Algebraic Constructions Applied to Theories

Algebraic Constructions Applied to Theories

By
Quang Minh Tran, Dipl.-Inf.

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 (2011) McMaster University
(Computer Science) Hamilton, Ontario

TITLE: Algebraic Constructions Applied to Theories
AUTHOR: Quang Minh Tran, Dipl.-Inf. (Furtwangen University, Germany)

SUPERVISOR: Dr. William M. Farmer

NUMBER OF PAGES: x, 125

ABSTRACT

MathScheme is a long-range research project being conducted at McMaster University with the aim to develop a mechanized mathematics system in which formal deduction and symbolic computation are integrated from the lowest level. The novel notion of a biform theory that is a combination of an axiomatic theory and an algorithmic theory is used to integrate formal deduction and symbolic computation into a uniform theory. A major focus of the project has currently been on building a library of formalized mathematics called the MathScheme Library. The MathScheme Library is based on the little theories method in which a portion of mathematical knowledge is represented as a network of biform theories interconnected via theory morphisms. In this thesis, we describe a systematic explanation of the underlying techniques which have been used for the construction of the MathScheme Library. Then we describe several algebraic constructions that can derive new useful machinery by leveraging the information extracted from a theory. For instance, we show a construction that can reify the term algebra of a (possibly multi-sorted) theory as an inductive data type.

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Mathematics is undoubtedly one of the most crucial tools for science and engineering. The scientific and technological advancement we are enjoying today is largely, directly or indirectly, driven by mathematics. The importance and immense size of mathematics has led to the need for *mechanizing mathematics*, namely the development of software systems that assist the user in using mathematics.

The purpose of this chapter is to give a brief introduction to the mechanizing mathematics project entitled MathScheme as well as the focus and organization of this thesis. We start the chapter by explaining the *mathematics process*, that is, the process-oriented way of perceiving mathematics.

We compare theorem proving systems and computer algebra systems. Then we introduce the MathScheme project, whose main goal is to combine the functionalities of theorem proving systems and computer algebra systems in one system. The objective and goals of the project are stated and explained.

We give an overview of the solution pieces which have been developed to achieve the project's goals. Finally, we discuss the scope of this thesis as well as how it is organized.

1.1 The Mathematics Process

So what is mathematics? The common definition of mathematics is an enormous body of knowledge containing definitions and facts. However, such a descriptive view

of mathematics does not capture the process by which mathematical knowledge is produced. Since we are concerned with developing a mechanized mathematics system, we are extremely interested in the mathematics process. In [13], Dr. Farmer defines mathematics as a "*process* of creation, exploration, and connection" which consists of three intertwined activities:

(1) *Model creation.* Mathematical models representing mathematical aspects of the world are created.

(2) *Model exploration.* The models are explored by stating and proving conjectures and by performing computations.

(3) *Model connection.* The models are connected to each other so that results obtained in one model can be used in other related models.

The mathematics process has produced an immense body of mathematical knowledge that is constantly being enlarged. Mathematical knowledge in turn provides materials for the mathematics process to create new knowledge.

The importance and immense size of mathematics naturally lead to the need for the development of *mechanized mathematics systems* (MMSs), software systems that assist the user in doing mathematics. We believe that a mechanized mathematics system should ideally support all of these three activities of the mathematics process.

1.2 Theorem Proving Systems vs. Computer Algebra Systems

Contemporary mathematics software systems can be categorized into two major groups: theorem proving systems and computer algebra systems.

Theorem proving systems emphasize proving conjectures. A theorem proving system is usually built upon the *axiomatic method* which will be discussed in more detail in Chapter 2. Essentially, the axiomatic method is applied with an underlying formal logic in which mathematical knowledge is formalized as axiomatic theories. A conjecture is expressed as a formula in some axiomatic theory and then one tries to prove it from the axioms of the theory by applying the inference rules of the logic. This proving process is also referred to as *formal deduction.* The biggest advantage of a theorem proving system is that, since theorems and their proofs are well-grounded

in a formal logic, they can be mechanically checked. Moreover, the proving process can also be assisted by the use of computers. Consequently, the confidence in the correctness of the proofs increases significantly.

On the other hand, computer algebra systems emphasize performing computation. A computer algebra system implements *symbolic computation*, i.e. implements a collection of algorithms for manipulating expressions. An algorithm takes an expression as input, symbolically manipulates it and returns the result. As opposed to theorem proving systems, computer algebra systems usually do not have a formal underlying logic. As a result, the behavior of algorithms cannot usually be easily formalized and reasoned about. Furthermore, stating and proving conjectures in a computer algebra system is usually not supported or even possible.

Formal deduction and symbolic computation are two intertwined aspects of mathematics. The unnatural separation of them into two different kinds of systems severely limits the ability of doing mathematics in either kind of system. In responding to the problem, instead of trying to add formal deduction to an existing computer algebra system or symbolic computation to a theorem proving system as an afterthought, we follow the more radical approach, namely, developing a new system in which formal deduction and symbolic computation are integrated from the ground up.

1.3 The MathScheme Project

The MathScheme Project [4] is a long-range research project lead by Dr. Carette and Dr. Farmer at McMaster University with the following stated objective [18]:

The Objective of MathScheme: *The objective of the project is to develop a new approach to mechanized mathematics in which computer algebra and computer theorem proving are integrated at the lowest level.*

Toward that end, the project has five goals [10]:

Goal 1 *Design a* framework *for integrating formal deduction and symbolic computation.*

As mentioned previously, formal deduction is the core functionality of a theorem proving system and symbolic computation is the core functionality of a computer

algebra system. In order to integrate formal deduction and symbolic computation, a framework needs to be developed that treats these in a uniform manner.

Goal 2 *Design and implement a* logic *that, among other things, supports reasoning about the syntax of expressions.*

Since symbolic computation is about manipulating the syntax of expressions, the integration of formal deduction and symbolic computation inevitably requires reasoning about syntactic expressions. Most contemporary logics such as first-order logic, ZF set theory, simple type theory etc. do not directly support reasoning about syntax and thus cannot be easily used as the underlying logic for our new system. The solution for this is to design and implement a new formal logic that supports reasoning about the syntax of expressions.

Goal 3 *Build a* library of formalized mathematics.

A framework for the integration of formal deduction and symbolic computation and a logic supporting reasoning about the syntax of expressions would provide an adequate infrastructure for building a library of formalized mathematics. For instance, the library would contain formalizations of algebraic structures, natural numbers, real numbers and lots of other mathematical structures. The unique characteristic of the library is that mathematical knowledge is expressed both axiomatically and algorithmically and is stored side-by-side.

