A LANGUAGE AND A LIBRARY OF ALGEBRAIC THEORY-TYPES

A LANGUAGE AND A LIBRARY OF ALGEBRAIC THEORY-TYPES

By

HUAN ZHANG, B.SC.

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

McMaster University

MASTER OF SCIENCE (2009)	McMaster Univeristy
(Computing and Software)	Hamilton, Ontario

TITLE:	A Language and a Library of Algebraic Theory-types
AUTHOR:	Huan Zhang, B.Sc. (The University of Western Ontario)
SUPERVISOR:	Dr. Jacques Carette
NUMBER OF PAGES:	ix, 78

Abstract

Many issues stand in the way of the development of contemporary *mechanized mathematics systems (MMS)* and the following are two major obstacles:

- Dedicated languages with mathematical specifications for MMS.
- A well-endowed theory library which serves as a database of mathematics.

We implement a *MathScheme Language (MSL)*, which represents *theory types* useful for covering basic algebraic structures and improving the expressive power of mathematical modeling systems. The development of MSL primarily focuses on language syntax and its logic independence.

More importantly, we present a *library* of theory types developed based on module systems of typed programming languages and algebraic specification languages. The modularity mechanism used in our library aims for the interface manipulation and high level expressivity of MMSs. The theories are organized according to the little theories method [10]. Our module system extensively supports several building operations to construct new theories from existing theories.

Acknowledgements

I would first and foremost like to express my deep-felt gratitude to my supervisor, Dr. Jacques Carette, who shared with me a lot of his expertise and research insight through my studies. This thesis was made possible by his advice, assistance and guidance.

My special thanks and appreciation goes to the members of my examination committee, Dr. Jacques Carette, Dr. William M. Farmer, and Dr. Spencer Smith.

Many helpful suggestions and comments by various members of the Math-Scheme project group have helped to improve the quality of this work.

And finally, I would like to thank my family and friends, for their love, encouragement and continuous support.

Table of Contents

List of Tables

Chapter 1

Introduction

1.1 Overview of MMS and MathScheme

The mathematics activity consists of formulating mathematical models and then exploring them by stating and proving conjectures and by performing calculations. The goal of *mechanized mathematics system (MMS)* is to produce a computer environment that is intended to support and improve rigorous mathematics activity. Today MMSs include two major types: *computer algebra systems (CAS)* and *theorem proving systems (TPS)*. The MathScheme [15] system is a *mechanized mathematics system* based on the integrated framework for *computer algebra systems* and *theorem proving systems*. CASs, such as Axiom [12], Maple [2] and Mathematica [21], provide high-level symbolic computation based on algorithmic theories, while TPSs, such as Coq [3], HOL [11], IMPS [9], Mizar [20] and PVS [18], provide low-level reasoning environments based on axiomatic theories. As an MMS, MathScheme is a computer system that aims to support and improve both aspects and provides tools for building and extending theories.

1.2 Programming and specification languages for MMS

Most programming languages and specification languages have module systems for organizing large software developments and specifications. Current MMSs involve various ways in which it intersects with programming languages and specification languages. Many systems for mathematics contain a dedicated input language and are frequently built in that same language. For instance, the Latc language in Coq, mathematical languages as in Mizar, and other languages built into computer algebra systems. Another important observation is that modeling languages used for many systems for mathematics cover both fields of functional languages and specification languages, such as a number of portions of HOL are similar to ML and Haskell and languages like CASL [16] and SPECTRAL [14] improve the expressivity of formal specification. We know that good programming and specification languages provide powerful means of expression within their domains, which will be examined in Chapter 2. The tendency in both programming language and specification language design seems to be towards providing structuring mechanisms with increasingly more power. Hence, it is our intention to implement a specialized language that can build theories as "types" and aid the detailed representation of their relationships as "interfaces". Such expressive language of "theory types" should be close to a typed functional language and contains very rich specification capabilities.

1.3 Theory library for MMS

An effective MMS needs a well organized and constructed theory library which contains sufficient mathematical knowledge to support mathematical activities. Unfortunately, very few MMSs have a sophisticated theory library supported by a good module system and there is no consensus in current MMSs for how the library should be organized and constructed. We developed a theory library built in MSL, the specialized language of "theory types", which employs and formalizes a collection of basic algebraic structures. The library emphasizes the interface-manipulation and expressivity of theory abstractions rather than full-fledged low-level axiomatic and algorithmic theories. The library also supports some *specification-building operations*, such as theory combination, renaming, and extension to construct new theories from existing theories in a structured fashion. In fact, our work on library module system construction derives from many structuring mechanisms developed in many algebraic specification languages.

Chapter 2

Intersection Between MMS and Programming and Specification Languages

We examine, in this chapter, the intersection between MMS and programming and specification languages. In particular, we are interested in the modularity mechanisms for programming and specification languages for MMSs.

2.1 Programming languages

Dialects of ML usually have a rich module system. An ML-family module system itself is a small typed functional language, where structures and functors are objects and signatures are types. While modules can be manipulated in various ways, types (signatures) are described by enumerating their parts. N. Ramsey, K. Fisher and P. Govereau [19] present a language that maniplu-ates signatures, e.g. two signatures can be combined to form a new signature. The language of signatures exposes no information about how a structure matches a signature. For instance, a signature that combines two signatures is the same as the *flat* signature that is the union of the two component signatures. A structure that implements the combination signature also implements the flat signature.

In the module system of our theory library, a module expression of a union type must be built from two subexpressions of corresponding component types. In another view, the language of signatures can be seen as our module type level in the sense that the operations are over module types instead of module expressions.

2.2 Specification languages

While most specification languages support theory building operations: extension, combination, and renaming as in module system of our theory library, they usually have a different parameterized specification mechanism. We have taken many useful idea from CASL [16]. CASL is an algebraic specification language based on partial first-order logic. A basic specification consists of a signature Σ and a set of sentences (axioms or constraints) over Σ.

CASL provides a number of mechanisms for structuring specifications, the followings have their counterparts in the module system of our library.

- Extension: SP_1 `then` ...`then` SP_n

  SP_1 determines an extension from the local environment to a complete signature Σ_1. For $i = 2, \ldots, n$ each SP_1 determines an extension from Σ_{i-1} to a complete Σ_i. The signature determined by the entire extension is then Σ_n.

- Union: SP_1 `and` ...`and` SP_n

The signature of the union is obtained by the ordinary union of the Σ_i (not the disjoint union). Thus all (non-localized) occurrences of a symbol in the SP_i are interpreted uniformly. If the same name is declared both as a total and a partial operation with the same profile (in different signature), the operation becomes total in the union.

- Translation: SP `with` SM

  This is a renaming mechanism and SM is a symbol mapping. The signature Σ given by SP and symbol mapping SM together determine a new signature $\tilde{\Sigma}$ and a morphism from Σ to $\tilde{\Sigma}$.

Chapter 3

Design Goals of Theory Library

This chapter presents the major design goals of our theory library. Some of design goals proposed in § 3.2 determine modularity mechanisms and features supported by the library.

3.1 Theory library in an interactive mathematics laboratory

The ultimate goal of the MathScheme project [15] is to build, on top of the mechanized mathematics system, an *interactive mathematics laboratory* (IML) [4, 7], a computer system that provides a set of integrated tools for facilitating the mathematics process and managing mathematical reasoning. An IML offers a formal, interactive, and mechanized environment that combines the capabilities of both computer theorem proving systems and computer algebra systems. To support the process the users use to explore, apply and extend the mathematics in such system, a library of rich mathematical knowledge is considered one of the essential elements. Mathematics library should be a web accessible environment that includes dynamically stored mathematical knowledge. We have chosen to use the term "theory library" to represent the mathematics library because it provides a theory development facility in our system. The theory library should help *end users* to have easy access to the network of well-organized algebraic structures; on the other hand, it should support the development of producing and formalizing new mathematical knowledge by *developers*.

