + 1 + 1) * (x + x) * x$, the user will have an easier time interpreting the answer if it was presented as $6x^2$ instead. Of course, that example assumed the calculation

[3]For the purposes of this paper, "automated" theorem proving includes computer-assisted proof development, where user guidance is usually a necessary part of proof discovery. These systems are also called *mechanized theorem proving* systems where it is implied that the system does not do much proof exploration on its own.

was performed over the field of integers (or the reals or other similar fields); if the calculation was done over $\mathbb{Z}_3$ (like the past example), then the result of 0 would be much more welcomed by the user.

Automated theorem provers can also benefit from this processor to prove certain identities. For example, equality or order checking can be performed by simplifying the two sides of an equality or inequality and comparing the simplified forms.

Chapter 2

Background

2.1 IMPS Algebraic Processor

IMPS [8, 11] is a mathematical proof system developed in the early 1990s at The MITRE Corporation. The idea of an algebraic processor was first introduced in IMPS for its expression simplification system. This paper builds upon the notion of algebraic processor in IMPS and formalizes its theory and structure.

The implementation present in IMPS is discussed in detail in Chapter 3.

2.2 Transformers

In order to perform algebraic manipulation, it is first required to establish a notion of symbolic computation on the expressions with enough machinery and information to be able to assign a meaning to each one of the transformations. In this paper, transformers [12] are used as the means of symbolic computation, and together with the appropriate semantics (as discussed in Chapter 4) they will define the basis for algebraic computation.

A *transformer* is a type of function that is an alternative to a re-write rule in symbolic computation. Transformers map expressions to expressions in such a way that the transformation is specified by either an algorithm or a formula. *Axiomatic transformers* are specified by an axiom or theorem, and are used for supplying proofs of transformations. *Algorithmic transformers* are simply programs that manipulate an expression without necessarily any notion of semantics.

Definition: Let T be an axiomatic theory. A *proper* (algorithm) *transformer* is an algorithmic transformer in T linked by a meaning formula to one or more axiomatic transformers in T.

The notion of an algebraic processor in this paper will be built on the idea of proper transformers since they can carry both an axiomatic proof of their correctness and an algorithm for performing the transformation.

Transformers are more powerful than standard re-write rules, as they are able to represent any symbolic manipulation. For the example given in the introduction chapter, it was mentioned that implementing the simplifier to replace numbers with their equivalent number modulo 3 was a rather difficult task to accomplish through basic re-write rules. For a transformer, this is a very basic task that can be accomplished using the following transformer:

Axiomatic: $A(a) = a' \quad \text{if} \quad 0 \leq a' < 3 \ \wedge \ a \equiv a' \pmod 3$

Algorithmic (in English): "Given number a, let b be the representation of a in base 3. Let a' be the least significant digit of b, then replace a with a' in this context."

The axiomatic aspect of the transformer is vital for automated theorem provers in order to provide proper proof for the transformation; the algorithmic aspect of the transformer is similar to the way computer algebra systems generally perform algebraic simplification.

By combining the deduction and computation powers of transformers, a powerful framework for algebraic processing can be designed that can handle many algebraic structures with ease, as well as provide satisfactory results for both types of mechanized mathematics systems.

2.2.1 Algorithmic Transformers

Let T be an axiomatic theory $T = (L, \Gamma)$, where L is a formal language defined over a logic K, and Γ is a set of formulas of L.

An *algorithmic transformer* τ is a function that maps expressions of L to expressions of L. τ preserves semantics if, for every expression A of T, $T \models A \equiv \tau(A)$, i.e. τ preserves the meaning of A in every model of T.

Algorithmic transformers can be automatically generated from oriented equations. When the input of the transformer matches the left side of an equation, the transformer can replace it with the right side, which means that an automatically generated algorithmic transformer can be just a usual re-write rule. A detailed description of a matching algorithm can be found in [1, Ch. 4]. This shows so far that utilizing algorithmic transformers is at least as powerful as the method of using re-write rules. The advantage to using algorithmic transformers here is that they are allowed to have any kind of algorithm — not just the ones generated from equations — thus making it possible to program more complex machinery for the transformation task.

An example is given later in this section.

2.2.2 Axiomatic Transformers and Proof Generation

Although axiomatic transformers are not directly used by the algebraic processor, their existence is crucial in generating a proof of correctness for the final outcome.

Let T be the same axiomatic theory as above, an *axiomatic transformer* τ is a function on the formulas of L. An axiomatic transformer is *sound* if, for every formula F of T, $T \vdash (F \iff \tau(F))$,[1] i.e. there exists a proof in T that $F \iff \tau(F)$. Let this proof be called π; it will be used later on in conjunction with the algorithmic transformer automatically generated from this theorem.

2.2.3 Other Types of Transformers

Deductive and computational transformers (called axiomatic and algorithmic here, respectively) are not the only types of transformers. This paper, however, uses only the concepts of these two transformers. For information on other transformer types, refer to [12].

2.2.4 Re-write Rules vs. Transformers: An example

Revisiting the example in the introduction section again, a re-write rule for $2a+a = 3a$ was used to perform term collection. In the case of a general simplification algorithm for collecting like terms, re-write rules can not provide enough flexibility to implement this with a convenient number of rules, since one needs to define a system of rules for every combination of coefficients and operations (as it was demonstrated in Section 1.1). A re-write rule specifying $xa + ya = (x + y)a$ is able to collect such terms in a finite number of iterations, but it will return $(2 + 1)a$ as the result instead of the more simplified form $3a$.

A transformer, being a program that manipulates expressions, can achieve this result quite easily. In fact, the task for collecting like terms can be generalized further into creating *tally charts* that are able to do so for any binary operation. A tally chart simply stores the results of all the simplifications of direct sub-terms in one map, and finally groups all the terms together according to their operations[2] before sending the

[1] Although the symbols $\models$ (models), $\equiv$ (preserves meaning) from the last definition, and $\vdash$ (proves) in this definition are all in the meta-theory, the symbol $\iff$ (if and only if) here must be defined in the logic K and present in language L of the theory T.

[2] Indeed, commutativity and associativity of the objects is a major consideration when doing so; this is discussed in detail in the next chapter.

groups to the simplifier for possible term collection and contraction.

For the example above, the transformer responsible for collecting + terms will create a tally chart for all the occurrences of the term a, and in the end make a simplified expression with a summation of all the factors for a. For example, the term $2a + 3a + ba$ is transformed to $(2 + 3 + b)a$ by this transformer — which can then be transformed to $(5 + b)a$ by another transformer responsible for simplifying numeric expressions.

The same tally chart can be used in another transformer for collecting $*$ terms, or any other operation of similar nature.

Chapter 3

The IMPS Algebraic Processor

The IMPS interactive mathematical proof system [8, 11] contains a component named the Algebraic Processor which is responsible for a significant portion of the simplification performed by the system on algebraic expressions. The IMPS algebraic processor handles the installation and execution of the instances of algebraic processor, which are user-defined additions to the simplifier that describe the structure of an algebra and the possible simplifications that can be performed on the terms belonging to it. Every algebra used in the theory library (i.e. reals, vector spaces, matrices, etc.) requires an instance of the algebraic processor for proper simplification, as the system's simplifier relies on the available algebraic processor for each type to define how an expression is simplified.

This chapter first introduces some historic notes on the design and implementation of the IMPS algebraic processors, followed by explanation of how they function within the system. Then the syntax for defining an algebraic processor in IMPS is explained, as well some sample processors for most common algebras. At the end, a short comparison between the IMPS algebraic processor and the methods of simplification in other computerized mathematics systems is given.

3.1 History

The IMPS algebraic processor unit was implemented in the early 1990s by F. Javier Thayer at The MITRE Corporation [8]. The design and implementation of the algebraic processor was completed a decade before the work on this thesis started and little technical documentation had been available prior to this. The limited documentation that is available is found in [9, Chapter 13.4], [8, Section 4.4.1], and [10, Section 3]. The first task in this research was to reverse engineer the IMPS algebraic processor

in order to be able to explain how it works, and then compare how it performs in contrast with the simplification routines available in other mechanized mathematics systems to demonstrate its usefulness and versatility. Later in the next chapter this theory will be formalized mathematically, and other improvements and expansions will be discussed.

The implementation of the IMPS system is written in the T programming language [20, 21] which is a Scheme-based Lisp dialect. Although the T language is not in use anymore, the T code is still present in IMPS and runs in a modified Common Lisp environment. This environment is a partial emulator for the T programming language written in Common Lisp — it is not a complete implementation of T, but it emulates enough of T to execute the IMPS code.

The original idea behind the development of the IMPS algebraic processor was to design a simplifier that can simplify expressions from any kind of user-defined algebra in the theory library of the automated theorem prover. The simplifier must be configurable enough to adjust its operations according to the structure of the defined type. It must also ensure that the applied simplifications are correct in the context of the expression and are provable in the current theory. Although the implementation of the algebraic processors in IMPS is able to simplify any algebraic structure from a monoid to a field, most of the optimization and simplification techniques are ring theory oriented and in general do not apply to simpler structures.

3.2 How It Works

As mentioned earlier, the IMPS simplifier requests a separate algebraic processor to be created for each user-defined algebra. If a certain type does not have an algebraic processor associated with it, then the simplifier will be very limited in its operations and will essentially not be able to perform proper simplification.

The algebraic processor contains a mapping between each user-defined algebra and an internal abstract algebraic theory. The structures that the processor can simplify range from a general monoid structure up to specialized fields. The complete structure of these objects is described in the next chapter.

3.2.1 Sorts and Operators

In general, there are three sorts[1] associated with each algebraic processor, but only one of them is mandatory to define. The *base sort* is the sort of the current algebra being processed. The *coefficient sort* is the sort for the coefficients (scalars) that may be scaling the objects of the base sort. In order to have coefficients for the processor, the base algebraic structure must be at least a module[2] (in the hierarchy described in Section 4.2), while the ring structure has the same sort for both the coefficients and the base. The *exponent sort* is the sort for exponents of the base object. These exponents generally denote repeated multiplication presented by a semi-ring. Although the requirements for the presence of an exponent sort are very limited, the algebraic processor can perform some very useful simplifications when the requirements are met — for example, the algebra of real numbers can benefit from simplification of exponents.

In the term cx^n; x is of the base sort, c is of the coefficient sort, and n is of the exponent sort.

The operations supported by the algebraic processor are listed as follow. It is important to note that not all of these operations may be present in a given processor.