Goal 4 *Build a* mechanized mathematics system *based on this framework, logic and library.*

The next step is to leverage the framework, logic and library to build a mechanized mathematics system. The system should be easy to use with a graphical user interface and visualization support, among other things. Most importantly, it should possess the power both of a theorem proving system and of a computer algebra system.

Goal 5 *Build an* interactive mathematics laboratory *on top of the mechanized mathematics system.*

This is a long-term vision proposed by Dr. Farmer in [7]. In the paper, he defined an interactive mathematics laboratory as "a computer system with a set of integrated tools designed to facilitate the mathematics process". Such a laboratory, once realized, would provide an interactive environment for the user, especially students, to create and explore mathematics and possibly "revolutionize mathematics education" [7, page 5].

1.4 Overview of Solution Pieces

This section gives an overview of the solution pieces that have been developed within the scope of MathScheme that address the goals mentioned previously.

Solution Piece 1 *The notion of a biform theory [8] is used to integrate formal deduction and symbolic computation. A biform theory is a generalization of an axiomatic theory and an algorithmic theory. We discuss axiomatic and algorithmic theories in Chapter 2.*

This solution piece addresses Goal 1. We discuss biform theories in Chapter 2.

Solution Piece 2 *A formal logic called* Chiron *[9] that supports reasoning about the syntax of expressions via quotation and evaluation has been developed.*

This solution piece addresses Goal 2. Formalizing biform theories requires a logic with support for reasoning about syntax [8]. Since Chiron supports reasoning about the syntax of expressions, it is well suited for formalizing biform theories [9].

Solution Piece 3 *A high-level specification language called the* MathScheme Language *(MSL) has been developed on top of Chiron. MSL is used for specifying and relating theories in a library of formalized mathematics.*

This solution piece addresses Goal 3. The motivation for having MSL is that formalizing biform theories directly in Chiron can be very verbose because Chiron is a low-level logic. MSL, which can be seen as high-level syntactic sugar for Chiron, is more convenient for specifying and relating theories when building the library of formalized mathematics. The Appendix contains the language description of MSL.

Solution Piece 4 *A library of formalized mathematics called the* MathScheme Library *has been developed. It contains, among other things, formalization of abstract algebra and basic data structures in MSL.*

This solution piece also addresses Goal 3. We discuss the MathScheme Library in Chapter 3.

Solution Piece 5 *There is an implementation of Chiron and MSL as well as a partial translation from MSL to Chiron in* OCaml *[1].*

This solution piece addresses Goal 4. This implementation will eventually become the core of the MathScheme mechanized mathematics system.

1.5 Contributions of the Thesis

The contribution of this thesis is twofold. First, we explain the techniques that have been developed and used to construct the MathScheme Library. In particular, we explain biform theories and the little theories method that are the key techniques behind the MathScheme Library. We give the definition of a theory morphism as well as several operations involving them.

Then we explain several algebraic constructions that construct new useful machinery by leveraging the information extracted from theories. In particular, we explain constructions for (1) reifying a theory as a dependent record type and a theory interpretation as a member of a dependent record type, (2) generating a theory of homomorphisms (as well as epimorphisms, monomorphisms, isomorphisms) from an existing theory, (3) generating a theory of substructures from an existing theory and (4) reifying the term algebra of a theory as an inductive data type as well as generating other useful syntactic machinery.

1.6 Organization of the Thesis

The thesis is organized as follows. Chapter 2 introduces axiomatic theories as building blocks for constructing a library of formalized mathematics. Here, the novel structure of a biform theory is introduced which is an extended version of an axiomatic theory augmented with symbolic computation. Different approaches to organizing mathematics such as the big theories method, the little theories method etc. are discussed.

Chapter 3 explains the library of formalized mathematics called the MathScheme library that has been developed within the scope of the MathScheme project. The requirements and design goals of the library are discussed. The formalizations of abstract algebra and concrete theories in the library are briefly explained.

Chapter 4 explains the algebraic constructions for reifying a theory as a dependent record type and a theory interpretation as an element of a dependent record type, respectively.

Chapter 5 explains the algebraic construction for deriving the notion of a homomorphism from an existing theory. Two generation approaches are discussed: one using the reification of theories as types developed in Chapter 4 and one using the pushout of theory morphisms introduced in Chapter 2.

Chapter 6 explains the algebraic construction for deriving the notion of a substruc-

ture from an existing theory. Also two generation approaches are discussed: one using the reification of theories as types and one using the pushout of theory morphisms.

Chapter 7 explains the algebraic construction for reifying the term algebra of a thery as an inductive data type. Here, the notion of a syntax framework is introduced and used to analyze the presented reification purpose.

Chapter 8 concludes the thesis and discusses possible future work.

1.7 Font Convention

The following font conventions are used in the thesis:

- *Italics* : for a term that is being defined in a definition.
- Sans serif: for expressions of the MathScheme Language.
- Bold sans serif: for keywords of the MathScheme Language.

CHAPTER 2

THEORIES OF FORMALIZED MATHEMATICS

Constructing a library of formalized mathematics is one of the biggest challenges of mechanizing mathematics. A library of formalized mathematics is where mathematical knowledge is organized and stored and thus can be seen as the heart of a mechanized mathematics system. This chapter aims to give a brief overview of the existing techniques and our techniques that have been developed and used for this purpose. In particular, we review the axiomatic method, introduce biform theories as well as morphisms between them. Approaches to organizing mathematical knowledge, most notably the little theories method, are discussed.

2.1 Axiomatic Method

Mathematical knowledge is an enormous network of definitions and facts that are related to each other in some way. Some mathematical concept may be seen as an extension of another concept. For instance, the concept of a monoid can be regarded as the concept of a semigroup augmented with an identity element. Some mathematical concepts can also be seen as a combination of several other concepts. A vector space has both the concept of a field and the concept of a vector in it.

Representing mathematical knowledge in a computer system appears to be an impossible task at first since a lot of mathematical concepts are infinite. How could

we store all (uncountably many) reals or even naturals in a computer? The solution to this question is a method called the *axiomatic method*.

The axiomatic method was first used by Euclid in his work on presenting the mathematics of his time in his series of books called *Elements*. In particular, in presenting euclidean geometry, he carefully chose a small set of assumptions called axioms which are assumed to be intuitively true, e.g. the parallel axiom. All further theorems are derived from these axioms. In the 90s, A. N. Whitehead and B. Russell axiomatized a portion of mathematics in the *Principia Mathematica* [25].

Basically, in the axiomatic method, a mathematical concept is formalized as an *axiomatic theory* in some formal logic, a process referred to as *axiomatization*. Conjectures are stated as formulas and proved by applying the inference rules of the logic.

The majority of theorem proving systems, most notably Isabelle [22], IMPS [11] etc. are based on the axiomatic method. Specifically, most of them are built upon either first-order logic, set theory or higher-order type theory. Mathematical concepts are formalized as axiomatic theories in the chosen logic.