3.2 Design goals

In this section, we explicitly present the design goals (DGs) we want to achieve. A theory library could be viewed and constructed in many different ways. A good modularity of theory library aids the expressivity of MMSs. It can help to build up contexts and allow the user to reuse the theorems developed within one context in other contexts with similar structure. Keeping the ultimate goal in mind, a desirable implementation of the library would be to create an "expressive" environment that provides the ability to allow the user and developer to browse and expand the theory library in sophisticated ways. Our work eventually integrates the ideas from modular programming systems and algebraic specification module systems to build a library of theory types suited to the MathScheme project.

3.2.1 DG1: A library of well-presented theories

Mathematics is conventionally done in informal (in theory, it could be made formal) high-level reasoning environments that include an integrated set of basic concepts and deductive, computational, visual, and other kinds of practical tools. To emulate a similar environment in a *mechanized mathematical* *system* (MMS), a library of theories should be abstract, highly structured, and interconnected mathematics in an organized fashion. Such library has the best chance to provide a rich set of abstract concepts and definitions without too many low-level details exposed. Not only does the end user have the mathematical information he or she needs, but also the mathematical information in the library would be well encapsulated. Moreover, it is certainly feasible if the same theory can be derived from several different underlying sets of axiomatic theories.

3.2.2 DG2: A library of well-structured theories

Our aim is to present end users a rich set of abstract theories. Such library can be constructed incrementally from low-level theories. A theory[1] consists of a language, a set of axioms, and a set of theorems. Low-level theories should be considered as part of the supporting infrastructure of the theory library. Certain desirable property enables MMS developers, who are interested in the structure of mathematics and the problems involved in formalizing mathematics, to facilitate the development and expansion of the library.

We favor using the *little theories* approach as the structuring mechanism for our system. The "little theories" method is one of several versions of the axiomatic method. It is described in W. M. Farmer, J. D. Guttman, and F. J. Thayer's paper "Little Theories" [10], pg. 2:

> *In the little theories method, a number of theories will be used in the course of developing a portion of mathematics. Different theorems*

[1]Our notion of a theory may be called a theory presentation in some other systems.

will be proved in different theories, depending on the amount of structure required.

There are several advantages to the little theories approach presented in this paper. We summarize them as follows:

i. **Relation to the big theory approach**

   The little theories approach is opposed to the modules in most programming language paradigms which are based on the *big theory* approach. In the big theory approach, one powerful set of axioms is used to model all objects of interest. Consequently, a module is a name scope mechanism, where an object developed in one module can be referred to from within another module by qualifying the object name. In the little theories approach, the work of establishing the network of interconnected theories can be carried out under different, separated contexts being modeled by theories.

ii. **A modular construction**

   In analogy with programming tools between the two extremes of programming paradigm and low-level primitives, like classes, modules, or functor, the little theories method assembles a modular construction that is recorded in the structure of the library of abstract theories. Thus the network of abstract theories is derived and instantiated in a structured fashion from its underlying set of basic theories.

iii. **Use of minimal axiomatization**

   The little theories approach also ensures the use of minimal axiomatization for specific groups of theorems through interpretation into other theories, or though direct inclusion in larger theories. Another advantage is to allow theorems to be written in simpler forms encoded in theory expressions in the library.

iv. **Supported in theorem prover**
Because of its usefulness for establishing consistency and independence, the little theories approach has become a deeply entrenched way of organizing mathematical knowledge. So far as we know, `IMPS`, an Interactive Mathematical Proof System [9], is the first interactive theorem prover to have been designed from the start to support little theories.

We will discuss some examples of theory construction and decomposition as how our library supports the approach in § 4.2. Our examples exemplify the usefulness of the little theories approach in MathScheme Language (MSL).

3.2.3 DG3: A library of interconnected and extended theories

In the little theories version of the axiomatic method, mathematical knowledge is distributed over a network of theories linked to one another via theory interpretations [8]. To use an object developed in another theory under the current context, we need to build a theory interpretation between them. Effectively, it will import a translated version of that theory implicitly and explicitly. Thus, the connection of one abstract theory to another abstract theory in the theory library is largely due to the power that theory interpretations provide. It should give us the utilities to achieve theory abstraction goals and have the MMS properly interpret the result.

We use theory importing and extension to be important modular technique to support theory reuse in our library. Each theory in our library is a concrete representation of some mathematical theory, in contrast to approaches (as in many algebraic specification languages) where a theory contains all theorems that are provable. And more complicated theory constructions can be expressed with supported operations such as renaming and extension. Theory building operations is an import technique described in § 4.3.

3.2.4 DG4: An expressive language of signatures

The modular system for our theory library should itself be a typed functional language, where structures and functors are objects and signatures are types. We need a language to describe theories. This language of "theory types" is intended primarily as a tool for program specification, but it also serves to represent mathematical knowledge in a manipulable form. A language that manipulates signatures [19] is presented by N. Ramsey, K. Fisher, and P. Govereau. While modules can be manipulated in various ways, signatures are described by enumeration their parts. The language of signatures exposes no information about how a structure matches a signature.

Chapter 4

High Level Tools and Techniques

This chapter presents some high level tools and techniques we intend to use to meet our design goals of theory library described in § 3.2.

4.1 Biform theory in Chiron

Developed by Hong Ni [17], the definition of the notion of a biform theory is implemented on a very basic basis with three experiments for some kind of environments. An *environment* is a well-designed interface for exporting the transformer implementation of the kernel theories and libraries of *Chiron.*

Chiron [6] is an exceptionally well-suited logic for formalizing biform theories since it has a high level of both theoretical and practical expressivity. Precisely speaking, the meaning formulas of rules can be directly expressed in Chiron.

4.2 Little theories in the library

As indicated in library design goals, our aim is to present to users a rich set of theories. For that purpose, we favor using the little theories method to organize mathematics in theory library. In little theories method, a complex body of mathematics is represented as a network of axiomatic theories. Bigger and advanced theories are composed of smaller and basic theories. Theories are linked by building operations and interpretations. Reasoning is distributed over the network. These can be assembled in a principled and modular fashion and implemented atop a module framework like Jian Xu's Mei [22].

4.2.1 Little theories and IMPS

The little theories [10] idea is a familiar ingredient in work on specification languages. This characteristic lies off the main path of our work in constructing module system of the theory library as the language of "theory type" holds the power of specification.

There has been some work on supporting little theories in logic framework and mechanized theorem proving. A great deal of previous work shows an approach of combining logics and theories to be proven beneficial in practical use. Developed at The MITRE Corporation by W. Farmer, J. Guttman, and J. Thayer, IMPS [9] is an Interactive Mathematics Proof System. The main approaches of IMPS are to support traditional mathematical techniques and human oriented instead of machine oriented. The IMPS methodology for formalizing mathematics is based on a particular version of the axiomatic method. IMPS is the first interactive theorem prover to have been designed from the start to support little theories for organizing mathematics, essentially for formalizing large portions of mathematics. As far as we know, IMPS provides stronger support for little theories than any other contemporary theorem proving system.

4.2.2 Examples

The little theories method is used both for encoding existing mathematics and for creating new mathematics. In our theory modular system, a number of theories are used in the course of developing a portion of other theories. Theories are logically linked together by theory building operations which serve as conduits to pass results from one theory to another. This approach of organizing algebraic structures across a network of linked theories is advantageous for managing complex structures by means of *abstraction* and *resue.* We give an example of how a classical concept *Group* is defined and how a properly decomposed version of Group is being done through the little theories method. This method applied in MathScheme Language (MSL) satisfies our second design goal (DG2).

Classically, a *group* G is a finite or infinite *set* of elements together with a *binary operation* that satisfy the four fundamental properties:

1. Closure: if A and B are two elements in G, then the product $A \cdot B$ is also in G.

2. Associativity: The defined multiplication is associative, i.e., for all $A, B, C \in G$, $(A \cdot B) \cdot C = A \cdot (B \cdot C)$.

3. Identity: There is an identity element e such that $e \cdot A = A \cdot e = A$ for every element $A \in G$.

4. Inverse: For each element $A \in G$, the set contains an element $B = A^{-1}$ such that $A \cdot A^{-1} = A^{-1} \cdot A = e$.