- $+$ for the addition operation (monoid operator) of type $\mathsf{base} \times \mathsf{base} \rightarrow \mathsf{base}$
- $*$ for the (scalar) multiplication operation (scalar multiplication for modules, and ring multiplication for rings) of type $\mathsf{coefficient} \times \mathsf{base} \rightarrow \mathsf{base}$
- $-$ for the group additive inverse operation of type $\mathsf{base} \rightarrow \mathsf{base}$
- ^ for the exponentiation operation of type $\mathsf{base} \times \mathsf{exponent} \rightarrow \mathsf{base}$
- / for the ring division operation of type $\mathsf{base} \times \mathsf{base} \rightarrow \mathsf{base}$
- *sub* for the group subtraction operation (composition of addition and additive inverse) of type $\mathsf{base} \times \mathsf{base} \rightarrow \mathsf{base}$

In addition to the operations defined above, there are certain constants that are required to exist in the three sorts defined for the algebraic processor, depending on which operations above were defined:

- An additive identity for the monoid (the zero element of the base sort).

[1] There has been much discussion on what is called a *sort* and what is a *type*. In this thesis the two terms are equivalent.

[2] Since "module" can be an ambiguous term, the term itself will always mean module in the mathematical sense, and when the other meaning of module as a programming unit is needed, the term "unit" is used in this paper in order to avoid the confusion.

- Additive and multiplicative identities of the ring of scalars, if there is a scalar type defined (the zero and one elements of the coefficient sort).
- The multiplicative identity in case of the presence of a division operation (the one element of the base sort, which is now a division ring).
- The zero and one elements of the exponents sort, if an exponentiation operation is defined.

3.3 Syntax and Definition

IMPS algebraic processors are defined by special "def-forms" (to be explained shortly) for each theory; the processors have a language including certain sorts assigned to them, and are installed directly into the simplifier as explained earlier. This, of course, is installed inside the current working theory only and is not considered in simplification when working in a different theory. In addition to the language and base sort, the user must provide the symbols corresponding to each operation inside the processor, and a set of options to configure the processor for the algebraic structure that is being described, such as commutativity, existence of division, and exponents. These optional configurations provide the processor the necessary information to refine its internal program for this structure.

A *def-form* [9, Ch. 4.2] in IMPS is a special construct which creates or modifies an IMPS object when evaluated. There are many different def-forms in IMPS, and each one deals with a different type of object. The specification for DEF-ALGEBRAIC-PROCESSOR is discussed here. For more details on def-forms and their specification, refer to [9].

3.3.1 Syntax of def-algebraic-processor

This is the general form for defining an algebraic processor in IMPS, with all the required and optional arguments (See [9, Ch. 17.1]):

```
(def-algebraic-processor
    *processor-name*                        ; required
    cancellative                            ; optional
    (language *language-name*)              ; required
    (base (                                 ; required
        (scalars *numerical-type*)          ; optional
        (operations *operations-pairs*)     ; required
```

```
        use-numerals-for-ground-terms       ; optional
        commutes                            ; optional
    ))
    (coefficient *coefficient-processor*)   ; optional
    (exponent *exponent-processor*)         ; optional
)
```

- `*processor-name*` is the name of the algebraic processor being currently defined. The name of the processor must be unique in the theory, and algebraic processors are paired up with their language and base sort in order for the simplifier to find the proper algebraic processor when running the simplification.
- `cancellative` is a modifier flag that tells the processor the laws of cancellation (Section 3.2.4) hold true for this processor.
- `language *language-name*` is a mandatory argument that tells the algebraic processor the language (in the current theory) for this processor. The language must define all the sorts for base, coefficients, and exponents, in addition to all the operations that are being used by this processor.
- `base` is a special form inside the definition that contains the structure and instructions for simplification of the base sort in the processor. Every algebraic processor contains at least a base sort, so this argument is mandatory.
  - `scalars *numerical-type*` is the internal numerical type for representing the numerals in this sort. The name "scalars" should not be confused with the coefficients type of the processor[3], as this is the object type inside Common Lisp that is used for simplifying arithmetic expressions as explained earlier. This argument is only needed when the user wants the processor to perform such simplification for numerals.
  - `operations *operations-pairs*` is a list of operation definitions, each one being a pair of a symbol of an operation type (such as the symbols "+" and "∗") defined in the algebraic processor's structure, and an actual operator defined in the processor's theory. This is where the algebraic processor maps an actual operation to the symbol that it knows internally. Of course, this matched constant must be of the correct type for the operation (as per Section 3.2.1), and it is verified during the soundness check. The soundness check is explained in Section 3.3.2.

[3]Although in most mathematical literature the two terms are equivalent, they have different meanings here in the IMPS algebraic processors due to a naming accident.

  - `use-numerals-for-ground-terms` is an optional modifier only to be used in conjunction with the `scalars` argument, and tells the algebraic processor that it is allowed to use the numerals returned from the Lisp computation engine in the returned expression to the user. The numerals from the arithmetic library are of the type defined in the scalars `numerical-type` section above. The actual process of this replacement and the advantages of doing so is explained later in Section 3.4.2.
  - `commutes` is another optional modifier for the processor, specifying that the ring multiplication operation is commutative (as per Section 4.2.6). The commutativity of this operator is verified during the soundness check if this flag is set.

- `coefficient`, similar to the `base` form, contains the instructions for the coefficients sort of the processor. Note that this definition is optional, but its inclusion in the processor guarantees at least a module structure on the processor (which needs to be verified as well). The coefficient processor can be defined with only reference to name of another algebraic processor, in order to avoid repetition of definitions. By default, if a coefficient sort is expected but none is supplied with the processor definition, it is assumed that the coefficient processor is same as the base processor.

- `exponent`, similar to the `coefficient` form, defines the structure of the exponents in the current processor. Again, it is implicitly assumed that the exponent processor is same as the base processor in case that one is expected but not defined (if it is not expected, meaning that no exponentiation operation was defined, then this assumption can be safely ignored since this processor is never referenced). The exponent processor can simply be a reference to another processor.

3.3.2 Verification and Installation

The structure of the algebra is determined through the defined sorts and operations supplied to the algebraic processor. A complete list of the axioms required for each structure is listed in the next chapter, but it is important to mention that the original implementation of algebraic processors in IMPS did not contain the verification routines for a few of the axioms which are necessary for completing the structure of some of these algebras.

The first step of the verification process for installing a new algebraic processor

is to confirm that the sorts and constants as listed in Section 3.2.1 are defined in the theory, and that the provided operations are indeed of the correct function type. Additionally, the system must verify that the coefficients and the base are the same sort if the structure is considered to be a ring (i.e. the coefficient sort is same as the base sort, and scalar multiplication is the ring multiplication).

The final step in the verification is to ensure that the current theory has a proof for every theorem that is used in the construction of simplification transformers. The transformers are determined through the structure of the current algebra — for example, if the specification implies that the current algebra is a ring, all the axioms of ring theory are verified by the installation process. In order for the system to verify these, each axiom of the structure must be present as a theorem in the current theory and contain a proof of correctness.

When the algebraic processor has been successfully verified, it is installed into the simplifier in a mapping between the set of base types to the algebraic processor instances responsible for simplifying expressions of the types, and the processor is called when simplification of each type is needed.

3.4 Simplification Algorithm

The simplification algorithm of the IMPS algebraic processor can be broken down into three separate sections: Term re-writing, arithmetic simplification, and finally term collection and ordering.

3.4.1 Term re-writing

When an expression is supplied to the algebraic processor for simplification, the first task is to consider which transformations are applicable on the expression. These transformations are re-write rules (sometimes conditional re-write rules [1, 2]) that are automatically generated from the axioms governing the structure of the current algebra. These axioms are listed in Section 4.2. It is important to mention that there is a selection process in the code for which rules are allowed for automatic application, since some of these rules may not lead to termination [23]. An example of such a rule is the commutativity axiom for addition.

In essence, this mechanism is similar to the method of simplification through term re-writing that is done in a basic computer algebra system. The difference is that in this method, the re-write rules are created from the structure of the algebra that the user has defined, thus making it a flexible simplifier that can be configured to only

consider the appropriate re-write rules for the user-defined algebra. For example, when a + operator is defined, the processor creates the appropriate rules for the additive identity and associativity as defined in Section 4.2.1 — this is because the smallest structure that the algebraic processor knows about which has a + operator is a monoid. Refer to Section 4.2 for the list of known algebraic structures in the algebraic processor.

One important aspect of the term re-writing component is that some of the rules are *conditional* [1, Ch. 11.3]. These rules carry a condition with them that must be true in the local context of the term before the re-writing can occur. For example, while working on the field of real numbers, consider the following rule of exponentiation: $x^m * x^n = x^{m+n}$ when x^m, x^n, and x^{m+n} are defined. If the user does not pay attention to the conditions of this rule, one can easily arrive at false conclusions such as: $(-1)^{(1/2)} * (-1)^{(-1/2)} = (-1)^0$ and further on simplify $(-1)^0 = 1$ through a different rule. The definedness constraints on this re-write rule guard against making such a wrong simplification since the sub-terms $(-1)^{(1/2)}$ and $(-1)^{(-1/2)}$ are undefined. The exponentiation operator for real numbers (or integers, or even complex numbers) is only defined when the exponent is an integer[4]; furthermore, it is defined for negative exponents only if a ring division operation exists (which does indeed exist for the case of real numbers).

3.4.2 Arithmetic Simplification

Another simplification that happens internally with the IMPS algebraic processor is the task of simplifying integer arithmetic. This task is closely tied with some of the options on the algebraic processor that were defined in Section 3.3. One of these options is to replace IMPS numerals with Lisp numerals during internal calculations, and another one of the options is to use the numerals in the ground terms. A numeral is a numeric constant such as the number 3, and a ground term is a closed term (meaning there are no variables) consisting of constants and operations of the algebra, such as $1 + 1 + 1$ in this case.

This process of utilizing numerals in ground terms involves using the Common Lisp implementation of the *bignum* arithmetic library to perform the calculations on numerals. For example, a calculation such as $(1 + 1 + 1) + (1 + 1)$ (where 1 is the ring unit for base sort) which consists of an addition between two ground terms, is

[4]The reader might have noticed at this point that the exponentiation operator is only for repeated multiplication, and must not be confused with the *exp* function. The re-write rule defined here is certainly not true for the case of *exp* function, and the soundness check would have failed prior to installation of the algebraic processor since such a theorem could not be proved.

replaced internally by the expression "3+2" and sent to the Lisp engine which returns a numeral 5. The definition of the algebraic processor would have needed to describe to the system that this operation is permitted, and define the object type (in Lisp) for the numerals to be replaced internally, as well as telling the simplifier that the + for this processor is indeed the same operator that is used in the internal numeric library.

Additionally, the algebraic processor needs to know if it is allowed to return the number 5 to the user, or if it needs to translate this back into a term (1+1+1+1+1) before returning the final result. This option is set by the
`use-numerals-for-ground-terms` flag in the definition of the algebraic processor.

Although the theorem prover itself is unable to supply an explicit proof of this operation to the algebraic processor, one must trust that their computer is able to perform basic integer arithmetic correctly.