2.1.1 Axiomatic Theory

The following gives the formal definition of an axiomatic theory:

Definition 2.1.1 (Axiomatic Theory) Given a logic, an axiomatic theory is a pair (L, Γ) in which

- L is a *language* (set of concepts) of the logic.
- Γ is a set of formulas formalized in the logic called *axioms*.

Example 2.1.2 The natural number can be axiomatized as an axiomatic theory of Peano Arithmetic where the language $L = \{nat, zero, suc\}$ and $\Gamma = \{A_1, A_2, A_3\}$.

In MSL, it may look as below:

```
Nat := Theory {
 Concepts
  nat  : type;
  zero : nat;
  suc  : nat -> nat;
```

```
 Facts
  axiom ax1 : forall x : nat . not (zero = suc(x));
  axiom ax2 : forall x, y : nat . (suc(x) = suc(y))
              implies (x = y);
  axiom ax3 : forall p : nat? .
     (p(zero) and (forall n : nat . p(n) => p(suc(n)))
              => forall n : p(n)) ;
}
```

□

Example 2.1.3 The concept of monoid can be represented by the following axiomatic theory expressed in MSL where $L = \{\text{M}, *, \text{e}\}$ is the language of monoid whereas the set of axioms $\Gamma = \{\text{associativity\_*\_}, \text{identity\_e\_}\}$ specifies the monoid axioms:

```
Monoid := Theory {
   M : type;
   * : (M,M) -> M;
   e : M;
   axiom associativity_*_ : associative((*));
   axiom identity_e_      : identity((*),e);
}
```

□

An axiomatic theory can be seen as a specification of its models. It captures one model or a collection of models by specifying its or their concepts in the language part and their required properties in the axiom part.

We distinguish between two types of theories, categorical and non-categorical theories:

Definition 2.1.4 A theory having exactly one model up to isomorphism is called a *categorical theory*. A theory that is not categorical is called a *non-categorical theory*.

The structure of natural numbers $\mathcal{N} = (\mathbb{N}, 0, suc)$ as well as all structures isomorphic with it are models of Nat. Furthermore, any other structure that is not isomorphic with $\mathcal{N}$ is not a model of Nat. As a result, Nat is a categorical theory. Here, the intention of Nat is to describe a single model, the natural numbers.

On the contrary, Monoid describes a collection of non-isomorphic models. For instance, both non-isomorphic structures $(\mathbb{N}, +, 0)$ and $(\mathbb{R}, *, 1)$ are models of Monoid. Therefore, Monoid is non-categorical.

2.1.2 Theory Development Process

According to Dr. Farmer's CAS 760 lecture notes [15] , the mathematics process, as mentioned in Chapter 1, can be regarded as *the theory development process* where theory creation corresponds to model creation, theory exploration corresponds to model exploration and theory connection corresponds to model connection. In the following sections, we briefly explain these three activities of the theory development process.

Theory Creation

Also according to [15], an axiomatic theory can be constructed by:

- Building it from scratch.
- Extending an existing theory (theory extension).
- Renaming an existing theory (theory renaming).
- Combining several existing theories (theory combination/union).
- Instantiating a parameterized theory (theory instantiation).

It is noteworthy that these theory constructions are actually syntactic operations on theories.

Build from Scratch Building a theory from scratch is the most straightforward way of creating a theory. We start from the empty theory and add concepts and axioms that describe the mathematical concepts we are interested in.

Theory Extension Theory extension is analogous to inheritance in object-oriented programming in the sense that a theory extension extends an existing theory with further conceptual units.

Definition 2.1.5 Formally, let $T_1 = (L_1, \Gamma_1)$ and $T_2 = (L_2, \Gamma_2)$ be two theories. T_2 is a *theory extension* of T_1 (and T_1 is a *subtheory* of T_2) if $L_1 \subseteq L_2$ and $\Gamma_1 \subseteq \Gamma_2$.

Intuitively, the theory extension T_2 is the obtained by adding new machinery, i.e. vocabulary and axioms, to T_1.

For instance, if we extend the concept of a semigroup with the property that its binary operation has an identity element we have the concept of a monoid. Consequently, the theory of monoid is the theory extension of the theory of semigroup with the added identity element.

In MSL, this may look as follows:

```
Semigroup := Theory {
  M : type;
  * : (M,M) -> M;
  axiom associative_* : associative((*));
}
Monoid := Semigroup extended by {
  e : M;
  axiom identity_*_ : leftIdentity((*), e) and
                      rightIdentity((*), e);
}
```

Theory Renaming Theory renaming creates a new theory from an existing theory by renaming some or all of its primitive symbols. Theory renaming helps make the language of a theory more intuitive and convenient for the user.

For instance, let BinaryRelation be the theory of a binary relation:

```
Empty := Theory {}
Carrier := Empty extended by {
  U : type
}
BinaryRelation := Carrier extended by {
  R : (U, U)?
}
```

Here, R : (U, U)? represents a predicate, R : (U, U) −> Bool, in the base logic.

Now, we want to define a theory of an ordered relation capturing the mathematical structure (U, <=). It is obvious that the theory BinaryRelation has almost everything

we need: a carrier set and a binary relation. However, we would like to call the binary relation <= to emphasize the ordering instead of the general notation of relation R. That is precisely what theory renaming is good for. Using theory renaming, OrderRelation is the theory resulted from renaming R to <= in BinaryRelation.
In MSL, it may look as below:

```
OrderRelation := BinaryRelation[ R |-> <= ]
```

In expanded form, it is equivalent to:

```
OrderRelation := Theory {
  U : type;
  <= : (U, U)?;
}
```

Theory Combination Theory combination is the most sophisticated way of creating a new theory. Basically, a theory combination generates a new theory by combining a list of theories over a common subtheory. The common subtheory specifies the part commonly occurring in the combined theories that we would like to have only one copy in the resulting theory.

For instance, let ReflexiveOrderRelation be the theory of an order relation being reflexive and TransitiveOrderRelation be the theory of an order relation being transitive:

```
ReflexiveOrderRelation := OrderRelation extended by {
   axiom forall x : U . x <= x
}
TransitiveOrderRelation := OrderRelation extended by {
   axiom forall x, y, z : U . (x <= y and y <= z)
         implies x <= z
}
```

The theory Preorder can be defined by combining ReflexiveOrderRelation and TransitiveOrderRelation over OrderRelation as illustrated in Figure 2.1.