Now we present how to decompose a theory of *Group* through little theories method. Theories in the classic definition i.e., set, binary operation, closure, associativity, identity, and inverse are necessary to be included. The theories are constructed step-by-step using theory building operations such as theory extension and combination (see § 5.3). We begin by defining the network of interrelated theories used in the *Group* theory construction.

- A theory of a carrier.
- A theory of a binary operation over a carrier set.
- A theory of a magma.
- A theory of a pointed magma.
- A theory of associativity.
- A theory of a loop (constructed incrementally from a theory of unital and a theory of quasi-group).
- A theory of a group.

From these theories we build a number of other theories through little theories method. We briefly explain the interrelated theories (terms) in the algebraic structure's definitions.

- A *Carrier* is a set of universal objects that is dependent on a *CarrierType*.

- A *BinaryOperation* $**$ on a carrier U is a binary function that maps elements of (U, U) to U.

- A *Magma* consists of a carrier S equipped with a single binary operation T.

- A *PointedMagma* consists of a magma and a pointed carrier along with a supported carrier.

- A *Unital* is a magma with an identity element.

- A *QuasiGroup* is a cancellative magma.

- A *Loop* is a quasigroup with an identity element.

- A *Group* is a loop with associative magma.

We have interpreted these concepts by using the language of "theory-types".

- A theory of a carrier set is obtained by extending a carrier type and a theory of binary operation is built as an extension of the carrier set.

  ```
  Carrier = CarrierType extended by { U:carrier };;
  ```

  ```
  BinaryOperation = Carrier extended by { **:(U, U)->U };;
  ```

- A theory of a pointed magma is obtained by including a magma with a pointed carrier along with a supported carrier.

  ```
  PointedMagma = Theory
    {
      combines Magma, PointedCarrier along Carrier
    };;
  ```

- Using a carrier equipped with a binary operation, a theory of magma is constructed with no specifications.

  ```
  Magma = Theory { BinaryOperation with ** = * };;
  ```

- A theory of associativity is formed as an extension of a theory of binary operation by adding an associative property.

  ```
  Associativity = Theory
    {
      property assoc(**)
        := forall x,y,z in U. (x**(y**z))=(x**y)**z
    };;
  ```

- A theory of loop is constructed incrementally from several subtheories. A theory of loop is built as a union of a theory of unital and a theory of quasigroup. A theory of quasigroup is obtained as an extension of a theory of magma by adding a cancellative property. As the same approach, a theory of unital is defined by extending a theory of pointed magma by adding an identity property.

```
Unital = PointedMagma extended by
  {
    import Identity;
    axiom identity(e,(*))
  };;

QuasiGroup = Magma extended by
  {
    import Cancellative;
    axiom cancellative(*)
  };;

Loop = Theory
  {
    combines Unital, QuasiGroup along Magma
  };;
```

- By applying theory building operation, a theory of *Group* is finally constructed with the combination of two subtheories *Associativity* and *Loop*.

```
Group = Loop extended by
  {
    import Associativity;
    axiom assoc(*)
  };;
```

4.3 Theory building operations

Although we can always formalize a theory from scratch, it is convenient if we can reuse previously developed theories. Our module mechanism should support theory building techniques, such as renaming, extension, and combination implemented in most algebraic specification systems. For example, renaming can be used to avoid unintended name clashes, or to adjust names of sorts and change notations for operations. These theory building operations are supported in many module systems such as CASL [16] and MEI [22]. We explain theory building operations used in our library module system through examples in § 5.3.

Chapter 5

A Module System of Theory Representations

As indicated in § 1.3, the library module system is built and developed upon many nice features supported by two families of module systems: the typed functional language module system and the algebraic specification module system. In this chapter we describe the library module system and its supported building mechanisms, followed by a formal presentation of its syntax and examples to clarify them.

5.1 Theories

Algebraic structures in our library are organized as modules called *theories*. A theory in our library may be called a *theory representation* of already proved theorems, as opposed to approaches where a theory consists of all the theorems that are provable. Thus, our notion of a theory is a syntactic object in terms of the underlying MMS. In other words, our module system representing the theory library manipulates only syntactic representations (*interface*) of algebraic structures.

5.2 Theory development

Taking experiences from theory development apporach from IMPS, the user creates a theory and the mathematical object associated with it by evaluating theory expressions. The theory expressions supplied by the system and created by the user can be stored in a file which can be parsed as needed into a running process. In this section, we give an overview of the tasks that are involved in creating a well-developed theory. By presenting MathScheme Language (MSL) syntax in examples, our design goals of constructing a well-organized, interconnected library are well met.

- **Built from scratch** The first task in developing a theory is to build a primitive, bare bones theory T.
- **Built from basic theory** Once the barebones theory is built, we can build more advanced and complex theories. There are theory building mechanisms for doing this that can be used separately or in combination. These building operations are presented in order in the next section. Such theory creation techniques include:
  - Extension of a theory.
  - Renaming of a theory.
  - Union of several theories.

5.3 Theory building operations

Theories can be written down explicitly one at a time. As soon as they get to be complex, we wind up with a large set of expressions that prevents us easily interpreting the theory itself. So we must build our theories up from small intelligible pieces. We often build one theory on top of another. Our work on theory building operations derives from many techniques adopted in most algebraic specification languages, as well as theory building operations, such as "combine" and "enrich" proposed in R. M. Burstall and J. A. Goguen's paper [1]. We will explain these operations informally, using examples.

5.3.1 Theory extension

Extension is a very useful reasoning technique to add machinery to a theory by means of a theory extension. Thus, extending an existing theory by adding new symbols is an approach to form a structured theory hierarchy. To develop a new theory, instead of starting from scratch, we can start from an existing theory and extend it by adding new language symbol and axioms. In our language, this operation is identified by conjoin keywords "extended by". This setup would eventually allow one to prove results in an *enriched theory* [1] and then transport them back to the unenriched theory.

To make our presentation concrete, let $L_i = (C_i, t_i)$ be a language for $i = 1, 2$. L_2 is an extension of L_1 (and L_1 is a sublanguage of L_2), written $L_1 \leq L_2$, if $C_1 \subseteq C_2$ and t_1 is a subfunction of t_2.

Definition 1 Let $T_i = (L_i, \Gamma_i)$ be a theory for $i = 1, 2$. T_2 is an extension of T_1 (and T_1 is a *subtheory* of T_2), written $T_1 \leq T_2$, if $L_1 \leq L_2$ and $\Gamma_1 \subseteq \Gamma_2$, where Γ is the set of axioms of T.

Example 1

```
CarrierType = Theory { carrier:type };;


Carrier = CarrierType extended by { U:carrier };;
```

A *CarrierType* is a type to represent a carrier set. With extending the theory of CarrierType, a *Carrier* is a set of universal objects that is dependent on a carrier type. We need it as a basic data type in our theory development. Hence, an extension of a theory T is obtained by adding new vocabulary and axioms to T. A theory development can be viewed as a sequence of theory extensions.

5.3.2 Theory renaming

Renaming is an important mechanism to avoid name clashes. A renaming is introduced by the keyword "with". This is illustrated in Example 2, one may use $**$ to represent a binary operation in a theory of `Magma` and it is necessary to rename $**$ to $*$, representing a binary operation on the same carrier set.

Example 2

```
Magma = Theory
  {
    BinaryOperation with ** = *
  };;
```

Another reason is to adjust name symbols according to the semantics. As shown in Example 3, without renaming, it is quite possible that `AbelianGroup` theory ends up using $*$ as the name of its binary operation and e as the name of its identity element in a theory of `Group`.