3.4.3 Term Collection and Ordering

Collection of similar terms is another important simplification task that the IMPS algebraic processor performs. As mentioned earlier in this section, some of the axioms of the algebras are not automatically instantiated as re-write rules; however, these axioms are essential to the process of some more complicated transformations. There are two more transformations that the IMPS algebraic processor performs when simplifying a term that are not directly obtained through the re-write rules: Term collection, and term ordering.

As discussed in Section 2.2.4, a tally chart is responsible for collecting the terms that utilize the same kind of operation and simplifying them. Collection of like-terms requires some theorems to be defined and proved in the theory, namely the commutativity, associativity, and distributivity theorems for the involved operations. When simplifying a term such as $2x + y + 3x$, the processor has to ensure that the + operation is commutative and associative before collecting the two terms involving x in them. Once the term is transformed into $2x + 3x + y$ through this, then the distributivity theorem is required (to be applied in reverse) to give the result of $(2 + 3)x + y$ by the term collector (which can then be further simplified into $5x + y$ by the numerical calculator explained above). The tally chart mechanism of the IMPS algebraic processor provides an elegant method for term collection that can be proved directly using the axioms of the current theory.

Term ordering is another simple transformation performed through applications of associativity and commutativity of the + and $*$ operations of the algebra. If one of the

operators exists, and the two relevant theorems for associativity and commutativity of that operation have been proved, then the terms involving that operation can be ordered. Ordering of the terms provides a unique syntax for representing equivalent terms that are made of the same element but represented in different orders. For example, the terms $x + z + y$ and $y + z + x$ can both be ordered [1] (provided that the two necessary axioms hold) to $x + y + z$.

Although this is not "simplification" in the manner discussed earlier (i.e. the length of the term is not shortened), it is still considered an important step in the simplification task as it will lead to construction of canonical forms. These canonical forms can later be used for simplifying equalities since the equivalent terms are syntactically equal after the simplification.

3.5 Sample Processors

IMPS' implementation of algebraic processors requires the user to map every operator defined in the structure to a user-defined one. The syntax makes it mandatory to provide a name and options for the processor first, followed by definitions for the language, base type, coefficient type, and the exponent type. If either the coefficient or exponent types are not defined, but the related operators for them are defined in the base type definition, it is assumed that they are the same type as the base objects.

3.5.1 Real Numbers Algebraic Processor

This is the algebraic processor for the IMPS representation of real numbers; where the base type, coefficient type, and exponent type are all the same type of reals[5]. The operations for this processor all have their usual meaning in the language of *numerical-structures*, meaning that the + constant in this language is exactly what we need for adding real numbers, and same case with the other operations.

This processor also takes advantage of the fact that any numerals present in a calculation with real numbers will have to be rational numbers. Any irrational number has to be expressed through either the result of a function application (for example, $\sqrt{2}$), or a constant symbol (such as π). Knowing this, the processor is safe to replace numerals with the built-in rationals type of the system, and perform simplification using the *bignum* library as a quotient of two integers.

```
(def-algebraic-processor rr-algebraic-processor
```

[5]The operations are partial functions, since for some cases — specially in exponentiation — the result is undefined in the current language.

```
  cancellative
  (language numerical-structures)
  (base ((scalars *rational-type*)
     (operations
      (+ +)
      (* *)
      (- -)
      (^ ^)
      (/ /)
      (sub sub))
     use-numerals-for-ground-terms
     commutes)))
```

3.5.2 Vector Spaces Algebraic Processor

This sample is an algebraic processor for the general theory of vector spaces. Vector spaces are different than the example defined earlier because the base type and coefficients are not the same object (and thus it is not a ring processor like the algebraic processor for the reals), and only a few operations make sense in the case of vectors. These include vector addition, scalar vector multiplication, and (additive) vector inverses.

The type of the coefficients, however, is handled by the processor for real numbers as defined above. This sample shows how more complex processors can be built upon other ones for special object types.

Notice that this processor is not cancellative or commutative anymore, since such options are not true for vectors, and replacing numerals with ground terms does not make any sense in this situation.

```
(def-algebraic-processor vector-space-algebraic-processor
    (language vector-spaces-over-rr)
    (base ((operations
          (+ ++)
          (* **)
          (sub sub_vv)
          (zero v0))))
    (coefficient rr-algebraic-processor))
```

3.6 Comparison with Other Systems

This section provides a short description of how some of the other theorem proving systems and computer algebra systems handle algebraic simplification in contrast to IMPS algebraic processors.

3.6.1 Isabelle

The Isabelle [18, 19] generic theorem prover implements a machinery named *simplification sets* [19, Ch. 10.2]. Parts of the simplification sets in Isabelle are similar to the IMPS method.

A simplification set is made of five components. The first two components, *rewrite rules* and *congruence rules*, are analogous to the axioms of the mathematical structure in the IMPS algebraic processor; however, the user is required to list the axioms as re-write rules, as opposed to the IMPS method of automatically instantiating them according to the operations defined for the algebra. The Isabelle method allows more flexibility as the user is able to define simplifiers with any re-write rules, but also brings the inconvenience of having to enter them all for each algebra, as well as not being able to take advantage of specialized routines such as term collection. An example of implementing term ordering for a simplifier is given in [19, Ch. 10.5].

The other extended components of the IMPS algebraic processor — tally charting and arithmetic simplifier — are not easily done with the tactics in Isabelle. The three remaining components of the Isabelle simplification sets are the subgoaler, the solver, and the looper. These components are responsible for simplifying and solving the sub-goals during the simplification process. Although the IMPS simplifier has internal routines for defining the methods of processing the sub-goals and solving the side conditions, the interface for overriding the default behavior of them is not present for the end users.

3.6.2 PVS

The PVS [22] theorem prover incorporates a more traditional attitude towards expression simplification. The core simplification strategies [22, Ch. 14.2] use re-write rules for simplification of terms, but there are multiple rules for simplification to use different decision procedures. The PVS command `simplify` [22, Ch. 4.12.10] provides most of the machinery that the IMPS simplifier contains; it breaks down the simplification into the following steps: beta reduction, arithmetic, conditional, data type, boolean, and quantifier simplifications, and finally rewriting. The arithmetic

simplification section of the PVS `simplify` command is the closest component to the IMPS algebraic processors; however, the PVS method is mostly limited to simplifying the algebra of real numbers and no machinery is provided for configuring user-defined algebras.

In order to be more flexible, PVS allows the users to modify the set of automatic re-write rules [22, Ch. 4.13] that are applied during simplification, thus giving it the possibility for the user to define his or her own axioms for an algebraic data type. This is similar to the method used in the first two components of the Isabelle simplification sets, but it does not allow the user to take advantage of the permutative re-write rules the way that IMPS or Isabelle do — such as the example of the term ordering rules created in Isabelle.

3.6.3 Maple

The Maple [15] computer algebra system has a very different method of dealing with algebraic simplification. In Maple, the simplification of polynomials and rational functions is handled by a separate system that basically manipulates the symbols of the expression [15, Ch. 7] through a set of pre-defined actions, such as expansion, factorization, normalization, collection, and sorting of the terms. Unlike IMPS, Isabelle, and PVS, the computer algebra system pays no attention to the semantics of the manipulation, and can even lead to incorrect results due to its assumptions about the mathematical structure [25]; instead, it performs most of the simplification using arithmetic [15, Ch. 14]. This method is more efficient than the techniques that the previous theorem provers use (and also the methods explained in [13]). It is also more convenient for the users to perform simplification since the system does not require its users to define the means of simplification by re-write rules or defining theorems, but as discussed in [25] it can give incorrect simplifications for special cases.

Chapter 4

Algebraic Processor

This chapter formalizes the theory of algebraic processors that IMPS uses and proves some properties of it. Furthermore, some solutions to the shortcomings of the IMPS model are given. The chapter starts by giving a definition of what an algebraic processor is, followed by a description of the mathematical structures that can be simplified using algebraic processors. The algorithms for generating the program and the proof for a simplifier is given in the final section.

4.1 Definition

The problem outlined in the introduction chapter stated that the task of algebraic simplification becomes difficult in the "traditional" model when the users are allowed to define custom algebras in their theories.

The solution proposed here is to allow each algebra to carry a simplification routine with it, providing the required machinery to the system's simplifier to allow it to simplify any expression from this algebra. This customized simplification program is called an *algebraic processor*. The goal of defining the machinery for an algebraic processor is to not only produce a simplifier for every user-defined algebra, but to be able to generate a proof of correctness for the simplified expression as well.

The algebraic processor carries a bundle of theorems for simplifying different combinations and aspects of each one of the operations in the underlying algebra, as well as a set of configurations for changing the simplification mechanism in special cases. For each option that the algebraic processor supports, some related theorems must be supplied and proved in order to take advantage of the simplification option. This way the algebraic processor will have a proof (or a meta-proof, for certain operations) for every step of the simplification, and still have the ability to let the user decide on

which methods they want to use, as well as allowing an optional set of re-write rules for custom simplification programs.

The program breaks down the simplification in several steps. Each step will do a small piece of work and provide the required information (namely the proof of correctness and the simplification program) back to the system. Every step of the simplification is defined as a pair of transformers. The axiomatic transformer is automatically generated from the defined axiom [12] and the algorithmic transformer is defined as either a simple re-write rule (for simple cases, which can also be automatically generated), or a special routine to perform the transformation (such as term collection or integer arithmetic). The sequence of transformers that the algebraic processor produces provides both a proof for the end result and an algorithm for producing the simplified expression on demand.

Definition: Given a logic K and axiomatic theory $T = (L, \Gamma)$ where the logic K is typed (such as extensions of [7]), and given an algebra A over a base type α (of T), the algebraic processor is a program generator of type $\mathsf{Options} \rightarrow (\mathsf{Simp}, \mathsf{ProofGen})$ where $\mathsf{Options}$ is a collection of properties of A (much like the options supplied to the IMPS algebraic processor). Simp is a function of type $\alpha\ \mathsf{expr} \rightarrow \alpha\ \mathsf{expr}$ such that, given an expression E of type α, the result of $\mathsf{Simp}(E)$ is simpler than E as defined in Section 1.1 and [4]. $\mathsf{ProofGen}$ is a program that for a given expression E produces a proof that $E = \mathsf{Simp}(E)$. The type for $\mathsf{ProofGen}$ is $\alpha\ \mathsf{expr} \rightarrow \mathsf{Proof}$

Each algebraic theory defined in the system will include an algebraic processor for its types[1], and they can all be plugged into the simplifier on request. This means that the simplifier has virtually no built-in simplification routines, but instead uses the specialized programs for each algebra to simplify the expressions. Users can also create algebraic processors with their custom defined algebras and plug them to the simplifier as extensions. User-defined processors require a proof for every theorem used in the simplifier (these theorems with their proofs are usually installed in the theory), so that the processor can utilize the theorems in its proofs of correctness for expression simplifications.