In MSL, Preorder would look as follows:

```
Preorder :=
   combine ReflexiveOrderRelation, TransitiveOrderRelation
   over OrderRelation
```

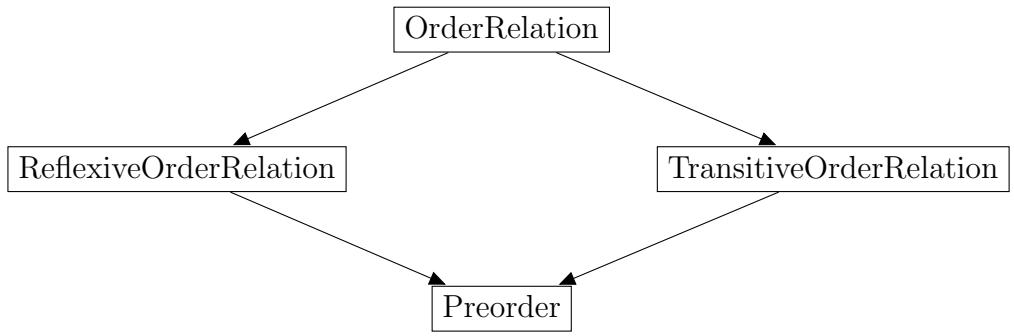

Figure 2.1: Example of a Theory Combination

And in the expanded form, it is:

```
Preorder := Theory {
   U  : type;
   <= : (U, U)?;
   axiom forall x : U . x <= x;
   axiom forall x, y, z : U . (x <= y and y <= z implies x <= z);
}
```

Notice that the common part U and <= in ReflexiveOrderRelation and TransitiveOrderRelation are not duplicated (because they occur in the common subtheory OrderRelation), but only one declaration for each of them is transported to the combined theory.

Theory Instantiation A theory can be constructed by instantiating a parameterized theory. This way of constructing a theory is actually not relevant to this thesis. Nevertheless, for the sake of completeness, it is briefly explained here.

For instance, the following parameterized theory/functor Comm, introduced by Jian Xu in [26], adds the commutative axiom to any theory with a binary operation over a set. Since parameterized theories are not yet implemented in MSL, we use pseudo code to declare Comm:

```
Comm := Theory (T : theory_type {ele : type; * : (C, C) -> C})
        extended by {
  axiom : forall x, y : ele . x * y = y * x;
}
```

The theory of Monoid defined previously is compatible to the theory signature required by Comm since it has a binary operation over a set. Applying Comm to Monoid, we get the theory of a commutative monoid:

```
CommMonoid := Theory {
  U : type;
  * : (U,U) -> U;
  e : M;
  axiom associativity_*_ : associative((*));
  axiom identity_*_ : leftIdentity((*), e)
        and rightIdentity((*), e);
  axiom forall x, y : U . x * y = y * x;
}
```

The MSL syntax for applying a parameterized theory to a concrete theory is theory instantiation (See Appendix).

Theory Exploration

Exploring an axiomatic theory means stating conjectures in form of formulas and trying to prove them by applying inference rules.

Theory Connection

Connecting theories is about relating theories to each other so theorems proven in one theory can be reused in another related theory. The main tool for theory connection is *theory interpretation*.

Since theory interpretation is based on theory translation, we first introduce theory translation.

Definition 2.1.6 A *theory translation* is a function Φ from a theory T_1 to another theory T_2 that maps the primitive symbols of T_1 to expressions of T_2 satisfying certain syntactic conditions.

Definition 2.1.7 A theory translation Φ from T_1 to T_2 is called a *theory interpretation* if for all formulas A of T_1 such that $\Phi(A)$ is defined, $T_1 \models A \Rightarrow T_2 \models \Phi(A)$.

Informally, Φ maps the logical consequences of T_1 to consequences of T_2 and thus it can be seen as a semantics-preserving theory translation.

In practice, this condition is usually very hard or even impossible to directly verify since there can be infinitely many theorems. Instead, normally the following lemma is used, provided things are set up properly:

Lemma 1 *Let Φ be a theory translation from a theory T_1 to a theory T_2. Φ is a theory interpretation if and only it maps all the obligations to logical consequences of T_2.*

Certainly, the definitions of theory translation and theory interpretation above, particularly the phrases "satisfying certain syntactic conditions" and "maps all the obligations into logical consequences", are very vague. Actually, we cannot give a precise definition of theory interpretation because such a definition varies from logic to logic. [6] is an excellent paper to understand theory interpretation. It is noteworthy that there can be more than one theory interpretation from one theory to another theory.

Example 2.1.8 Suppose we have a theory of natural numbers:

```
Nat := Theory {
  nat  : type;
  zero : nat;
  suc  : nat -> nat;
  +    : (nat,nat) -> nat
  ...
}
```

We define $\Phi = [\mathrm{M} \mapsto \mathrm{nat}, \mathrm{e} \mapsto \mathrm{zero}, * \mapsto +]$ to be a theory translation from Monoid to Nat (the definition of Monoid is given previously in this chapter). Furthermore, the proof obligations (1) + is associative and (2) zero is the identity of + are theorems of Nat. Consequently, Φ is a theory interpretation from Monoid to Nat.

Φ establishes a connection between Monoid and Nat in the sense that it tells us how to interpret the model $(nat, zero, +)$ of Nat as a monoid. □

2.1.3 Theories as Modules

It turns out that theories closely resemble modules. Jian Xu, in his PhD thesis [26], developed a module system named *Mei* for organizing mathematical knowledge

by combining and further developing good ideas from module systems used in ML-languages and specification languages. In Mei, theories can be extended, combined and related to each other in much the same way as modules.

2.2 Symbolic Computation

Axiomatic theories are excellent for representing mathematical structures, algebras and data types etc. in a declarative way. Nevertheless, they are not suitable for representing symbolic computation. Basically, symbolic computation is the aspect of mathematics in which mathematical expressions are transformed in a symbolic way. Transformation algorithms are implemented by programs that take expressions as input and return the transformed expression. Computer algebra systems, for example Maple [16] and Axiom [17] , implement symbolic computation.

A set of algorithms that manipulate expressions can be formalized as an algorithmic theory. But first, we formalize the notion of an algorithm. In MathScheme, an algorithm is formalized as a transformer.

Definition 2.2.1 Formally, a *transformer* is a pair $(\pi, \hat{\pi})$ in which

- π is a function that maps a list of expressions to an expression, i.e. $\pi : (E_1, \ldots, E_n) \to E_{n+1}$ where E_i are types of expressions and $n \geq 0$.
- $\hat{\pi}$ is the associated program that implements an algorithm for realizing π.

For instance, the derivative rule $\frac{d(u.v)}{dx} = \frac{du}{dx}.v + u.\frac{dv}{dx}$ can be represented by a transformer $\mathsf{prod\text{-}diff} : \mathsf{DerivExpr} \to \mathsf{DerivExpr}$ that maps an expression of the form $\frac{d(u.v)}{dx}$ to the expression $\frac{du}{dx}.v + u.\frac{dv}{dx}$. Corresponding to $\mathsf{prod\text{-}diff}$ is a program implementing the algorithm of the derivative rule.