Example 3

```
AbelianGroup = Theory
  {
    Group with * = +, e = 0;
    import Commutativity;
    axiom comm(+)
  };;
```

5.3.3 Theory combination

One motivation for the design of "theory type" language is a rich combination of concepts. Similarly, an expressive approach to constructing a theory relies on the union of two or many simpler theories or properties. In other words, to develop a new theory, instead of stating its language and set of axioms, we can start from combining existing theories. For instance, an operation to build a theory of `Ring` is to extend the combination of a theory of `AbelianGroup` and a theory of `Monoid` with renaming operation (see § 5.3.2). The idea is illustrated in the following example:

Example 4

```
AbelianGroup = Theory
```

```
  {
    Group with * = +, e = 0;
    import Commutativity;
    axiom comm(+)
  };;

Monoid = Theory
  {
    combines Unital, SemiGroup along Magma
  };;

Ring = Theory
  {
    Import Distributivity;
    S1 := AbelianGroup;
    S2 := Monoid with e = 1;
    combines S1, S2 along S2;
    axiom distri( *, + );
  };;
```

By renaming, we identify the binary operation as additive operation $+$ and multiplicative operation $*$, separate e as 0 and 1, and combine `AbelianGroup` and `Monoid` with `Distributivity` property.

Chapter 6

MathScheme Language Generation and Implementation

This chapter describes the generation and implementation of MathScheme Language (MSL). This language contains many abstracted expressions built from low-level and basic algebraic theories. We mainly focus on syntax implementation of MSL and this language is logic independent. The lexical conventions are presented in § 6.3. The abstract syntax structure is described in section § 6.5.

6.1 Lexing and parsing overview

Compilers and interpreters take as input programs in string form. Most of the foremost interaction with mathematical modeling systems can also be considered as to build a formula in a convenient and human-pleasant way. A parser takes a formula represented as a string and produces a formula represented as a data structure that the system can deal with.

Lexing and parsing are the first two steps towards converting this string input into *abstract syntax tree (AST)* in the language that can then be interpreted. That is, the process of building an internal-expression from an input string can be broken up into two parts:

- Lexing or tokenization of the input string. This can be thought of as breaking the input string into a sequence of smaller different syntactic categories, called *tokens.* Tokens are often separated by spaces, newlines, operators, and other characters like semicolon and parenthesis.
- Parsing of the sequence of tokens. Once a string has been tokenized into a sequence of tokens, the parsing takes the sequence and converts lists of tokens into ASTs according to the rules defined by the grammar.

6.2 Lexer and parser generator

The tokens of a programming language are specified using regular expressions, and thus the lexing process involves a great deal of regular-expression matching. It would be tedious to take the specification for the tokens of our language, convert the regular expressions to a Deterministic Finite Automaton (DFA), and then implement the DFA in code to actually scan the text.

Instead, most languages come with tools that automate much of the process of implementing a lexer in those languages. To implement a lexer and a parser with these tools, you simply need to define the lexing behavior and parser grammar in the tool's specification language. The tool will then compile your specification into source code for an actual lexer and parser that you can use.

We have chosen OCaml tools to build our lexer and parser.

- *ocamllex* the lexer-generator, that produces a lexical analyzer from a set of regular expressions with associated semantic actions.

- *ocamlyacc* the parser-generator, that produces a parser from a grammar with associated semantic actions.

The core and support machinery of the lexer and parser were written in two specification files in particular formats that *ocamllex* and *ocamlyacc* can process, and they generate pure OCaml code that can be executed to lex and parse strings.

6.3 Lexical conventions

This section covers lexical conventions for our language. Blanks, comments, and identifiers are given in Appendix A. Keywords and operators are discussed in the subsequent sections.

6.3.1 Blanks, comments and identifiers

Refer to Appendix A.

6.3.2 Keywords and logical operators

Keywords

The identifiers below are reserved words, defined as *keywords* in Table 6.1.

signature	property	axiom	theorem
Theory	implies	iota	combine
combines	along	with	enrich
Inductive	by	extended	type_plus
Import	import	in	conservatively
Concept	Concepts	concept	concepts
Transformer	Transformers	transformer	transformers
Definition	Definitions	definition	definitions
Fact	Facts	fact	facts

Table 6.1: Keywords

Each of these keywords is designed to represent a start of a language element, a membership, a relationship, or a theory building operation (see § 5.3).

- `signature`: indicates a definition of a signature and is used when you want to create a new type.
- `property`: indicates the start of a property.
- `axiom`: indicates the start of the definition of an axiom.
- `theorem`: indicates the start of the definition of a theorem.
- `Theory`: indicates the start of the definition of a theory.
- `implies`: indicates an implication.
- `iota`: indicates an iota expression, which is a definite description operator for objects of kind.

- `combine, combines`: represents theory combination building operation. It indicates that a list of theories is combined.
- `along`: indicates supporting element of a theory.
- `with`: represents the renaming building operation.
- `enrich`: indicates the start of an enriched theory declaration.
- `Inductive`: indicates an induction expression.
- `by`: followed by the keyword "extended".
- `extended`: represent the extension building operation. It indicates that the theory you are writing has an inheritance.
- `type_plus`: indicates *plus types* for the domains of quantification.
- `Import, import`: indicates a "theory import" declaration.
- `in`: indicates membership.
- `conservatively`: indicates a model conservative extension, which introduce new symbols that are defined in terms of old vocabulary.
- `Concept, Concepts, concept, concepts`: indicates the start of "a concept" or "concepts".
- `Transformer, Transformers, transformer, transformers`: indicates the start of "a transformer" or "transformers".
- `Definition, Definitions, definition, definitions`: indicates the start of the definition of a variable.

- `Fact, Facts, fact, facts`: indicates the start of "a fact" or "facts".

Logical operators

All the logical operators defined in our language is grouped together in Table 6.2.

`forall`	`exists`	`and`	`or`	`not`

Table 6.2: Logical operators

We explain what each logical operator represent.

- `forall`: refers to universal qualification.
- `exists`: refers to existential qualification.
- `and`: represents the relationship of Boolean operator "and".
- `or`: represents the relationship of Boolean operator "or".
- `not`: represents the relationship of Boolean operator "not".

6.3.3 Special tokens

The following sequences of characters are special tokens, as defined in Table 6.3.

Each token has its own rule in the lexer specification. The lexer tries to match the longest string possible each time. Hence "`;;`" is match with `DOUBLESEMI`, instead of two `SEMICOLON` tokens.

,	:	;;	;	->	.	(
)	{	}	[	]	=	|
:=	?					

Table 6.3: Operators

Also the regular expression rules are tried on the input from top to bottom. Hence, if "fun" is defined as a reserved word (FUN) ahead of alphanumeric identifiers, the lexer will recognize it as FUN and not the variable "fun".

6.4 Representing parse trees in OCaml

Our end goal of parsing will be to build an OCaml data structure representing the parsed form of a program. Our general strategy can be broken down into multiple phases:

- Create an OCaml datatype for each syntactic category in the language.
- Use this datatype, most likely to be *mutually recursive*, to represent the inherent recursive structure of language definitions.
- Generate an OCaml term, using these mutually recursive types, representing the parsed form of the program - containment in a type constructor shows that the contained items are children of the containing item in the AST.

The next section will discuss more of the AST structure in our language.

6.5 AST structure

As the result of parsing, an AST of the input program will be returned. The abstract syntax for our language is given by the following mutually-recursive OCaml type definitions.

6.5.1 Top-level expressions

Block 1 Top level expressions

```
type assign = Assign of ident * theory_expr


type toplevel_expr = assign list
```

A program is a class of *top level expressions*, assigned by a list of theory expressions, as shown in Block 1. A top level expression has multiple cases with:

- `T = theory expression.`
- `T := theory expression.`
- `enrich T with { declaration }.`

6.5.2 Theory expressions

Block 2 Type definition of theory expressions

```
type theory_expr =

  | ThyExpr of declaration list

  | ThyName of simple_app

  | ThyFunc of thytypedecl list * declaration list

  | ThyExtend of theory_expr * declaration list * qual
```

A *theory expression*, as shown in Block 2, can be four types of theories:

- ThyExpr a basic theory declaration, such as $T = \mathit{Theory}\ \{\ \}$.
- ThyName theory declaration with copy in extension.
- ThyFunc theory declaration containing functor(s).
- ThyExtend theory declaration with extension operation such as $T = \mathit{Theory\ extended\ by}\ E\ \{\ \}$.