4.1.1 Categories of Processors

The algebraic processor works with the theories for different algebraic structures; however, it is important to define the interface to the objects used in the simplification process. This common interface provides the framework for every custom algebraic processor to override the simplification procedure according to its needs.

[1] At this point, the reader can assume that we're only concerned about the types that require algebraic simplification.

The most basic processor starts with simplifying a monoid operation (+) between the elements of the domain. There is not much simplification that can be performed on a monoid other than eliminating unit elements, but given a slightly richer structure of a group, more simplification options are available – in this case cancellation of inverses. Of course, one additional option on such a structure would be commutativity that allows some re-ordering and possible further simplifications.

More interesting simplifications can be done on richer structures such as modules or rings, where a link between element addition and multiplication is established (distribution) and further simplification can be done by collecting like terms. Exponentiation is another optional operator in the structure, with axioms similar to the distribution law in order to simplify repeated multiplication.

Multiple algebraic processors can be linked to each other, thereby dividing the task of simplification to smaller terms of the same type, and when each sub-term has been simplified locally, the "outer" algebraic processor will collect the simplified subterms and put them together in the larger context. The process of simplification will continue until the entire expression has been traversed and every algebraic processor in the context has run its course. The chain of calling between algebraic processors will terminate since a different processor is only invoked when a sub-term of a different type is encountered, and there can only be a finite number of them inside a term.

4.2 Structure

This section defines the mathematical axioms of the processors outlined above, and the structure of the objects on which they can act. These mathematical objects provide a basis for the structures that the algebraic processor accepts as input. The program for simplification is able to process and simplify any objects that define or extend any of the outlined mathematical structures.

The *base sort* is the sort of the current algebra being processed. The *coefficient sort* is the sort for the coefficients (or scalars) that may be scaling the objects of the base sort. Respectively, the *exponent sort* is the sort for exponents of the base object. For example, in the term $c \cdot x^n$, x is of the base sort, c is of the coefficient sort, and n is of the exponent sort.

The equations defined in this section are instantiated as re-write rules that replace the left side of the equation with the right side (with the exception of the ones marked with a ($\star$)). The unification algorithm [1] does not match the left hand side of an equation if one of the elements is undefined, for example in the equation $0 * x = 0$ (Section 4.2.7) the replacement is not performed if x is not defined. In

certain cases additional definedness criteria are required to ensure the correctness of the replacement, for example the exponentiation operator ^ is a partial operator and is not defined for certain values, even when both arguments to the operator are well-defined. For such cases the definedness of the whole term is separately enforced by imposing extra conditions on the re-write rule.

In the following definitions, assume x, y, and z are of the base sort; c and d are of the coefficient sort; n and m are of the exponent sort.

4.2.1 Monoid Processor

In this case, no scalars or exponents are supplied, and only a monoid structure is imposed on the objects. The following axioms define the monoid processor, and most of the axioms are automatically instantiated as re-write rules (which are a form of an algorithmic transformer) by the algebraic processor machinery. Some of the axioms are not directly used as re-write rules, but are necessary to maintain the structure of the algebra; such axioms are marked with a ($\star$) in the lists below.

In the following axioms, + is the monoid operation, and 0 is the identity of it.

- $x + 0 = x$
- $0 + x = x$
- $(x + y) + z = x + (y + z)$ ($\star$)

The additive commutativity axiom can be added as an option for a commutative monoid:

- $x + y = y + x$ ($\star$)

4.2.2 Group Processor

The Group processor is an extension of the monoid processor where the additive inverse operation $-$ is introduced with the following axioms:

- $x + (-x) = 0$
- $(-x) + x = 0$

The additive commutativity axiom can also be applied here to form an Abelian group.

4.2.3 Module Processor

A ring of scalars is added to the Abelian Group Processor to impose a module structure on the objects. The scalar multiplication operator is $\cdot$; the operators $+_c$ and $*_c$ are the ring addition and multiplication operations of the coefficient sort, while 0_c and 1_c are the additive and multiplicative identities of the coefficient ring. These elements are subscripted with c here to emphasis they belong to the coefficient sort and not the base sort.

The following axioms are added for the module structure in addition to the rules for scalars being a ring and the base sort being an Abelian group:

- $0_c \cdot x = 0$
- $1_c \cdot x = x$
- $c \cdot y + c \cdot z = c \cdot (y + z)$
- $c \cdot x + d \cdot x = (c +_c d) \cdot x$
- $(c *_c d) \cdot x = c \cdot (d \cdot x)$ $(\star)$

4.2.4 Cancellative Module Processor

Cancellation axioms are added to the Module Processor:

- $x + (-1_c) \cdot x = 0$
- $c \cdot x = c \cdot y \iff x = y$ when $c \neq 0_c$
- $c \cdot x = 0 \iff c = 0_c \vee x = 0$

4.2.5 Ring Processor

When the base sort is the same as the coefficients sort on a Module Processor, the $\cdot$ and $*$ operations become the same, and a ring structure is imposed by the following axioms:

- $1 * x = x$
- $x * 1 = x$
- $(x * y) * z = z * (y * z)$ $(\star)$

4.2.6 Commutative Ring Processor

This is a Ring Processor with the additional (multiplicative) commutativity law:

- $x * y = y * x$ $(\star)$

4.2.7 Division Ring Processor

The cancellation laws can now be added on a Ring Processor to create a processor for a division ring. In addition to the Ring and Cancellative Module Processors, the following axioms are added:

- $0 * x = 0$
- $x * 0 = 0$
- $x * y = 0 \iff x = 0 \vee y = 0$
- $x * y = x * z \iff y = z$ when $x \neq 0$

4.2.8 Ring with Exponents Processor

This is a Ring Processor with added support for exponents. Exponentiation here is the task of repeated multiplication, and thus only certain values for the exponent would lead to a value, causing the exponentiation operator to be partial (meaning that it may be undefined for some elements of the domain).

The operations and elements of the exponents sort are subscripted with e to be easily distinguishable from base elements. Since exponentiation is a partial operator, the following axioms are applied *only* when all operations of the equation are defined on both sides:

- $x^{1_e} = x$
- $0^n = 0$ when $n \neq 0_e$
- $1^n = 1$
- $x^{0_e} = 1$ when $x \neq 0$
- $(x^m)^n = x^{(m*n)}$
- $x^m * x^n = x^{(m +_e n)}$
- $(x^m) * (y^m) = (x * y)^m$ when the ring is commutative

4.2.9 Division Ring with Exponents Processor

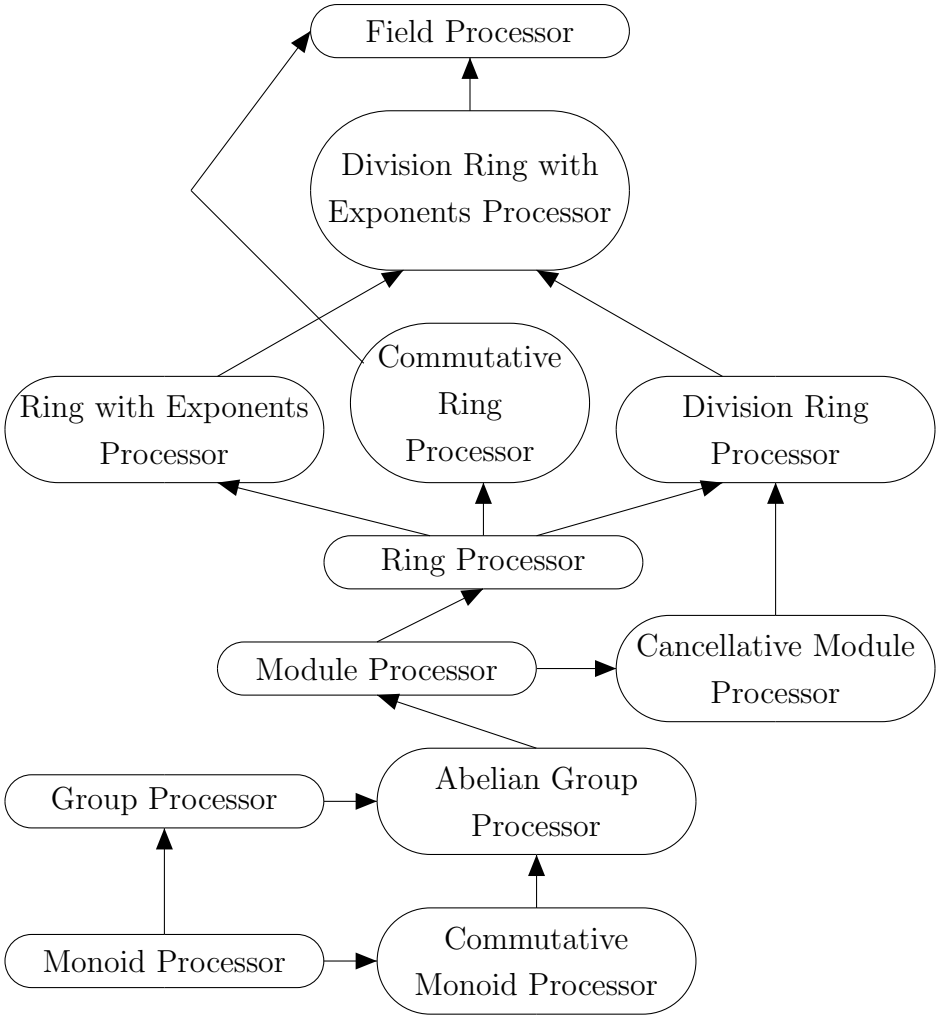
Division Ring and Ring with Exponent Processors can be combined to create a new structure. The following axiom relates division to the exponentiation, and is only applicable when both sides are defined:

- $x/y = x * (y^{-1})$

4.2.10 Field Processor

A Field Processor is essentially a combination of all the structures up until now, it is a Commutative Ring Processor which supports division and exponentiation. It can be defined by adding the multiplication commutativity axiom to the Division Ring with Exponents Processor.

4.2.11 Diagram



4.3 Implementation

Having defined the structure of the algebraic processor and the desired output of it, one may rightfully ask if it is possible to implement the algebraic processor as defined in this chapter. This section explains a method on how this processor can be implemented and presents solutions to the problems that arise during the implementation of the algebraic processor machinery.

As explained in the background material, the processor will be given in two alternative forms: an axiomatic form and an algorithmic form.

4.3.1 Overview

From a high-level point of view, the algebraic processor is broken into two components which together form a complete simplifier.