Definition 2.2.2 An *algorithmic theory* is a pair $(L, \mathcal{T})$ where L is a language and $\mathcal{T}$ is a set of transformers.

2.3 Biform Theory = Axiomatic Theory + Algorithmic Theory

So far, we have seen that axiomatic theories and algorithmic theories capture two key aspects of mathematics: formal deduction and symbolic computation. These two

aspects are often intertwined. Symbolic computation may require formal deduction and vice versa.

Unfortunately, as we already mentioned before, despite their close relationship, formal deduction and symbolic computation are treated separately in contemporary systems. Theorem proving systems focus on formal deduction while computer algebra systems focus on symbolic computation.

In MathScheme, the integration of formal deduction and symbolic computation is done via a novel structure called a *biform theory* which essentially merges an axiomatic theory and an algorithmic theory. Several papers, most notably [8], are devoted to explaining biform theories. It is noteworthy that the precise definition of a biform theory varies in the papers. Perhaps, the simplest definition is the following:

Definition 2.3.1 A biform theory is a triple $(L, \mathcal{T}, \Gamma)$ in which:

- L is a set of concepts called a language.
- $\mathcal{T}$ is a set of transformers.
- Γ is a set of facts that are statements about the concepts and transformers. Since facts can also be statements about transformers, they can specify programs.

A program can be either written in a programming language such as OCaml, Java, C++ etc. or expressed directly, for example as a lambda term, in the logic in which the biform theory is formalized.

If the program is implemented outside of the logic, we have to treat it as a black box. However, the specification about the program stated using the facts in Γ still allows us to reason about the properties of the program.

If the program itself is expressed in the logic, then we can reason about the implemented algorithm and may eventually be able to formally prove its correctness.

An axiomatic theory and an algorithmic theory can be seen as a special case of a biform theory. If a biform theory has no transformers, i.e. $\mathcal{T}$ is empty, it reduces to an axiomatic theory which contains only an axiomatization but no symbolic computation. Conversely, if a biform theory has no facts, i.e. Γ is empty, it reduces to an algorithmic theory which contains only a set of transformers.

Formalizing transformers in a biform theory requires the ability to represent and reason about the syntax of expressions. Since traditional logics do not directly support the ability to reason about the syntax of expressions, they are not suitable for

formalizing biform theories. This motivated Dr. Farmer to develop a new formal logic called Chiron [9]. In Chiron, one can refer to syntactic expressions and reason about them via quotation and evaluation.

Formalizing biform theories directly in Chiron could be, however, very verbose since Chiron is very low-level. This is the motivation for the MathScheme Language, a high-level language defined on top of Chiron, which is more convenient for expressing biform theories.

From now on, we use the word theory and biform theory interchangeably.

2.4 Theory Morphisms

As we mentioned previously, the axiomatic method allows us to represent mathematical knowledge as theories. Theories can be constructed from scratch or from other theories as discussed in 2.1.2. One thing we notice is that a theory only contains the concepts and axioms of the mathematical concept being formalized but not the information about the construction path leading to it.

For instance, when looking at the theory of group, the only information we know is that it contains a carrier set, an associative binary operation, an identity element and an inverse operation. This theory may have been constructed by extending a monoid with an inverse operation which is in turn constructed from a semigroup. It could also be the result of extending semigroup directly by adding an identity element and inverse function.

The theory interpretations between theories turn out to contain lots of useful information, especially on how theories are constructed. This is one of the reasons that, in MathScheme, we are currently experimenting with the idea of placing more emphasis on these theory interpretations between theories instead of on the theories themselves.

Definition 2.4.1 We call a theory interpretation (see Definition 2.1.2) a *theory morphism*.

Informally, a theory morphism is a triple (T, T', Φ) consisting of a source theory T, a target theory T' and a semantics-preserving mapping Φ from T to T'.

$$\boxed{\text{Semigroup}} \xrightarrow{\Phi = [U \mapsto U, * \mapsto *]} \boxed{\text{Monoid}}$$

Figure 2.2: Example of an Identity Injection

2.4.1 Injections

There is a special kind of theory morphism which we call an *injection*. Basically, an injection is a theory morphism (T, T', Φ) in which Φ injectively maps the primitive symbols of T to primitive symbols (and not expressions) of T'.

Intuitively, an injection defines an embedding of T into T'. There are two kinds of injections:

- Identity injection.
- Renaming injection.

An identity injection is a theory morphism in which Φ maps each symbol of T to exactly the same symbol in T'. Such an injection can only exist if T' is either identical to, or a theory extension, of T.

A renaming injection maps symbols between two isomorphic structures T and T' by renaming the symbols in T to match those in T'.

From now on, we use the notation $[c_1 \mapsto c'_1, \ldots, c_n \mapsto c'_n]$ to express an injection mapping c_i of the source theory to c'_i of the target theory. Moreover, $[\,]$ denotes the empty injection.

Example 2.4.2 We use the example of Semigroup and Monoid given previously, Monoid is a theory extension of SemiGroup. The theory morphism (SemiGroup, Monoid, Φ) in which $\Phi = [U \mapsto U, * \mapsto *]$, is an identity injection. This is graphically depicted in Figure 2.2.

We see that the identity injection shows us how SemiGroup is a subtheory Monoid. □

Example 2.4.3 We use the example of BinaryRelation and OrderRelation above. OrderRelation is a theory renaming of BinaryRelation. The theory morphism (BinaryRelation, OrderRelation, Φ) in which $\Phi = [U \mapsto U, R \mapsto \leq]$ is a renaming injection. This is graphically depicted in Figure 2.3.
Here, BinaryRelation and OrderRelation are isomorphic and Φ defines a renaming to turn the former one into the latter one. □

$$\boxed{\text{BinaryRelation}} \xrightarrow{\Phi = [U \mapsto U, * \mapsto \leq]} \boxed{\text{OrderRelation}}$$

Figure 2.3: Example of a Renaming Injection

$$\begin{tikzcd} T_1 \arrow[r, "\Phi_1"] \arrow[rr, bend right, "\Phi_2 \circ \Phi_1"'] & T_2 \arrow[r, "\Phi_2"] & T_3 \end{tikzcd}$$

Figure 2.4: Composition of Theory Morphisms

Generally speaking, a theory extension T' from T is represented by the identity injection (T, T', id) (id is the identity mapping on the language of T). Furthermore, a theory renaming T' from T is represented by the renaming injection (T, T', Φ) where Φ renames symbols in T to match those in T'.

2.4.2 Operations on Theory Morphisms

In this section, we introduce some useful operations on theory morphisms.