6.5.3 Declarations

Block 3 Type definition of declarations

```
type declaration =

  | Rename of theory_expr * subst list

  | Prop of property

  | DefWithRenam of ident * theory_expr * subst list

  | TypDecl of typedecl

  | FuncDecl of funcdefn

  | AxBase of ident * expr * bool

  | AxFunc of expr * bool

  | Inductive of simp_app * constructor list

  | CombDecl of simple_app list * simple_app option

  | Import of ident list

  | Concept of typedecl list

  | Definition of funcdefn list

  | LocalThyExtend of theory_expr * declaration list * qual
```

Block 3 shows type definitions of *declaration* in our program. They are broken up into:

- `Rename` denotes definition of "T(S) with renaming" with most parts optional.
- `Prop` denotes a property declaration.
- `DefWitRenam` denotes definition of "T := Theory(T) with renaming".

- `TypDecl` denotes a type declaration.
- `FuncDecl` denotes a single function declaration.
- `AxBase` denotes basic axiom or theorem declaration.
- `AxFunc` denotes axiom or theorem as functor declaration.
- `Inductive` denotes an induction declaration.
- `CombDecl` denotes combination of theories.
- `Import` denotes a "theory imports" declaration.
- `Concept` denotes a concept block.
- `Definition` denotes a definition block.
- `LocalThyExtend` denotes a theory extension.

6.5.4 Expressions

Block 4 Type definition of expressions

```
type expr =
  | Ident of ident                (* identifier *)
  | Oper of oper                  (* operator *)
  | EqOp of expr * expr           (* 'equal' operation *)
  | PairOp of expr * expr         (* pair operation *)
  | And of expr * expr            (* 'and' operation *)
  | InOp of expr * string         (* 'in' operation *)
  | Or of expr * expr             (* 'or' operation *)
  | Not of expr                   (* 'not' operation *)
  | Implies of expr * expr        (* implication *)
  | Appl of application           (* application *)
  | Forall of var_spec * expr     (* for all *)
  | Exists of var_spec * expr     (* there exists *)
  | Iota of var_spec * expr       (* iota expression *)
```

The abstract syntax for *expressions* has obvious meanings, with explanatory comments as shown in Block 4.

6.5.5 Type definition of application

Block 5 Type definition of application

```
and application =
  | ExprApp of ident * expr list
  | OpApp of ident * ident list
  | BinOp of op * expr * expr
```

The abstract syntax for *applications* is shown in Block 5.

- `ExprApp` denotes an expression application such as *T(S1, S2,... ,S)*.
- `OpApp` denotes an operator application such as *T(**, ++)*.
- `BinOp` denotes a binary operation such as *expr1 op expr2*.

6.5.6 Type definition of types

Block 6 Type definition of types

```
and typedecl =
  | TBase of ident * type_comp
  | TExtension of ident list * type_comp
```

The abstract syntax for *types* is shown in Block 6.

- `TBase` denotes a basic type declaration such as $a : Int \longrightarrow Int$.
- `TExtension` denotes an extended type declaration such as $m,\ n,\ \ldots\ u,\ v : Int \longrightarrow Int$.

6.5.7 Function definition, theory type declaration, simple application and substitutions

Block 7 Type definition of function, theory type declaration, simple application and substitutions

```
and funcdefn = simple_app * expr

and thytypedecl = ident * ident

and simple_app = SimpApp of ident * ident list

and subst = ident * ident

and subste = ident * expr

and substt = ident * type_comp
```

The abstract syntax of *function definition*, *theory type declaration*, *simple application*, and *substitutions* are shown in Block 7.

- `funcdefn` denotes a function definition as of type of a simple application and an expression.
- `thytypedecl` denotes a theory type declaration as of identifiers.
- `simple_app` denotes a simple application as of type of substitution list.
- `subst` denotes a substitution list as of type of identifiers.
- `subste` denotes a substitution list as of type of an identifier and an expression.
- `substt` denotes a substitution list as of type of an identifier and an composed type expression.

6.5.8 Type definition of composed types

Block 8 Type definition of composed types

```
and type_comp =
  | TId of simple_app
  | TProd of type_comp list
  | TPlus of type_comp list
  | TArrow of type_comp * type_comp
  | TInduct of constructor list
  | TPredicate of type_comp
```

The abstract syntax of *composed types* is shown in Block 8.

- `TId` defines basic unit of type declaration.
- `TProd` denotes composed type declaration such as $a : (T, S, \ldots\ V) \rightarrow Int$.
- `TPlus` denotes type plus declaration.
- `TArrow` denotes composed type declaration such as $a : Int \rightarrow Int \rightarrow Int$.
- `TInduct` denotes an inductive type declaration.
- `TPredicate` denotes a predicate type declaration.

6.5.9 Type definition of property and constructor

Block 9 Type definition of property

```
and property = simple_app * expr


and constructor =

  | TConstr of ident * type_comp
```

The abstract syntax of *property* is shown in Block 9.

- `property` denotes a property declaration such as "property T(x) :=".
- `constructor` denotes a constructor declaration such as "T : type def".

6.5.10 Type definition of variable specification

Block 10 Type definition of variable specification

```
and var_spec =

  | VarSpec of ident list * type_comp
```

The abstract syntax of *variable specification* is shown in Block 10. A variable specification consists of a variable list and a composed type definition. For example, a variable specification in our language can be Id_1, Id_2, $\ldots Id_n$ in CompType or Id_1, Id_2, $\ldots Id_n$:CompType.

Chapter 7

Implementation of Algebraic Structures in Theory Library and Some Useful Utilities

In the forgoing chapter, we have seen examples of constructing modules (theories) through the language of "theory types" together with building operations. This chapter presents the implementation of basic algebraic structures in an organized fashion as adopted by *Common Algebraic Specification Language* (CASL) [16].

7.1 Overview of theory library

The creation of libraries facilitates the module system building mechanisms of theories. The collection of algebraic structures presented here consists of the following libraries:

- Relations and Orders
- Basic Algebraic Structures

7.2 Notion of theory

Each algebraic structure is syntactically denoted as a "Theory". Our notion of theory is expressed through the following entities:

```
ConceptName

Definition

Theory Code
```

Please refer to Appendix B for instantiated representations of each theory element.

7.3 Theory extractor - A tool for literate programming

The idea of Literate Programming is by Donald E. Knuth (see also [13]) in Literate Programming. CSLI, 1992, pg. 99:

> *Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.*

Our system provides support for literate programming with a utility: "theory extractor" that extracts out *theory* source code in LaTeX version of theory library. The library is written using a special format, which means the LaTeX and theory code are interwoven into a single source document. This approach to mixing theory description with theory source code also encourages the user to adopt literate programming from the outset, so that the end product of their own input is reproducible and readable program.

This Perl program written for thesis, by default, extracts the code out of the "theory" environments containing the theory source portionwise and distributes contents over a single file or different target files which could be parsed directly.

7.3.1 Usage and Options

Usage: `MS-xtr.pl` <*options*> *texfile* <*codefile*>

`MS-xtr.pl` searches for a marker pattern followed by a blank and *codefile* and copies the content of the following environment into the target file *codefile*. This can be used to extract only the contents corresponding to a certain part of the program instead of the whole lot. If no *codefile* is given, the default output file is "lib.msl" so it can be parsed as input file.

Options:

`-a` extract all code into different target files, so it is not supposed to be used together with *codefile*. This is useful if you have program code which is intended to be distributed over separate files; when documenting it you will use a "marker pattern", such as `%%@ code.msl` preceding each theory environment containing parts of `code.msl`, and similarly with the other target files. With the `-a` option, the extractor will interpret `code.msl` as the name of the file which shall contain the content of the following environment, and will copy the contents of all environments of the selected type into the respective target files. The use of a marker pattern is necessary here, and no *codefile* should be specified.

-e*environment* extract content of *environment* instead of the default *theory* environment.

-p*pattern* marker pattern for environments to extract; default is %%@. A null pattern is possible.

-h show help, usage and synopsis of standard options.