The first component is recognition and traversal of the expression. The expression is broken up into a representation of its sub-terms connected through its operators, and the processor traverses through this expression. The expression that is entered to the simplifier is very similar to a parse tree, and usually the host system decides its internal representation. However, when the algebraic processor is acting as a separate external component communicating with a different host environment, it is important to describe and establish a unique representation for every expression. This is discussed in more detail further in this chapter on the communication section.

At each stage of the traversal, an algebraic processor for the type of the term is instantiated and asked to perform its own simplification. There must be an algebraic processor defined for every type of object that the expression contains. For example, if the expression is a conditional term where the condition is a comparison between two terms of type α, and the result of the expression is of type β, then there must be two algebraic processors defined with the types α and β as their base sort respectively. The simplifier sub-system is responsible for storing these algebraic processors in an internal cache and providing a term's processor (by looking up the term type in the cache) for the simplification of the object during the traversal.

At each step, the current term's processor will first attempt to simplify the term by applying the known transformers of this type. When no further transformers are applicable, a processor can call the algebraic processor responsible for simplifying each sub-term (a breadth0first algorithm) and collect all the simplified sub-terms. This result is then passed up to the super-term which will in turn collect the rest of the simplified terms from the sibling nodes and simplify the parent term. The process will continue until the entire expression has gone through the processors responsible for each sub-term and the base expression (the largest one, at the top of the tree during the traversal) has been simplified.

The proof that this task terminates needs to be done in two parts:

First, show that only a finite number of the re-write rules (as defined as the axioms in the structure of the algebra) can be applied to each term. This can easily be checked by going through all the listed rules that are marked to be installed as re-write rules and verifying that the length of their output is shorter than the length of the input. In fact, the only rules that are marked with a ($\star$) are exactly the ones that do not provide any simplification directly, and may cause a non-terminating sequence of repeated applications of a transformer. This is not to say that these

axioms are "useless" in any way, as they are essential for the correctness of some of the transformers that will take place on the next component.

Second, show that there is a finite number of term traversals during a simplification process. Since the length of each sub-term is shorter than its parent term, by structural induction there are only a finite number of terms to be simplified through an algebraic processor. In the first part of the proof it was shown that each one of these terms has a terminating sequence of re-write rules applied during the simplification, thus concluding that this entire process terminates and returns a simplified expression.

The second component of the processor is called at each step after the simplified results of the sub-terms have returned. This component is responsible for applying transformations that are not performed by normal re-write rules, such as simplifying arithmetic expressions, collecting similar terms, and term ordering (as explained in Section 3.4). Of course, not all of these transformers may be applicable for an algebraic processor, and their presence depends on the options given to the processor. For example, collection of similar terms is only correct when distributivity, commutativity, and associativity theorems are present and proved, whereas term ordering requires commutativity and associativity (of either addition or multiplication, depending on whether it is ordering a summation term or a product) in order to be correct.

4.3.2 Transformers

There are two groups of transformers [12] that are used in the simplification process.

The first group are those transformers created from the given axioms of the processor (the ones mentioned in the previous section). The axiomatic transformer for each one simply corresponds to the theorem itself (generated from the equation), and the proof for it must be supplied by the author as an installed theorem. The algorithmic transformer of each is automatically created as a normal re-write rule from the theorem (as explained in the background section).

The second group of transformers are the simplifications performed by the final stage of the processor as defined above. These transformers require some processing since the proof for each one needs to be customized. What this means is that the proof is different for each term that is being simplified. For example, when the term ordering transformer is applied to the term $y+x$ to return the result $x+y$,[2] the proof will consist of a single application of the additive commutativity law. In contrast,

[2]Assuming the terms are ordered alphabetically, but in general this depends on the implementation of the processor

when the term $z + y + x$ is being transformed into $x + y + z$, the proof will consist of an application of the additive associativity law as well as the commutativity for it.

An easier way of dealing with the proofs of these transformers is to have a general proof of the meta-theorems defining each one of term order, term collection, and arithmetic calculations. This can be greatly beneficial for shortening the length of the generated proofs. As an example, the proofs for integer arithmetic are usually generated through applications of Peano arithmetic axioms. When multiplying two large integers this proof can become overwhelmingly large and tedious, but if the algebraic processor has an implementation for using the system's internal numeric types for fast calculation of arithmetic expressions (similar to how IMPS utilizes the *bignum* library in Lisp) and has provided a (meta-)proof of correctness for its numeric library, then it can present this proof as evidence for the correctness of the calculation without the need to go through the long (and rather uninteresting) proof of it.

4.3.3 Local Contexts of Transformers

While the algebraic processor is traversing through the structure of an expression and applying the transformers from the first group, it is also required to keep track on which sub-term the transformation is taking place. This is required in order to generate a transformer for the whole expression from the transformer that was applied locally on a smaller term. The local context of each sub-term is also important for a transformer since some of the axioms defined in the structure were conditional rules (and thus making the equivalent transformer a conditional re-write rule).

Given an expression F with a sub-term f whose position in the expression is p and a transformer τ, the context $C_{F,p}$ is a conjunction of all the terms in local context [16] of F at position p. The fundamental meta-theorem to bring the transformation to global scope is: $(C_{F,p} \implies f = \tau(f)) \implies (F = F[f/\tau(f)]_p)$
i.e. if the transformation is valid locally, then it is valid globally within F at position p.

This meta-theorem allows a local transformer to be lifted into a global transformer. For the special case of non-conditional rules, the context $C_{F,p}$ is empty and thus the resulting global transformer is independent of the position of application p.

Given a conditional re-write rule $C \implies f = \tau(f)$, the algebraic processor is required to prove that C holds in the local context of f before applying the axiomatic transformer in the proof of correctness for the simplification. For example, consider the expression F : $\mathsf{if}\ (x = 0)\ \mathsf{then}\ 1\ \mathsf{else}\ x^0$. While simplifying this expression, the conditional rule τ : $x \neq 0 \implies x^0 = 1$ is applicable. If the position of x^0 sub-term

is p in F, then $C_{F,p}$ is $x \neq 0$.

To prove that the transformation is valid in global scope, one can construct a new transformer that re-writes a larger term (being the parent of the current term, or any of its ancestors) as far high in the term construction as needed in order to provide the proper context for the transformer. For the example above, consider this new transformer: $\tau' :$ (if $(x = 0)$ then 1 else x^0) = (if $(x = 0)$ then 1 else 1)
The proof steps for transforming the base term F is:

1. $C \implies f = \tau(f)$
2. $(C_{F,p} \implies C) \implies (f = \tau(f) \implies f = \tau'(f))$
3. $F = F[f/\tau'(f)]_p$

In the proof sketch above, the first step is the proof of the transformer itself, the second step is to prove that the conditional rule can be embedded in a non-conditional rule, and finally the last step is to show that the newly obtained transformer can be applied globally. For further reading on embedding of conditional re-write rules, refer to [2, 16].

4.3.4 Generating the Proof and Program

In the final stage of the processor, the entire sequence of transformers that was collected over the previous stages is then assembled together to provide the full simplifier for this term to the system. Assume the expression E was given to the two components Simp and ProofGen of the algebraic processor. The sequence $T_1, T_2, ..., T_n$ of transformers is collected in the order they were applied during the traversal of the expression, where the axiomatic transformer is named T_i^a, and the corresponding algorithmic transformer is named T_i^p. Furthermore, assume the clause for embedding the conditional transformer (step 2 of the description in the last section) is named E_i. If the transformer T_i is not conditional, then E_i can simply be `truth`. Also assume that the global transformer (step 3 above) is named G_i. Then:
$\langle T_1^p; ...; T_n^p \rangle$ is a program that performs the simplification on request; and
$\langle T_1^a; E_1; G_1; ...; T_n^a; E_n; G_n \rangle$ is the sequence of steps to generate a proof of correctness for the simplification, with each T_i^p in first sequence is related to the triplet $(T_i^a; E_i; G_i)$ for its proof; this triplet is equivalent to step 1-3 in Section 4.3.3.

The first sequence of transformers is the generated program of the simplifier. This is the simplifier program after having optimized the simplification routines according to the mathematical types used in the expression. This program specifies the code

that should be executed on the input expression E in order to obtain the simplified output, whose correctness is proved by the prescriptive proof in the second sequence.

The second set of transformers is the output of the proof generator component. This proof is called a "prescription" as opposed to a descriptive proof since only the steps of the proof are outlined, and the proof checker needs to follow the steps one by one and apply the mentioned actions in order to obtain a complete proof.

The client may require that every step of this prescription is checked and validated before the simplified expression is accepted. Some systems may prefer the more convenient path and present the simplifier as an atomic expression (as if it were a black box) and provide the proof of this step separately upon request.

4.3.5 Communication

The algebraic processor works as a separate unit from the mechanized mathematics system; this way it can be used by both an automated theorem prover and a computer algebra system without need of change. In order for the processor to communicate properly with the host environment, a mutual communication channel must be established by both ends such that the host environment can ask the algebraic processor to process a certain expression through the channel, and the AP can respond to the host in a meaningful manner.

The communication channel must be meaning preserving [3] in such a way that the expressions on the host end of the line and the ones received on the algebraic processor side would mean the same thing in the context of their theory. Therefore it is important for both sides to agree on the underlying theory and language before performing any operations. Once the request for algebraic simplification is successfully sent to the algebraic processor, it may request certain proofs of transformers from the host environment. It is responsibility of the user to provide accurate information to the processor at its discretion. For example, a computer algebra system may not be very interested in the axiomatic proofs of every transformer[3] and would respond to the algebraic processor that every requested transformer is proved internally.

The algebraic processor would return the two combined transformers (the proof and the program) back to the host environment, and the host can now either insert the proof script of the transformation in its own proofs, or apply the simplification program to retrieve the simplified expression. It is again required for the response messages to preserve the semantics of the transformers when communicating.

[3]Some computer algebra systems such as Maple[15] and Mathematica[27] do not have any proofs for most of their simplification rules, and some of the transformations applied are not even deductively sound in certain cases.

Chapter 5

Conclusion

Before concluding this paper, a few other types of processor and some topics are worth of mentioning.

5.1 Related Processors

Algebraic processors are not the only type of processors that can aid with the task of algebraic simplification; for the purposes of mechanized mathematics systems, other types of processors can be used to handle other tasks of algebraic comparison.

There are two types of comparison processors: Equality Processors and Order Processors. The reason they are not both under the same machinery is that an order processor requires at least a partial order defined on the object types, whereas for equality a partial order is not a necessity.

An implementation of both order and equality processors is also present in the IMPS interactive mathematical proof system [9].

5.1.1 Equality Processor

Equality processors are extensions to the algebraic processors that simplify the predicates of equality over a certain type. Defining an equivalence relation on the base type in an algebraic processor allows it to be able to simplify queries of equality after performing the algebraic simplification on the two sides of the comparison. This has limited use and only simplifies (sub-)terms of boolean type, but the ability to simplify such expressions can greatly improve expressions with conditional branching in them.