Projection

For convenience, we define three projection functions source, target and mapping that return the source theory, target theory and the mapping of a theory morphism, respectively. That means, given a theory morphism $M = (T, T', \Phi)$, $M.\mathsf{source} = T$, $M.\mathsf{target} = T'$, $M.\mathsf{mapping} = \Phi$.

Composition

Another useful operation is *composition* of two theory morphisms. Intuitively, composition of two theory morphisms mimics the action of moving from one theory to another theory via an intermediate one by following the arrows between them.

Formally, suppose $M_1 = (T_1, T_2, \Phi_1)$ and $M_2 = (T_2, T_3, \Phi_2)$ are two theory morphisms where the target theory of M_1 and the source theory of M_2 are the same, i.e. $M_2.\mathsf{target} = M_1.\mathsf{source} = T_2$. Then $M_3 = M_2 \circ M_1$ is called the *composition* of M_1 and M_2 and $M_3 = (T_1, T_3, \Phi_2 \circ \Phi_1)$. Here, $\Phi_2 \circ \Phi_1$ is the composition of the two mappings Φ_1 and Φ_2. This is graphically shown in Figure 2.4.

$$\begin{tikzcd}
\boxed{\text{BinaryOperation}} \arrow[r, "\Phi_1 = [U \mapsto U, R \mapsto \leq]"] \arrow[rr, bend right, "\Phi_2 \circ \Phi_1 = [U \mapsto U, R \mapsto \leq]"'] & \boxed{\text{OrderRel.}} \arrow[r, "\Phi_2 = [U \mapsto U, \leq \mapsto \leq]"] & \boxed{\text{ReflexiveOrderRel.}}
\end{tikzcd}$$

Figure 2.5: Example of a Composition of Theory Morphisms

Example 2.4.4 Again, we take the example of BinaryRelation and OrderRelation above. OrderRelation is a theory extension of BinaryRelation. We define the theory of ReflexiveOrderRelation by extending OrderRelation with the reflexivity axiom:

```
ReflexiveOrderRelation := OrderRelation extended by {
    axiom reflexive_<= : reflexive((<=))
}
```

Suppose we have two theory morphisms $M_1 = (\text{BinaryOperation}, \text{OrderRelation}, \Phi_1)$ and $M_1 = (\text{OrderRelation}, \text{ReflexiveOrderRelation}, \Phi_1)$ in which Φ_1 is a renaming injection, $\Phi_1 = [\text{U} \mapsto \text{U}, \text{R} \mapsto \leq]$, and $\Phi_2 = [\text{U} \mapsto \text{U}, \leq \mapsto \leq]$ is the identity injection. Then $M_1 \circ M_2$ is the theory morphism (BinaryOperation, ReflexiveOrderRelation, $\Phi_2 \circ \Phi_1$) in which $\Phi_2 \circ \Phi_1 = [\text{U} \mapsto \text{U}, \text{R} \mapsto \leq]$. Figure 2.5 graphically depicts this. □

Pushout

Another essential operation is the *pushout* of two theory morphisms. The combination of theory morphisms is closely related to a theory combination.

Definition 2.4.5 Formally, let $M_1 = (T, T_1, \Phi_1)$ and $M_2 = (T, T_2, \Phi_2)$ be two theory morphisms having a common source theory T. $M_1 \bigoplus M_2$ constructs the whole commutative diagram as graphically depicted in Figure 2.6 and is called the pushout of M_1 and M_2.

In the above commutative diagram, the pushout of M_1 and M_2 is the whole commutative diagram including the theory T_3 and three theory morphisms Φ_{13}, Φ_{23} and Φ_3. These theory morphisms are injections.

For convenience, in the following, we define notations for accessing the components of the commutative diagram of a pushout of theory morphisms.

$$\begin{tikzcd}
T \arrow[r, "\Phi_1"] \arrow[d, "\Phi_2"'] \arrow[dr, "\Phi_3"] & T_1 \arrow[d, "\Phi_{13}"] \\
T_2 \arrow[r, "\Phi_{23}"] & T_3
\end{tikzcd}$$

Figure 2.6: Pushout of Theory Morphisms

- $(M_1 \bigoplus M_2)$.PushoutTheory $= T_3$.
- $(M_1 \bigoplus M_2)$.Right $= (T, T_2, \Phi_2)$.
- $(M_1 \bigoplus M_2)$.Left $= (T, T1, \Phi_1)$.
- $(M_1 \bigoplus M_2)$.Diagonal $= (T, T_3, \Phi_3)$.

Let us take a look at some examples of how theory morphism combination works.

Example 2.4.6 Assume that we have two theory morphisms (Magma, CommutativeMagma, Φ_1) and (Magma, Semigroup, Φ_2) where Magma, Semigroup and CommutativeMagma are defined as below:

```
Magma := Theory {
  U : type;
  * : (U, U) -> U;
}
Semigroup := Magma extended by {
  axiom associativity_*_ : associative((*));
}
CommutativeMagma := Magma extended by {
  axiom commutativity_*_ : commutative((*));
}
```

Moreover, Φ_1 and Φ_2 are identity injections with $\Phi_1 = [\mathrm{U} \mapsto \mathrm{U}, * \mapsto *]$ and $\Phi_2 = [\mathrm{U} \mapsto \mathrm{U}, * \mapsto *]$.

Since both theory morphisms have the same source theory Magma, we can calculate their pushout as shown in Figure 2.7.

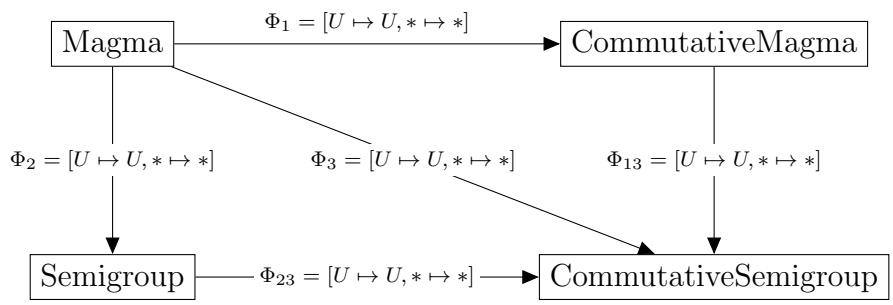

Figure 2.7: Example of a Pushout of Theory Morphisms

In particular, the pushout contains:

- A theory CommutativeSemigroup which is the theory combination of Semigroup and CommutativeMagma over Magma.
- A theory injection (CommutativeMagma, CommutativeSemigroup, Φ_{13}) where Φ_{13} is an identity injection and $\Phi_{13} = [\mathrm{U} \mapsto \mathrm{U}, * \mapsto *]$.
- a theory injection (Semigroup, CommutativeSemigroup, Φ_{23}) where Φ_{23} is the identity injection, $\Phi_{23} = [\mathrm{U} \mapsto \mathrm{U}, * \mapsto *]$.
- A theory injection (Magma, CommutativeSemigroup, Φ_3) where Φ_3 is an identity injection and $\Phi_3 = [\mathrm{U} \mapsto \mathrm{U}, * \mapsto *]$.