7.3.2 Example

Consider a partial LaTeX file of MathScheme theory library "MS-lib.tex" with the following description of two theory expressions:

```
\concept{Abelian Group}
\abbreviation{AbGrp}
\libclass{\Alg}
\begin{MSdefinition}
{\rm An \cptnm{AbelianGroup} is a commutative group.}
%%@ algebra.msl
\begin{theory}
AbelianGroup = Theory
  {
    Group with * = +, e = 0;
    import Commutativity;
    axiom comm(+)
  };;
\end{theory}
```

```
\end{MSdefinition}
```

With `MS-xtr.pl -a MS-lib.tex`, all the code in the theory environment will be extracted, copying everything marked with `%%@ algebra.msl` into a file `algebra.msl`. Some theory expressions that are marked with `%%@ relations-orders.msl` will be extracted into a file, namely `relations-orders.msl`. With `MS-xtr.pl MS-lib.tex algebra.msl` you can extract everything marked with `%%@ algebra.msl`; with `MS-xtr.pl -eMSdefinition MS-lib.tex` you can extract all lines of all MSdefinition environments.

7.4 Theory expander

As a mathematical object, a *theory* consists of a language L and a set of sentences in L called *axioms*. Theory expander provides the technique for expanding abstracted "theory" in our theory library and reconstructing it as a language and a set of axioms.

The theory expander is currently being developed by Dr. Jacques Carette.

7.4.1 Example single-level expansion

Theory of Reflexivity is a Theory of UnaryRelation extended by a reflexive property and expressed with the following syntax:

```
Reflexivity = UnaryRelation extended by
  {
    property refl(R) := forall x in U. (x 'R x)
  };;
```

The expander expends `UnaryRelation` with a theory of UnaryRelation, which is a `BinaryRelation` extended by an axiom,

```
UnaryRelation = BinaryRelation extended by
  {
    axiom U = V
  };;
```

the expander expands `BinaryRelation` with a theory of BinaryRelation, which is `CarrierType` extended by a binary relation representation.

```
BinaryRelation = CarrierType extended by
  {
    U:carrier;
    V:carrier;
    R:(U, V)->Bool
  };;
```

A Theory of BinaryRelation is a theory of CarrierType extended by a type of carrier, and it reaches the lowest level where no more theory can be extended.

```
CarrierType = Theory { carrier:type };;
```

therefore, a theory of Reflexivity is decomposed through the three subtheories and expressed one by one as follows:

```
Reflexivity = Theory {
  carrier:type;
  U:carrier;
  V:carrier;
  R:(U, V)->Bool;
  axiom U = V;
  property refl(R) := forall x in U. (x 'R x) };;
```

7.4.2 Example multi-level expansion

We have showed how theory expander works on a single-level expansion in the previous example, a more complex expansion of a classical `Ring` is decomposed in the following steps:

A theory of Ring is expressed as a theory of Carrier extended by several properties:

```
Ring = Carrier extended by
  {
    Import Distributivity;
    S1 := AbelianGroup with ** = +, e = 0;
    S2 := Monoid with ** = *, e = 1;
    combines S1, S2;
    axiom distrib( *, + );
  };;
```

Starting from *CarrierType*, *Carrier*, and *Singular*, a *Ring* is constructed through these three basic theories.

```
CarrierType = Theory
  {
    carrier:type
  };;
Carrier = CarrierType extended by
  {
    U:carrier
  };;
Singular := Theory
  {
    property singular(x,V) := forall y in V. x = y
  };;
```

To build a theory of *Distributivity*, a theory of *BinaryOperation* is essential on a *Carrier* set.

```
BinaryOperation = Carrier extended by
  {
    **:(U, U)->U
  };;
```

A theory of *Distributivity* is obtained by importing LeftDistributivity and RightDistributivity in addition to a distributive property

```
Distributivity = Theory
  {
    import LeftDistributivity, RightDistributivity;
    property distri(**, ++) :=
      (ldistri(**, ++)) and (rdistri(**, ++))
  };;
```

therefore, a theory of Ring is decomposed through a list of subtheories and expressed one by one as follows:

```
Ring = Theory
  {
    carrier:type;
    U:carrier;
    property singular(x, V) := forall y in V. x = y;
    **:(U, U)->U;
    property distri(**, ++) :=
      ((ldistri(**, ++)) and (rdistri(**, ++)));
    S1 := AbelianGroup with ** = +, e = 0;
    S2 := Monoid with ** = *, e = 1;
    using AbelianGroup with ** = +, e = 0;
    using Monoid with ** = *, e = 1;
    axiom distrib(*, +)
  };;
```

Chapter 8

Conclusion and Future Work

8.1 Conclusion

MathScheme Language (MSL) is an expressive language providing both a formal semantics and a rich specification capability. This language can be a tool for theory specification, but it also serves to present mathematical knowledge in a manipulable form. Since MSL is an abstract and well-organized language, it builds our theory library in a compact and encapsulated fashion. Thus our design goal of a library of well-presented theories is achieved by the presentational feature of MSL. Through the little theories approach, our theory library is constructed incrementally from a portion of existing theories. The example of *Group* demonstrates that such method applied in MSL links theories together by theory building operations. Our deign goal of a library of well-structured theories is met by the usefulness of the little theories approach in MSL. MSL also meets our design goal of a library of interconnected and extended theories since its modular technique supports theory reuse and extension described in § 4.3. A design goal of an expressive language of signatures is achieved by the orthogonality between the one hand basic specifications providing means to write algebraic structures in a specific theory module system, which constructs our theory library; and on the other hand structured and architectural specifications, which have a logic-independent semantics.

8.2 Future work

A library of algebraic theory-types in Appendix B have been developed in the language of theory types. We have showed the library can be implemented using the language of theory-types. The work continues as using the language of theory types to specify and implement many categories of biform theories, such as more advanced algebraic structures, numbers, simple data types like boolean, pair, string, structured data types like array, various kinds of trees, list, map, queue, set, stack and model-building tools from mathematics. These data structures and tools will be used to carefully build up advanced mathematical knowledge in the library.

Bibliography

[1] R. M. Burstall and J. A. Goguen. Putting Theories Together To Make Specifications. *Proc. Fifth Int. Joint Conf. on Artificial Intelligence*, pages 1045–1058. Cambridge, Mass., 1977.

[2] B. W. Char, K. O. Geddes, G. H. Gonnet, B. L. Leong, M. B. Monagan and S. M. Watt. *Maple V Language Reference Manual.* Springer-Verlag, 1991.

[3] Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.1*, INRIA-Rocquencourt, 2006.
Available at *http://coq.inria.fr/doc/main.html*

[4] W. M. Farmer. A proposal for the development of an interactive mathematics laboratory for mathematics education. in: E. Melis, editor, *Proceedings of the Workshop on Deduction Systems for Mathematics Education*, pages 20–25. Carnegie Mellon University, Pittsburgh, Pennsylvania, 2000.

[5] W. M. Farmer. Biform theories in Chiron. in: M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, editors, *Towards Mechanized Mathematical Assistants*, volume 4573 of *Lecture Notes in Computer Science*, pages 66-79. Springer-Verlag, 2007.

[6] W. M. Farmer. Chiron: A multi-paradigm logic. in: R. Matuszewski and A. Zalewska, editors, *From Insight to Proof: Festschrift in Honour of An-*

drzej Trybulec, volume 10(23) of *Studies in Logic, Grammar and Rhetoric*, pages 1-19, University of Bialystok, 2007.

[7] W. M. Farmer. The Interactive Mathematics Laboratory. in: *Proceedings of the 31st Annual Midwest Instruction and Computing Symposium (MICS '98)*, pages 84–94, Fargo, North Dakota and Moorhead, Minnesota, 1998.

[8] W. M. Farmer. Theory interpretations in simple type theory. in: J. Heering et al., editors, *Higher-Order Algebra, Logic, and Term Rewriting*, volume 816 of *Lecture Notes in Computer Science*, pages 96-123, 1994.

[9] W. M. Farmer, J. D. Guttman and F. J. Thayer Fábrega. IMPS: An updated system description. in: M. McRobbie and J. Slaney, editors, *Automated Deduction–CADE-13*, volume 1104 of *Lecture Notes in Computer Science*, pages 298-302, 1996.