The equality processor uses the entire algebraic processor machinery to simplify the two sides of equality to a normal form. When both sides of the equality are in

the most simplified form, then it is much easier to decide if the two sides are within the same equivalence relation or not.

5.1.2 Order Processor

Similar to the equality processors, *order processor* s are also extensions to the AP system. Given a partial order on the base type, the system can perform additional tasks for simplifying inequalities to improve the task of simplification.

An order processor can simplify equalities as well, but the difference is that an equality processor will always either reduce an expression to a boolean ground term, or not simplify it at all; whereas an order processor is able to reduce an inequality to a simpler one by pruning the branches of inequality that it can replace.

5.2 Other Approaches

Algebraic Processors are not the only method of providing algebraic computation in theorem proving systems; currently there is research being performed on implementing a certified computer algebra system called HOLCAS on top of HOL LIGHT [14]. For information on this system refer to [26] The HOLCAS system also provides a proof for every step of the algebraic processing, but is not limited to only simplification of expressions. This program provides the full functionality of a computer algebra system, and checks the correctness of each step of the work on the proof assistant software.

The methods used for algebraic simplification in other mechanized mathematics systems were discussed in Chapter 3.6, but none of the systems discussed are able to generate a proof of correctness for their simplification.

5.3 Conclusion

Algebraic processors have many uses in mechanized mathematics systems. The range of applications includes simplifying expressions, proving equalities, producing normal forms, and computing integer/rational arithmetic expressions. This paper also showed how a processor with such powers is able to provide a prescriptive proof of correctness for its simplification. The study of the IMPS algebraic processor shows the practicality of this approach and that it can function efficiently and correctly inside an automated theorem proving system.

Chapter 6

Acknowledgments

I have many people to thank here for their support and help through this process.

First, I would like to thank my adviser, Dr. William Farmer, for guiding me through this research, and for all the reviews and feedback on every revision of my thesis, for there were many, many of them.

Second, I thank Dr. Jacques Carette and Dr. Wolfram Kahl of my thesis committee on their helpful and detailed response on the first draft of my thesis.

Thirdly, many thanks to Javier Thayer for his great work on the design and the implementation of algebraic processors in IMPS; your program is a work of art.

Thanks to Freek Wiedijk and Cezary Kaliszyk for allowing me to read their paper on related work before its publication.

Thanks to my family for their support in my studies; if it weren't for them, I would not have been in graduate school.

And my final thanks goes to all my friends, many of whom have reviewed my writings and given me valuable feedback. I will not list the names in fear that I might miss someone, but you all know how I appreciate your help.

Appendix A — IMPS Code for Algebraic Processor

This appendix briefly demonstrates how the concepts of algebraic processing are implemented in IMPS. All the code fragments presented here can be used within a Common Lisp environment augmented with the IMPS Oolong libraries.

6.1 Soundness Check

This code checks the soundness of the algebraic processor by first creating a list of all the theorems that are required in the type of processor (as outlined in Chapter 4), and also making sure that all the required operations are present for this type of processor and are all of the correct type. Then it checks the list of theorems to verify that they are proved and installed in the current theory.

Defining the structure of an algebraic processor:

```
(define-structure-type algebraic-processor language scalars-type
    exponent-processor coefficient-processor numeral-to-term-function
    constant-recognizer-function term-to-numeral-function
    faithful-numeral-representation? -r +r *r ^r sub-r /r reduced-terms
    handled-operators commutes expand cancellation-valid?
    sum-partitioner rewrite-rules
    (((algebraic-sub-processor soi) soi)
     ((processor-validity-conditions soi)
      (algebraic-processor-validity-conditions soi))
     ((processor-reduced-terms soi)
      (algebraic-processor-reduced-terms soi))
     ((partition-summation processor expr params)
      (funcall (algebraic-processor-sum-partitioner processor)
               processor expr params))
     ((processor? soi) 'lisp:t)))

(define (operation-sorts op)
        (if op
            (make-set
```

```
            (cons (higher-sort-range (expression-sorting op))
                  (higher-sort-domains (expression-sorting op))))
        the-empty-set))
```

`algebraic-processor-validity-conditions` extracts all the required conditions for an algebraic processor:

```
(define (algebraic-processor-validity-conditions processor)
        (if (and (not (ring-processor? processor))
                 (commutative? processor))
            (imps-error
                "ALGEBRAIC-PROCESSOR-VALIDITY-CONDITIONS: commutativity is an
                 invalid declaration for a non-ring algebraic processor"))
        (if (and (not (ring-processor? processor))
                 (or (^r processor) (/r processor)))
            (imps-error
                "ALGEBRAIC-PROCESSOR-VALIDITY-CONDITIONS: algebraic operation ~A
                 is not allowed for a non-ring algebraic processor"
                (or (^r processor) (/r processor))))
        (let ((sorts+ (operation-sorts (+r processor)))
              (sorts* (operation-sorts (*r processor)))
              (sorts^ (if (^r processor)
                          (make-set
                              (list (higher-sort-range
                                     (expression-sorting
                                      (^r processor)))
                                    (car
                                     (higher-sort-domains
                                      (expression-sorting
                                       (^r processor))))))
                          the-empty-set))
              (sorts-sub (operation-sorts (sub-r processor)))
              (sorts-minus (operation-sorts (-r processor)))
              (sorts-/ (operation-sorts (/r processor))))
          (if (and (ring-processor? processor)
                   (< 1
                      (cardinality
                          (big-u (list sorts^ sorts-sub sorts-minus
                                       sorts-/ sorts+ sorts*)))))
              (imps-error
                  "ALGEBRAIC-PROCESSOR-VALIDITY-CONDITIONS: algebraic ring
                   operations have improper sortings."))
          (if (< 1
                 (cardinality
                     (big-u (list sorts+ sorts-sub sorts-minus))))
              (imps-error
                  "ALGEBRAIC-PROCESSOR-VALIDITY-CONDITIONS: algebraic operations
                   do not have identical domains and ranges")))
        (let* ((0-sort (number->scalar-constant processor 0))
               (1-sort (number->scalar-constant processor 1))
               (0-coefficient-sort
                   (number->scalar-constant
```

```
            (coefficient-processor processor) 0))
        (1-coefficient-sort
            (number->scalar-constant
                (coefficient-processor processor) 1))
        (0-exp-sort (number->exponent-constant processor 0))
        (1-exp-sort (number->exponent-constant processor 1))
        (-1-exp-sort
            (or (number->exponent-constant processor -1)
                (apply-operator
                    (-r (exponent-processor processor))
                    1-exp-sort)))
        (formulas nil)
        (sort (car (higher-sort-domains
                       (expression-sorting (+r processor)))))
        (exp-sort
            (if (^r processor)
                (cadr (higher-sort-domains
                          (expression-sorting (^r processor))))
                sort))
        (coefficient-sort
            (car (higher-sort-domains
                     (expression-sorting (*r processor)))))
        (x (find-variable 'x sort)) (y (find-variable 'y sort))
        (z (find-variable 'z sort))
        (m (find-variable 'm exp-sort))
        (n (find-variable 'n exp-sort))
        (c (find-variable 'c coefficient-sort))
        (d (find-variable 'd coefficient-sort))
        (+exp (lambda (a b)
                (apply-operator
                    (+r (exponent-processor processor)) a b)))
        (*exp (lambda (a b)
                (apply-operator
                    (*r (exponent-processor processor)) a b)))
        (*op (lambda (a b) (apply-operator (*r processor) a b)))
        (*ext-op (lambda (a b)
                   (apply-operator (*ext-r processor) a b)))
        (+ext-op (lambda (a b)
                   (apply-operator
                       (+r (coefficient-processor processor)) a
                       b)))
        (+op (lambda (a b) (apply-operator (+r processor) a b)))
        (/op (lambda (a b) (apply-operator (/r processor) a b)))
        (subop (lambda (a b)
                 (apply-operator (sub-r processor) a b)))
        (^op (lambda (a b) (apply-operator (^r processor) a b)))
        (-op (lambda (a) (apply-operator (-r processor) a))))
    (or 0-sort
        (imps-error
            "ALGEBRAIC-PROCESSOR-VALIDITY-CONDITIONS: processor has no
             zero element."))
    (push formulas
          (equality (funcall +op x y) (funcall +op y x)))
    (push formulas (equality (funcall +op x 0-sort) x))
```

```
(push formulas
      (equality (funcall +op (funcall +op x y) z)
          (funcall +op x (funcall +op y z))))
(if (*r processor)
    (block (or 1-coefficient-sort
               (imps-error
                   "ALGEBRAIC-PROCESSOR-VALIDITY-CONDITIONS: processor
                    has no multiplicative unit."))
      (if (commutative? processor)
          (push formulas
                (equality (funcall *op x y)
                    (funcall *op y x))))
      (if (processor-cancellation-valid? processor)
          (if (or (-r processor) (sub-r processor))
              (push formulas
                    (biconditional
                        (equality (funcall *op x y) 0-sort)
                        (disjunction (equality x 0-sort)
                            (equality y 0-sort))))
              (push formulas
                    (implication
                        (equality (funcall *op x y)
                            (funcall *op x z))
                        (equality y z)))))
      (if (not (-r processor))
          (push formulas
                (equality (funcall *op 0-coefficient-sort x)
                    0-sort)))
      (push formulas
            (equality (funcall *op 1-coefficient-sort x) x))
      (push formulas
            (equality (funcall *op c (funcall +op y z))
                (funcall +op (funcall *op c y)
                         (funcall *op c z))))
      (if (ring-processor? processor)
          (if (not (commutative? processor))
              (push formulas
                    (equality
                        (funcall *op (funcall +op y z) x)
                        (funcall +op (funcall *op y x)
                                 (funcall *op z x))))
              (push formulas
                    (equality
                        (funcall *op (funcall *op x y) z)
                        (funcall *op x (funcall *op y z)))))
          (if (*ext-r processor)
              (block (push formulas
                           (equality
                            (funcall *op
                             (funcall +ext-op c d) x)
                            (funcall +op (funcall *op c x)
                             (funcall *op d x))))
                (push formulas
                      (equality
```

```
                          (funcall *op (funcall *ext-op c d)
                           z)
                          (funcall *op c (funcall *op d z)))))))))
(if (^r processor)
    (block (push formulas
                 (implication
                     (defined-in
                         (funcall *op (funcall ^op x m)
                          (funcall ^op x n))
                         sort)
                     (equality
                         (funcall ^op x (funcall +exp m n))
                         (funcall *op (funcall ^op x m)
                          (funcall ^op x n)))))
      (push formulas
            (implication
                (disjunction
                    (defined-in
                        (funcall *op (funcall ^op x m)
                                 (funcall ^op y m))
                        sort)
                    (defined-in
                        (funcall ^op (funcall *op x y) m)
                        sort))
                (equality
                    (funcall *op (funcall ^op x m)
                             (funcall ^op y m))
                    (funcall ^op (funcall *op x y) m))))
      (push formulas (equality (funcall ^op x 1-exp-sort) x))
      (push formulas
            (implication
                (defined-in (funcall ^op x 0-exp-sort) sort)
                (equality (funcall ^op x 0-exp-sort) 1-sort)))
      (push formulas
            (implication
                (defined-in (funcall ^op 1-sort n) sort)
                (equality (funcall ^op 1-sort n) 1-sort)))
      (push formulas
            (implication
                (defined-in (funcall ^op 0-sort m) sort)
                (equality (funcall ^op 0-sort m) 0-sort)))
      (push formulas
            (implication
                (defined-in (funcall ^op (funcall ^op x m) n)
                    sort)
                (equality (funcall ^op (funcall ^op x m) n)
                    (funcall ^op x (funcall *exp m n)))))
      (push formulas
            (biconditional
                (conjunction
                    (defined-in (funcall ^op x m) sort)
                    (defined-in (funcall ^op x n) sort))
                (defined-in (funcall ^op (funcall ^op x m) n)
                    sort)))))
```

```
        (if (and (/r processor) (^r processor))
            (push formulas
                  (implication
                      (disjunction (is-defined (funcall /op x y))
                          (is-defined
                              (funcall *op x
                                       (funcall ^op y -1-exp-sort))))
                      (equality (funcall /op x y)
                          (funcall *op x (funcall ^op y -1-exp-sort))))))
        (if (sub-r processor)
            (if (-r processor)
                (push formulas
                      (equality (funcall subop x y)
                          (funcall +op x (funcall -op y))))
                (block (push formulas
                             (equality (funcall subop x y)
                                 (funcall +op x
                                  (funcall subop 0-sort y))))
                  (push formulas
                        (equality
                            (funcall +op x (funcall subop 0-sort x))
                            0-sort)))))
        (if (-r processor)
            (push formulas
                  (equality (funcall +op x (funcall -op x)) 0-sort)))
        (if (not (eq? processor (exponent-processor processor)))
            (set formulas
                 (append formulas
                         (processor-validity-conditions
                             (exponent-processor processor)))))
        (if (not (eq? processor (coefficient-processor processor)))
            (set formulas
                 (append formulas
                         (processor-validity-conditions
                             (coefficient-processor processor)))))
        (union (map rewrite-rule-formula
                    (algebraic-processor-rewrite-rules processor))
               formulas)))
```