The resulting theory CommutativeSemigroup is the theory combination of CommutativeMagma and Semigroup over Magma:

```
CommutativeSemigroup := combine CommutativeMagma, Semigroup
                        over Magma
```

And in the expanded form:

```
CommutativeSemigroup := Theory {
  U : type;
  * : (U, U) -> U;
  axiom associativity_*_ : associative((*));
  axiom commutativity_*_ : commutative((*));
}
```

□

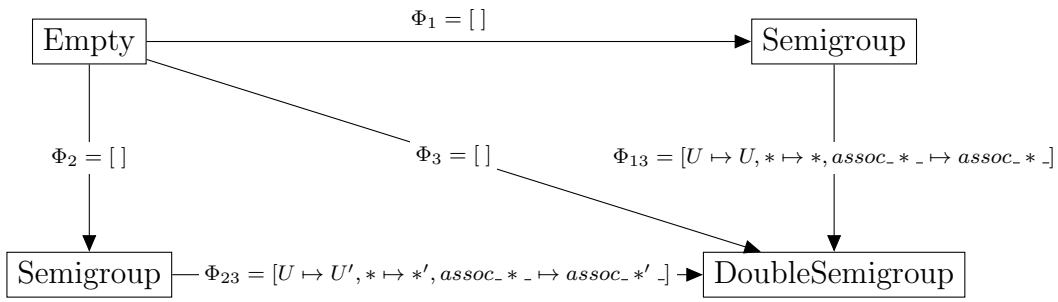

Figure 2.8: Example 2 of a Pushout of Theory Morphisms

Example 2.4.7 Assume that we have a theory morphism $M = (\mathsf{Empty}, \mathsf{Semigroup}, \Phi)$ such that Empty is the empty theory, Semigroup the theory of semigroup as defined previously and Φ the empty injection from the Empty to Semigroup. The pushout of M with itself $M \bigoplus M$ is the commutative diagram in Figure 2.8.

In particular, the pushout contains:

- A theory DoubleSemigroup which is the theory combination of Semigroup with itself over Empty.
- An identity injection $\Phi_{13} = [U \mapsto U, * \mapsto *, associativity\_*\_ \mapsto associativity\_*\_]$.
- An identity injection $\Phi_{23} = [U \mapsto U', * \mapsto *', associativity\_*\_ \mapsto associativity\_*'\_]$.
- An empty injection $\Phi_3 = [\,]$.

DoubleSemigroup is the the result of the following theory combination:

```
DoubleSemigroup := combine Semigroup, Semigroup over Empty
```

In expanded form, it is:

```
DoubleSemigroup := Theory {
  U  : type;
  U' : type;
  *  : (U, U) -> U;
  *' : (U', U') -> U';
  axiom associativity_*_ := associative((*));
```

```
  axiom associativity_*'_ := associative((*'));
}
```

Here, since we are disjointly unioning two identical theories, their symbols are renamed to avoid collision. □

2.5 Approaches to Organizing Mathematical Knowledge

We have seen that theories are building blocks for formalizing mathematical concepts in logic. Furthermore, thanks to biform theories, knowledge in the form of formal deduction and symbolic computation can both be formalized using the same kind of structure. In this section, we take a look at several approaches to organizing mathematical knowledge.

2.5.1 Big Theories Method

In the big theories method [14], a set of powerful axioms are chosen such that any model satisfying these axioms contains all of the objects we are interested in. A portion of mathematical knowledge is then formalized within the big theory. Furthermore, theorems are stated and proven from the chosen axioms within the big theory . The most widely used big theory is Zermelo-Fraenkel (ZF) set theory and its variants. For instance, the prominent Mizar system [23] is built upon the Tarski-Grothendieck set theory which is ZF set theory augmented with Tarski's axioms.

2.5.2 Little Theories Method

In the little theories method [14], a number of theories is used in the process of formalizing a portion of mathematical knowledge. The result of the formalization is a network of theories in which complex theories are constructed from simpler theories using theory creation (see subsection 2.1.2).

Theorems proven in one theory are transfered to other contexts via an explicit construction of a theory interpretation. As a result, in contrast to the big theories method, in the little theories method, both mathematical knowledge and reasoning are distributed over the theory network instead of in a single big theory. According to Dr. Farmer's CAS 760 course notes [15], the biggest advantage of the little theories

method is that a theory can developed be "in the right language at the right level of abstraction".

For instance, suppose we would like to formalize abstract algebra, e.g. semigroups, monoids, groups, rings, vector spaces. Following the little theories method, we would construct a network of theories, e.g. theory of a semigroup, theory of a monoid etc. Over the course of the formalization of a theory, we only focus on the essence of that theory ignoring information irrelevant to the task at hand.

If we are formalizing a theory of a monoid, the language would consist of a carrier set M, a binary operation $*$ and an element e. The set of axioms would contain the monoid axioms, i.e. the associativity axiom of $*$ and the identity axiom of e. In fact, we are free to choose the right language for the theory. In the example of monoid, we may call its binary operation $\circ$ instead of $*$, its identity element a instead of e etc.

Facts about monoids are proven within the local context of the theory and can then be reused in, say a theory of groups, by establishing a theory interpretation from the theory of monoids to the theory of groups.

Due to its advantages, we firmly believe in the little theories method over the big theories method. The paper [14] explains the little theorie method in more detail.

2.5.3 Tiny Theories Method

A special version of the little theories method is called the *tiny theories method*. Essentially, the tiny theories method is the extreme version of the little theories method in the sense that in the tiny theories approach, a theory and its intermediate descendant theory differs only by a conceptual unit (hence the word "tiny"). By a conceptual unit, we mean a concept or an axiom.

For instance, suppose that we have already defined the theory of a semigroup. Now, since a monoid differs from a semigroup by having an identity element e (both left and right identity), we want to define the theory of a monoid from the theory of semigroup. In the little theories method, we are allowed to add three conceptual units (1) an element e, (2) the axiom specifying that e is left identity and (3) an axiom specifying that e is right identity to the theory of a semigroup at once to build the theory of monoid as graphically depicted in Figure 2.9.