[10] W. M. Farmer, J. D. Guttman and F. J. Thayer. Little Theories. in: D. Kapur, editor, *Automated Deduction—CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 567-581. Springer-Verlag, 1992.

[11] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[12] R. D. Jenks and R. S. Sutor. *Axiom: The Scientific Computation System.* Springer-Verlag, 1992.

[13] D. E. Knuth. Literate Programming. CSLI. Stanford, CA, 1992.

[14] B. Krieg-Brückner and D. Sannella. Structuring Specifications in-the-Large and in-the-Small: Higher-Order Functions, Dependent Types and Inheritance in SPECTRAL. *Proc. Joint Conf. on Theory and Practice of Software Development (TAPSOFT)*, volume 494 of *Lecture Notes in Computer Science*, pages 313–336. Springer-Verlag, London, 1991.

[15] MathScheme. An Integrated Framework for Computer Algebra and Computer Theorem Proving.
Website at *http://imps.mcmaster.ca/mathscheme/*.

[16] T. Mossakowski. CASL Basic Libraries. in: M. Bidoit and P. D. Mosses, editors, *CASL User Manual: Introduction to Using the Common Algebraic Specification Language*, volume 2900 of *Lecture Notes in Computer Science*, pages 143–154, Springer-Verlag, London, 2004.

[17] H. Ni. Chiron: Mechanizing Mathematics in OCaml. Masters thesis, McMaster University, July 2009.

[18] S. Owre, S. Rajan, J. M. Rushby, N. Shankar and M. Srivas. PVS: Combining specification, proof checking, and model checking. in: R. Alur and T. A. Henzinger, editors, *Computer Aided Verification: 8th International Conference, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[19] N. Ramsey, K. Fisher and P. Govereau. An Expressive Language of Signatures. *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 27–40. ACM Press, New York, 2005.

[20] P. Rudnicki. An overview of the MIZAR project. Technical report, Department of Computing Science, University of Alberta, 1992.

[21] S. Wolfram. *Mathematica: A System for Doing Mathematics by Computer.* Addison-Wesley, 1991.

[22] J. Xu. Mei - A module system for mechanized mathematics systems. in: *Programming Languages for Mechanized Mathematics Workshop*, Hagenberg, Austria, 2007.

Appendix A

Lexical Conventions

A.1 Blanks

The following characters are considered as blanks: *space*, *newline* and *horizontal tabulation.* Blanks are ignored, but they separate tokens, such as adjacent identifiers, literals and keywords that would otherwise be confused as one single entity.

A.2 Comments

Comments in the language are enclosed between *(@** and **@)*, with no intervening blanks. Comments can also be nested. They can contain any character. Comments are treated as blank characters.

(@ This is a single-line comment. *@)*

(@ This should be a multi-line comment, but breaks line-counting. *@)*

(@ (@* This is a nested comment *@) *@)*

A.3 Identifiers

Identifiers, written *ident*, are sequences of alphabets, digits, _ and \, shown as Table A.1. Identifiers are case-sensitive. They are recognized by the following lexical class: (in ocamllex syntax)

alpha := ['a'-'z' 'A'-'Z']

digit := ['0'-'9']

goodchars := (alpha | digit | _ | \)

ident := goodchars+

Table A.1: Identifiers

Appendix B

Representation of Theory Library

This appendix represents the implementation of a collection of basic algebraic structures in our theory library. The contents are directly extracted from "MS-lib.tex", a LaTeX version of MathScheme theory library.

CarrierType

Definition 2 A *CarrierType* is a type to represent a carrier set. It is the fundamental algebraic structure served in the library.

```
CarrierType = Theory
  {
    carrier:type
  };;
```

Carrier

Definition 3 A *carrier* is a set of universal objects that is dependent on a carrier type. We need it as a basic data type.

```
Carrier = CarrierType extended by
  {
    U:carrier
  };;
```

Pointed Carrier

Definition 4 A *PointedCarrier* is a carrier set having distinguish elements 0 and 1 that is dependent on a carrier set. We will use it to define *PointedBinaryOperation.*

```
PointedCarrier = Carrier extended by
  {
    e:U
  };;
```

Singular

Definition 5 A *Singular* is a pointed carrier C with a *singular* property.

```
Singular := Theory
  {
    property singular(x,V) := forall y in V. x = y
  };;
```

One

Definition 6 A *One* is a one-pointed carrier C, i.e. the theory of Singular where the property is made into an axiom.

```
One = PointedCarrier extended by
  {
    import Singular;
    axiom singular(e,U)
  };;
```

Two

Definition 7 A *Two* is a two-pointed carrier C, i.e. the co-product of One with itself, along Carrier.

```
Two = Theory
  {
    T := One with e = e1;
    combines One, T along CarrierType
  };;
```

Binary Operation

Definition 8 A *BinaryOperation* $**$ on a carrier U is a binary function that maps elements of (U, U) to U. Binary operations are the keystone of algebraic structures studied in abstract algebra: they form part of groups, monoids, semigroups, rings, and more. Most generally, a magma is a set together with any binary operation defined on it.

```
BinaryOperation = Carrier extended by
  {
    **:(U, U)->U
  };;
```

Pointed Binary Operation

Definition 9 A *PointedBinaryOperation* is a pointed carrier set having a binary operation. It is the supporting element of algebraic structures such as Identity and Inverse.

```
PointedBinaryOperation = Theory
  {
    combines BinaryOperation, PointedCarrier along Carrier
  };;
```

Boolean

Definition 10 A *Boolean* is defined as theory instantiation of a *Two* with two renamed elements *true* and *false*. It is served as a basis algebraic structure.

```
Boolean = Theory
  {
    Two with e = true, e1 = false, U = Boole
  };;
```

Binary Relation

Definition 11 A *BinaryRelation* on two carriers U and V is represented as a function $(U, V) \rightarrow Bool$. Binary relations are heavily used in many other theories as a fundamental theory.

```
BinaryRelation = CarrierType extended by
  {
    U:carrier;
    V:carrier;
    R:(U, V)->Boole
  };;
```

UnaryRelation

Definition 12 A *UnaryRelation* on a carrier U is represented as a function $R : (U, U) \to \mathit{Bool}$. This is really a special case of a BinaryRelation where $U = V$, but is here axiomatized separately.

```
UnaryRelation = BinaryRelation extended by
  {
    axiom U=V
  };;
```

Associativity

Definition 13 A binary operation $**$ on a carrier U is said to be *associative* if:

```
Associativity = Theory
  {
    property assoc(**)
      := forall x,y,z in U. (x**(y**z))=(x**y)**z
  };;
```

Commutativity

Definition 14 A binary operation T on a carrier S is said to be *commutative* if:

```
Commutativity = Theory
  {
    property comm(**) := forall x,y in U. (x**y)=y**x
  };;
```

Idempotency

Definition 15 A binary operation T on a carrier S is said to be *idempotent* if:

```
Idempotency = Theory
  {
    property idem(**) := forall x in U. (x**x)=x
  };;
```

Left Absorption

Definition 16 Given a carrier U and two binary operations $**$ and $++$ on U, then the operation is said to be *left-absorptive* over $++$ if:

```
LeftAbsorption = Theory
  {
    property labsor(**, ++)
      := forall x,y in U. ((x**(x++y))=x) and ((x++(x**y))=x)
  };;
```

Right Absorption

Definition 17 Given a carrier U and two binary operations $**$ and $++$ on U, then the operation is said to be *right-absorptive* over $++$ if:

```
RightAbsorption = Theory
  {
    property rabsor(**, ++)
      := forall x,y in U. (((x++y)**y)=y) and (((x**y)++y)=y)
  };;
```

Absorption

Definition 18 Given a carrier and two binary operations, then the operation is said to be *absorptive* over $++$ if it is both *left-absorptive* and *right-absorptive.*

```
Absorption = Theory
  {
    import LeftAbsorption, RightAbsorption;
    property absorption(**, ++)
      := (labsor(**,++)) and (rabsor(**,++))
  };;
```