`processor-sound-in-theory?` checks if all the theorems (validity conditions of the algebraic processor) are provable in the given theory:

```
(define (processor-sound-in-theory? processor theory)
        (or (memq? processor (theory-valid-processors theory))
            (let ((valid? (every? (lambda (x)
                                    (let
                                     ((thm? (theory-theorem? theory x)))
                                      (if (not thm?)
                                       (format 'lisp:t
                                        "~A fails to be a theorem.~%"
                                        x))
                                      thm?))
```

```
                          (processor-validity-conditions
                              processor))))
            (if valid?
                (set (theory-valid-processors theory)
                     (add-set-element processor
                         (theory-valid-processors theory))))
            valid?)))
```

6.2 Structure Traversal and Simplification

This is the code for traversing through an expression and checking for applicable transformers at each stage.

`algebraic-processor-apply-rewrite-rules` traverses through an expression and applies all the (conditional) re-write rules to the local context of each sub-term:

```
(define (algebraic-processor-apply-rewrite-rules processor expr params)
        (iterate loop
                 ((rules (algebraic-processor-rewrite-rules processor))
                  (expr expr))
                 (if (null? rules) expr
                     (receive (new-expr reqs ())
                              (funcall (car rules)
                                       (processor-parameters-context
                                        params)
                                       expr
                                       (processor-parameters-persistence
                                        params))
                              (set (processor-parameters-requirements
                                    params)
                                   (set-union
                                    (processor-parameters-requirements
                                     params)
                                    reqs))
                              (loop (cdr rules) new-expr)))))
```

`algebraic-processor-simplify-with-requirements` checks the local context of the given expression and applies all the simplification rules available for that context:

```
(define (algebraic-processor-simplify-with-requirements processor
            context expr persist)
        (if (and (application? expr)
                 (memq (operator expr)
                       (algebraic-processor-handled-operators
                           processor)))
```

```
            (let ((params (make-processor-parameters)))
              (set (processor-parameters-persistence params) persist)
              (set (processor-parameters-context params) context)
              (let ((simplified
                        (algebraic-processor-simplify processor expr
                            params)))
                (return
                  simplified
                  (processor-parameters-requirements params)
                  'lisp:t)))
            (return expr nil lisp:nil)))
```

algebraic-processor-simplify is the main simplification function that applies an algebraic processor to an expression as per Section 3.4.1:

```
(define (algebraic-processor-simplify processor expr params)
        (if (processor-reduced? processor expr params) expr
            (let ((expr (algebraic-processor-insistently-apply-rewrite-rules
                            processor expr params)))
              (if (application? expr)
                  (select (operator expr)
                          (((+r processor))
                           (annotate-expression-as-reduced processor
                               (simp+ processor expr params) params))
                          (((^r processor))
                           (annotate-expression-as-reduced processor
                               (if (and
                                    (algebraic-processor-expand processor)
                                    (commutative? processor))
                                   (expand^ processor expr params)
                                   (simp^ processor expr params))
                               params))
                          (((*r processor))
                           (annotate-expression-as-reduced processor
                               (if (eq? processor
                                    (coefficient-processor processor))
                                   (if
                                    (algebraic-processor-expand processor)
                                    (expand* processor expr params)
                                    (simp* processor expr params))
                                   (simp*-1 processor expr params))
                               params))
                          (((-r processor))
                           (annotate-expression-as-reduced processor
                               (simp- processor expr params) params))
                          (((sub-r processor))
                           (annotate-expression-as-reduced processor
                               (simp-sub processor expr params) params))
                          ((((/r processor))
                           (annotate-expression-as-reduced processor
                               (simp/ processor expr params) params))
                          (else (simplify-by-transforms
```

```
                                (processor-parameters-context params)
                                expr
                                (processor-parameters-persistence params))))
                (simplify-by-transforms
                    (processor-parameters-context params) expr
                    (processor-parameters-persistence params))))))
```

6.3 Special Optimizations

The special optimizations that take place are the aforementioned transformers for replacing numerals with ground terms and collecting similar terms. Tally charts are used for collecting like terms in both additive and multiplicative operations.

Definition and operations on tally charts:

```
(define-structure-type tally-chart scalar scalar-accumulator
    label-accumulator label-equivalence comparator object-list)

(define (init-tally-chart scalar-init scalar-accumulator
           label-accumulator label-equivalence comparator)
        (let ((atc (make-tally-chart)))
          (set (tally-chart-scalar atc) scalar-init)
          (set (tally-chart-scalar-accumulator atc) scalar-accumulator)
          (set (tally-chart-label-accumulator atc) label-accumulator)
          (set (tally-chart-label-equivalence atc) label-equivalence)
          (set (tally-chart-comparator atc) comparator)
          (set (tally-chart-object-list atc) nil)
          atc))

(define (accumulate-scalar atc increase)
        (set (tally-chart-scalar atc)
             (funcall (tally-chart-scalar-accumulator atc)
                      (tally-chart-scalar atc) increase)))

(define (accumulate-label atc label increase)
        (cond
          ((tally-chart-comparator atc)
           (iterate loop ((rest (tally-chart-object-list atc)))
                    (cond
                      ((null? rest)
                       (push (tally-chart-object-list atc)
                             (init-tally-object label increase)))
                      ((funcall (tally-chart-label-equivalence atc)
                                label (tally-object-label (car rest)))
                       (tally (car rest) increase
                              (tally-chart-label-accumulator atc)))
                      (else (loop (cdr rest))))))
          (else (cond
```

```
                ((and (tally-chart-object-list atc)
                      (funcall (tally-chart-label-equivalence atc)
                               label
                               (tally-object-label
                                   (car
                                    (tally-chart-object-list atc)))))
                 (tally (car (tally-chart-object-list atc)) increase
                        (tally-chart-label-accumulator atc)))
                (else (push (tally-chart-object-list atc)
                            (init-tally-object label increase)))))))

(define (label-tallies atc)
        (cond
          ((tally-chart-comparator atc)
           (sort (tally-chart-object-list atc)
                 (lambda (a b)
                   (funcall (tally-chart-comparator atc)
                            (tally-object-label a)
                            (tally-object-label b)))))
          (else (tally-chart-object-list atc))))
```

`sum-expression-list` uses tally charts above to collect all the similar terms in a summation as per Section 3.4.3:

```
(define (sum-expression-list processor expr-list params)
        (let ((chart (make-weighted-sum-tally-chart processor)))
          (walk (lambda (x)
                  (weighted-sum-accumulate-expression processor chart
                      x))
                expr-list)
          (weighted-sum-tally-chart->expression processor chart params)))

(define (weighted-sum-accumulate-expression processor ptc x)
        (let ((1num (coerce-type
                        (scalars-type
                            (coefficient-processor processor))
                        1)))
          (cond
            ((scalar-constant? processor x)
             (accumulate-scalar ptc
                 (scalar-constant->numerical-object processor x)))
            ((formal-symbol? x) (accumulate-label ptc (list x) 1num))
            ((addition? processor x)
             (walk (lambda (z)
                     (weighted-sum-accumulate-expression processor ptc
                         z))
                   (arguments x)))
            ((multiplication? processor x)
             (let ((arguments
                       (multiplicative-associative-arguments processor
                           x)))
               (cond
```

```
                ((scalar-constant? (coefficient-processor processor)
                     (car arguments))
                 (accumulate-label ptc (cdr arguments)
                     (scalar-constant->numerical-object
                         (coefficient-processor processor)
                         (car arguments))))
                (else (accumulate-label ptc arguments 1num)))))
          (else (accumulate-label ptc (list x) 1num)))))

(define (weighted-sum-tally-chart->expression processor ptc params)
        (iterate loop ((accum nil) (fc-tally-list (label-tallies ptc)))
                (cond
                  ((null? fc-tally-list)
                   (+scalar processor (tally-chart-scalar ptc)
                            (reverse accum) params))
                  ((numerical-=0?
                       (tally-object-weight (car fc-tally-list)))
                   (require-convergence-every-factor processor params
                       (tally-object-label (car fc-tally-list)))
                   (loop accum (cdr fc-tally-list)))
                  (else (let ((weighted-product
                                  (*scalar processor
                                   (tally-object-weight
                                    (car fc-tally-list))
                                   (tally-object-label
                                    (car fc-tally-list))
                                   params)))
                          (loop
                            (cons weighted-product accum)
                            (cdr fc-tally-list)))))))
```

`multiply-expression-list` uses tally charts above to collect all the similar terms in a multiplication as per Section 3.4.3:

```
(define (multiply-expression-list processor expr-list params)
        (let ((chart (make-weighted-product-tally-chart processor)))
          (walk (lambda (x)
                  (weighted-product-accumulate-expression processor
                      chart x))
                expr-list)
          (weighted-product-tally-chart->expression processor chart
              params)))

(define (make-weighted-product-tally-chart processor)
        (init-tally-chart (coerce-type (scalars-type processor) 1)
            numerical-* append! alpha-equivalent?
            (if (commutative? processor) quick-compare nil)))

(define (weighted-product-accumulate-expression processor mtc x)
        (let ((1exp (if (inhibit-exponentiation? processor) 1
                        (number->exponent-constant processor 1))))
          (cond
```

```
            ((scalar-constant? processor x)
             (accumulate-scalar mtc
                 (scalar-constant->numerical-object processor x)))
            ((formal-symbol? x) (accumulate-label mtc x (list 1exp)))
            ((multiplication? processor x)
             (walk (lambda (z)
                     (weighted-product-accumulate-expression processor
                         mtc z))
                   (arguments x)))
            ((exponentiation? processor x)
             (accumulate-label mtc (1starg x) (list (2ndarg x))))
            (else (accumulate-label mtc x (list 1exp))))))

(define (weighted-product-tally-chart->expression processor mtc params)
        (let ((sub (exponent-processor processor)))
          (iterate loop
                   ((accum nil) (be-tally-list (label-tallies mtc)))
                   (cond
                     ((null? be-tally-list)
                      (*scalar processor (tally-chart-scalar mtc) accum
                               params))
                     (else (let ((exponent-list
                                    (tally-object-weight
                                     (car be-tally-list)))
                                 (base (tally-object-label
                                        (car be-tally-list))))
                             (if (and (^r processor)
                                      (> (length exponent-list) 1))
                                 (walk (lambda (x)
                                         (require-convergence processor
                                          params
                                          (apply-operator
                                           (^r processor) base x)))
                                       exponent-list))
                             (let ((factor
                                    (if
                                     (inhibit-exponentiation?
                                      processor)
                                     (^formal-inhibiting-exponentiation
                                      processor base
                                      (apply + exponent-list) params)
                                     (^formal processor base
                                      (sum-expression-list sub
                                       exponent-list params)
                                      params))))
                               (cond
                                 ((scalar-constant-=1? processor
                                      factor)
                                  (loop accum (cdr be-tally-list)))
                                 ((scalar-constant-=0? processor
                                      factor)
                                  (require-convergence-every processor
                                      params accum)
                                  (map (lambda (x)
```

```
                                    (require-convergence processor
                                     params
                                     (tally-object-label x))
                                    (require-convergence-every
                                     processor params
                                     (tally-object-weight x)))
                                  (cdr be-tally-list))
                           factor)
                          ((scalar-constant? processor factor)
                           (accumulate-scalar mtc
                               (scalar-constant->numerical-object
                                processor factor))
                           (loop accum (cdr be-tally-list)))
                          (else (loop
                                  (cons factor accum)
                                  (cdr be-tally-list)))))))))))
```

`repeated-sum-of-ones->numeral` is a part of the transformation for arithmetic simplification in Section 3.4.2, it replaces the IMPS numerals with Lisp numerals:

```
(define (repeated-sum-of-ones->numeral processor zero unit expr)
        (iterate loop ((top 'lisp:t) (expr expr))
                 (cond
                   ((eq? expr zero)
                    (coerce-type (scalars-type processor) 0))
                   ((eq? expr unit)
                    (coerce-type (scalars-type processor) 1))
                   ((and top (sign-negation? processor expr))
                    (let ((n (loop lisp:nil (1starg expr))))
                      (if n (numerical-minus n) lisp:nil)))
                   ((and (addition? processor expr)
                         (eq? (2ndarg expr) unit))
                    (let ((n (loop lisp:nil (1starg expr))))
                      (if n (numerical-+ 1 n) lisp:nil)))
                   (else lisp:nil))))
```

`numeral->repeated-sum-of-ones` replaces the Lisp numerals back to IMPS numerals. This is the case when the algebraic processor has not allowed the option to use numerals in ground terms:

```
(define (numeral->repeated-sum-of-ones processor zero unit n)
        (or (funcall (numerical-type-recognizer
                        (scalars-type processor))
                     n)
            (imps-error "~A is not of numerical type ~A." n
                (scalars-type processor)))
        (cond
          ((numerical-=0? n) zero)
          ((numerical-=1? n) unit)
```

```
          ((numerical-< n (coerce-type (scalars-type processor) 0))
           (if (-r processor)
               (apply-operator (-r processor)
                   (numeral->repeated-sum-of-ones processor zero unit
                       (numerical-minus n)))
               lisp:nil))
          (else (apply-operator (+r processor)
                    (numeral->repeated-sum-of-ones processor zero unit
                        (numerical-+ n
                            (numerical-minus
                                (coerce-type (scalars-type processor)
                                    1))))
                    unit))))
```

use-numerals-for-ground-term is the transformation required for displaying numerals to the user as a part of the ground terms defined in section 3.4.2:

```
(define (use-numerals-for-ground-terms processor)
        (let ((language (processor-language processor)))
          (set (algebraic-processor-faithful-numeral-representation?
                   processor)
               'lisp:t)
          (set (algebraic-processor-numeral-to-term-function processor)
               (lambda (x) (find-constant language x)))
          (set (algebraic-processor-term-to-numeral-function processor)
               (lambda (x) (name x)))
          (set (algebraic-processor-constant-recognizer-function
                   processor)
               (lambda (expr)
                 (funcall (numerical-type-recognizer
                              (scalars-type processor))
                          (name expr))))
          processor))
```

Bibliography

[1] F. Baader and T. Nipkow, *Term rewriting and all that*. New York, NY, USA: Cambridge University Press, 1998.

[2] J. A. Bergstra and J. W. Klop, "Conditional rewrite rules: Confluence and termination," *J. Comput. Syst. Sci.*, vol. 32, no. 3, pp. 323–362, 1986.

[3] J. Carette, W. M. Farmer, and J. Wajs, "Trustable communication between mathematics systems," in *Calculemus 2003* (T. Hardin and R. Rioboo, eds.), (Rome, Italy), pp. 58–68, Aracne, 2003.

[4] J. Carette, "Understanding expression simplification," in *ISSAC '04: Proceedings of the 2004 International Symposium on Symbolic and Algebraic Computation*, (New York, NY, USA), pp. 72–79, ACM Press, 2004.

[5] J. Carette, "Gaussian elimination: A case study in efficient genericity with metaocaml.," *Sci. Comput. Program.*, vol. 62, no. 1, pp. 3–24, 2006.

[6] A. Chaieb and M. Wenzel, "Context aware Calculation and Deduction — Ring Equalities via Gröbner Bases in Isabelle," *LNAI — Towards Mechanized Mathematical Assistants*, vol. 4573, pp. 27–39, 2007.

[7] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.

[8] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "IMPS: An Interactive Mathematical Proof System," *Journal of Automated Reasoning*, vol. 11, pp. 213–248, 1993.

[9] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "The IMPS user's manual," Tech. Rep. M-93B138, The MITRE Corporation, 1993. Available at http://imps.mcmaster.ca/.

[10] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "Contexts in mathematical reasoning and computation," *Journal of Symbolic Computation*, vol. 19, pp. 201–216, 1995.

[11] W. M. Farmer, J. D. Guttman, and F. J. Thayer Fábrega, "IMPS: An updated system description," in *Automated Deduction—CADE-13* (M. McRobbie and J. Slaney, eds.), vol. 1104 of *Lecture Notes in Computer Science*, pp. 298–302, Springer-Verlag, 1996.

[12] W. M. Farmer and M. von Mohrenschildt, "Transformers for symbolic computation and formal deduction," in *CADE-17 Workshop on the Role of Automated Deduction in Mathematics* (S. Colton, U. Martin, and V. Sorge, eds.), pp. 36–45, 2000.

[13] K. O. Geddes, S. R. Czapor, and G. Labahn, *Algorithms for computer algebra*. Norwell, MA, USA: Kluwer Academic Publishers, 1992.

[14] J. Harrison, "HOL light: A tutorial introduction," in *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design* (M. Srivas and A. Camilleri, eds.), pp. 265–269, 1996.

[15] A. Heck, *Introduction to Maple*. New York, NY, USA: Springer-Verlag, 1995.

[16] L. G. Monk, "Inference rules using local contexts," *J. Autom. Reason.*, vol. 4, no. 4, pp. 445–462, 1988.

[17] J. Moses, "Algebraic simplification: A guide for the perplexed," in *SYMSAC '71: Proceedings of the second ACM symposium on Symbolic and algebraic manipulation*, (New York, NY, USA), pp. 282–304, ACM Press, 1971.

[18] L. C. Paulson, "Isabelle: A Generic Theorem Prover," *Lecture Notes in Computer Science 828*, 1994.

[19] L. C. Paulson, *The Isabelle Reference Manual*. University of Cambridge; Computer Laboratory, May 1997.

[20] J. A. Rees, N. I. Adams, and J. R. Meehan, *The T Manual*. Computer Science Department, Yale University, fifth ed., 1988.

[21] J. A. Rees and N. I. Adams, "T: A dialect of Lisp or, Lambda: The ultimate software tool," in *ACM Symposium on Lisp and Functional Programming*, pp. 114–122, 1982.

[22] N. Shankar, S. Owre, J. M. Rushby, and D. W. J. Stringer-Calvert, *PVS Prover Guide*. Computer Science Laboratory, SRI International, Menlo Park, CA, Sept. 1999.

[23] J. R. Slagle, "Automated Theorem-Proving for Theories with Simplifiers Commutativity, and Associativity," *J. ACM*, vol. 21, no. 4, pp. 622–642, 1974.

[24] The Coq Development Team, *The Coq Proof Assistant Reference Manual – Version 8.0*. INRIA-Rocquencourt-CNRS-ENS Lyon, June 2004. http://coq.inria.fr.

[25] M. J. Wester, "A Critique of the Mathematical Abilities of CA Systems," *Computer Algebra Systems: A Practical Guide*, pp. xvi+436, 1999. Revised on January 4, 1999 and published separately.

[26] F. Wiedijk and C. Kaliszyk, "Certified computer algebra on top of an interactive theorem prover," *LNAI — Towards Mechanized Mathematical Assistants*, vol. 4573, pp. 94–105, 2007.

[27] S. Wolfram, *The Mathematica Book*. Wolfram Media, Incorporated, 2003.