Left Distributivity

Definition 19 Given a carrier U and two binary operations $**$ and $++$ on U, then the operation is said to be *left-distributive* over $++$ if:

```
LeftDistributivity = Theory
  {
    property ldistri(**, ++)
      := forall x,y,z in U. (x**(y++z))=(x**y)++(x**z)
  };;
```

Right Distributivity

Definition 20 Given a carrier U and two binary operations $**$ and $++$ on U, then the operation is said to be *right-distributive* over *++* if:

```
RightDistributivity = Theory
  {
    property rdistri(**, ++)
      := forall x,y,z in U. ((y++z)**x)=(y**x)++(z**x)
  };;
```

Distributivity

Definition 21 Given a carrier and two binary operations, then the operation is said to be *distributive* over *++* if it is both *left-distributive* and *right-distributive*.

```
Distributivity = Theory
  {
    import LeftDistributivity, RightDistributivity;
    property distri(**, ++)
      := (ldistri(**, ++)) and (rdistri(**, ++))
  };;
```

Unipotency

Definition 22 A binary operation T on a carrier U is said to be *unipotent* if:

```
Unipotency = Theory
  {
    property unipot(**) := forall x,y in U. (x**x)=y**y
  };;
```

Left Identity

Definition 23 An element e of a pointed carrier U with a binary operation T is called a *left identity* if:

```
LeftIdentity = Theory
  {
    property lident(e,(**)) := forall x in U. (e**x)=x
  };;
```

Right Identity

Definition 24 An element e of a pointed carrier U with a binary operation T is called a *right identity* if:

```
RightIdentity = Theory
  {
    property rident(e,(**)) := forall x in U. (x**e)=x
  };;
```

Identity

Definition 25 A binary operation T on a carrier U is said to be *idempotent* if: An element e of a pointed carrier U with a binary operation T is called a *two-sided identity*, or simply an *identity* if:

```
Identity = Theory
  {
    import LeftIdentity, RightIdentity;
    property identity(a, (++))
      := (lident(a, (++))) and (rident(a, (++)))
  };;
```

Left Inverse

Definition 26 Let U be a pointed carrier with a binary operations T, then an element x is said to be a *left inverse* if:

```
LeftInverse = Theory
  {
    property linv(a, (**)) := forall x,y in U. (x**y)=a
  };;
```

Right Inverse

Definition 27 Let U be a pointed carrier with a binary operations T, then an element x is said to be a *right inverse* if:

```
RightInverse = Theory
  {
    property rinv(a, (**)) := forall x,y in U. (y**x)=a
  };;
```

Inverse

Definition 28 Let S be a pointed carrier with a binary operations T, if an element x is both a *left inverse* and a *right inverse* of y, then x is said to be a *two-sided inverse*, or simply an *inverse*, of y if:

```
Inverse = Theory
  {
    import LeftInverse, RightInverse;
    property inverse(a, (**))
      := (linv(a, (**))) and (rinv(a, (**)))
  };;
```

Antisymmetry

Definition 29 We say a relation R on a carrier U is *antisymmetric* if:

```
Antisymmetry = Theory
  {
    property antisym(R)
      := forall x,y in U.((x ‘R y) and (y ‘R x)) implies (x = y)
  };;
```

Asymmetry

Definition 30 We say a relation R on a carrier U is *asymmetric* if:

```
Asymmetry = Theory
  {
    property asym(R)
      := forall x,y in U. not(x 'R y implies y 'R x)
  };;
```

Symmetry

Definition 31 We say a relation R on a carrier U is *symmetric* if:

```
Symmetry = Theory
  {
    property sym(R) := forall x,y in U. x 'R y implies y 'R x
  };;
```

Transitivity

Definition 32 We say a unary relation R on a carrier U is *transitive* if:

```
Transitivity = Theory
  {
    property trans(R)
      := forall x,y,z in U.
        ((x 'R y) and (y 'R z)) implies x 'R z
  };;
```

Function

Definition 33 A *Function* on two carriers U and V is a mapping f from elements of U to elements of V.

```
Function = CarrierType extended by
  {
    U:carrier;
    B:carrier;
    f:U->V
  };;
```

Reflexivity

Definition 34 We say a relation R on a carrier U is *reflexive* if:

```
Reflexivity = Theory
  {
    property refl(R) := forall x in U. x 'R x
  };;
```

Irreflexivity

Definition 35 We say a relation R on a carrier U is *irreflexive* if:

```
Irreflexivity = Theory
  {
    property irrefl(R) := forall x in U. not(x 'R x)
  };;
```

Magma

Definition 36 A *Magma* consists of a carrier S equipped with a single binary operation T. A binary operation is closed by definition, but no other axioms are imposed on the operation. In abstract algebra, a magma is a basic and very important kind of algebraic structure.

```
Magma = Theory
  {
    BinaryOperation with ** = *
  };;
```

PointedMagma

Definition 37 A *PointedMagma* consists of a magma and a pointed carrier along with a supported carrier.

```
PointedMagma = Theory
  {
    combines Magma, PointedCarrier along Carrier
  };;
```

Left Cancellative

Definition 38 Let U be a carrier with a binary operations T, an element z is *left cancellative* if:

```
LeftCancellative = Theory
  {
    property lcancel(**)
      := forall x,y,z in U. ((z**x)=z**y) implies x=y
  };;
```

Right Cancellative

Definition 39 Let U be a carrier with a binary operations T, an element z is *right-cancellative* if:

```
RightCancellative = Theory
  {
    property rcancel(**)
      := forall x,y,z in U. ((x**z)=y**z) implies x=y
  };;
```

Cancellative

Definition 40 Let S be a carrier with a binary operations T, i.e. a *Magma*, an element z is *cancellative* if it is both *left-cancellative* and *right-cancellative*.

```
Cancellative = Theory
  {
    import LeftCancellative, RightCancellative;
    property cancellative(**) := (lcancel(**)) and (rcancel(**))
  };;
```

Unital

Definition 41 A *Unital* is a magma with an identity element.

```
Unital = PointedMagma extended by
  {
    import Identity;
    axiom identity(e,(*))
  };;
```

QuasiGroup

Definition 42 A *QuasiGroup* is a cancellative magma.

```
QuasiGroup = Magma extended by
  {
    import Cancellative;
    axiom cancellative(*)
  };;
```

Loop

Definition 43 A *Loop* is a quasigroup with an identity element.

```
Loop = Theory
  {
    combines Unital, QuasiGroup along Magma
  };;
```

SemiGroup

Definition 44 A *Semigroup* is an associative magma.

```
SemiGroup = Magma extended by
  {
    import Associativity;
    axiom assoc(*)
  };;
```

Band

Definition 45 A *Band* is a semigroup of idempotents.

```
Band = SemiGroup extended by
  {
    import Idempotency;
    axiom idem(*)
  };;
```

Group

Definition 46 A *Group* is a loop with associative magma.

```
Group = Loop extended by
  {
    import Associativity;
    axiom assoc(*)
  };;
```

Abelian Group

Definition 47 An *AbelianGroup* is a commutative group.

```
AbelianGroup = Theory
  {
    Group with * = +, e = 0;
    import Commutativity;
    axiom comm(+)
  };;
```

Monoid

Definition 48 A *Monoid* is a unital semigroup.

```
Monoid = Theory
  {
    combines Unital, SemiGroup along Magma
  };;
```

Commutative Monoid

Definition 49 A *CommutativeMonoid* is a monoid whose operation is commutative.

```
CommutativeMonoid = Monoid extended by
  {
    import Commutativity;
    axiom comm(*)
  };;
```

Ring

Definition 50 A *Ring* is a carrier equipped with an Abelian group under addition; a Monoid under multiplication; and satisfying the Distributivity law of multiplication over addition.

```
Ring = Theory
  {
    Import Distributivity;
    S1 := AbelianGroup;
    S2 := Monoid with e = 1;
    combines S1, S2 along S2;
    axiom distri( *, + );
  };;
```

Commutative Ring

Definition 51 A *CommutativeRing* is a ring with commutative multiplication.

```
CommutativeRing = Ring extended by
  {
    import Commutativity;
    axiom comm(*)
  };;
```