However, in the tiny theories method, three conceptual units are not added at the same time. Instead, one conceptual unit is added at a time to construct a descendant theory. Figure 2.10 depicts one (hyper-theoretical) way of constructing the theory of

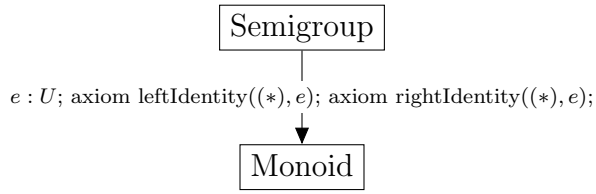

Figure 2.9: Example of the Little Theories Method

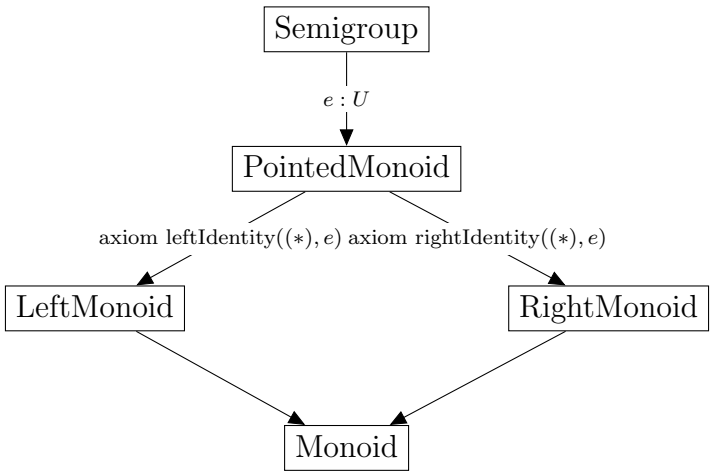

Figure 2.10: Example of the Tiny Theories Method

monoid from the theory of semigroup using the tiny theories method.

First, we add the concept e of type U to Semigroup to construct a new theory called PointedMonoid. Then, we add the axiom of e being the left identity to construct the theory LeftMonoid. Likewise, we add the axiom of e being the right identity to construct the theory RightMonoid. The theory of monoids is the theory combination of LeftMonoid and RightMonoid over PointedMonoid.

It is noteworthy that using the tiny theories approach, it can happen that some constructed theories may not correspond to any concepts used in mathematics. For instance, PointedMonoid is an artifical theory whose name has been invented. Furthermore, there are a lot of ways to construct a theory. Figuring out the most meaningful way of constructing theories and giving them good (possibly artifical) names are two major challenges when using the tiny theory approach.

2.5.4 High-Level Theories Method

As mentioned previously, the little theories method and its special version the tiny theories method allow for formalizing mathematical knowledge in the right language in the right level of abstraction while constructing a library of formalized mathematics. This is especially convenient for the developer of the library. However, end users tends to prefer to have a high-level view of the mathematical knowledge stored in the library. They are mostly not very interested in the implementation detail. This motivates the work on *high-level theories method* [2]. Nevertheless, since in this thesis, our central interest is the little (and tiny) theories method, we do not further discuss the high-level theories method.

CHAPTER 3

THE MATHSCHEME LIBRARY

In Chapter 2, we discussed the techniques that have been developed for representing and organizing mathematical knowledge. Within the scope of the MathScheme Project, a library of formalized mathematics called the *MathScheme Library* is currently being developed based on these techniques. This chapter aims to give an overview of the top-level requirements, the design decisions of the library and the content of the library. [3] gives a nice overview of the techniques behind the MathScheme Library.

3.1 Requirements

The following are the top-level requirements of the MathScheme Library as stated in [10]:

Requirement 1 (Usability) *The MathScheme Library should serve users who want to explore and apply the knowledge it contains.*

One of the disadvantages of contemporary libraries of formalized mathematics is that they only serve a handful of experts in the field and are totally inaccessible to a wide audience. We believe that a library of formalized mathematics is much more valuable if it can be used by a large group of mathematics practitioners ranging from students to engineers and mathematicians.

Requirement 2 (Developability) *The MathScheme Library should serve developers who want to organize and expand the knowledge it contains.*

The development of the library requires the availability of necessary techniques and tools. A developer of the library should be able to use these techniques and tools to organize the library and later on extend the knowledge it contains.

Requirement 3 (Universality) *The MathScheme Library should hold mathematical knowledge of all formalizable forms and kinds.*

Mathematical knowledge comes in different forms and kinds. The library should be capable of holding knowledge of all these forms and kinds so that users and developers do not have to maintain a significant body of knowledge outside of the library.

3.2 Design Decisions

Having discussed the top-level requirements, the following lists the design decisions of the MathScheme Library [10] that have been identified by MathScheme's project leaders Dr. Carette and Dr. Farmer.

Design Decision 1 *The MathScheme Library is a network of biform theories.*

The library should be built as a network of biform theories. Each biform theory captures a mathematical concept that can be in the form of both axiomatization and symbolic computation. The biform theories are interconnected via theory morphisms. As a result of the little theories method, mathematical knowledge and reasoning are distributed over the network. Additionally, a theory and its direct descendant theory differs only by one conceptual unit.

Design Decision 2 *In the library network, complex theories are built from simpler theories by theory extension, theory combination, theory renaming and theory instantiation.*

As we see in Chapter 2, there are different ways of constructing theories. In the process of constructing the library, primitive theories are built from scratch. Moreover, complex theories are built from simpler ones by theory extension, theory combination, theory renaming and theory instantiation. This way, the theories network can be constructed in a stepwise manner where more complex theories are built upon previously constructed simpler theories.

Design Decision 3 *For developers, the MathScheme Library includes a network of tiny theories.*

For developers, who develop and extend the library, the theories network is a network of tiny theories that show all the details of how the library is constructed.

Design Decision 4 *For users, the MathScheme Library will include a collection of high-level theories [2].*

For users, who explore the library, the theories network is a network of high-level theories that provides a high-level view of the library. The high-level view is more convenient for users since it hides all implementation details that are irrelevant to them.

3.3 Current Implementation

At the time of this writing, the MathScheme Library contains formalizations of abstract algebra, which was mainly developed by Dr .Carette and Dr. O'Connor, and data types such as character, string, stack, queue etc. which were mainly developed by Filip Jeremic and Vincent Maccio.

The language for formalization is MSL. There is an OCaml implementation of MSL that can parse a theory file into an internal representation and process it as well as type check it. There is also an experimental implementation of the translation from MSL to Chiron. However, here we do not focus on the implementation but rather on how the mathematical knowledge is formalized in the library.

3.3.1 Abstract Algebra

The portion of abstract algebra in the library is significant in size and is built upon the tiny theories method. To demonstrate how the formalization works, in the following we show how the theory of monoid can be built up starting from the empty theory in a stepwise manner. This is graphically depicted in Figure 3.1. The complete formalization of abstract algebra can be found at [19].

In the figure, the root theory is the Empty theory which contains nothing:

Empty := **Theory** {}

Carrier is Empty extended with with a carrier set U:

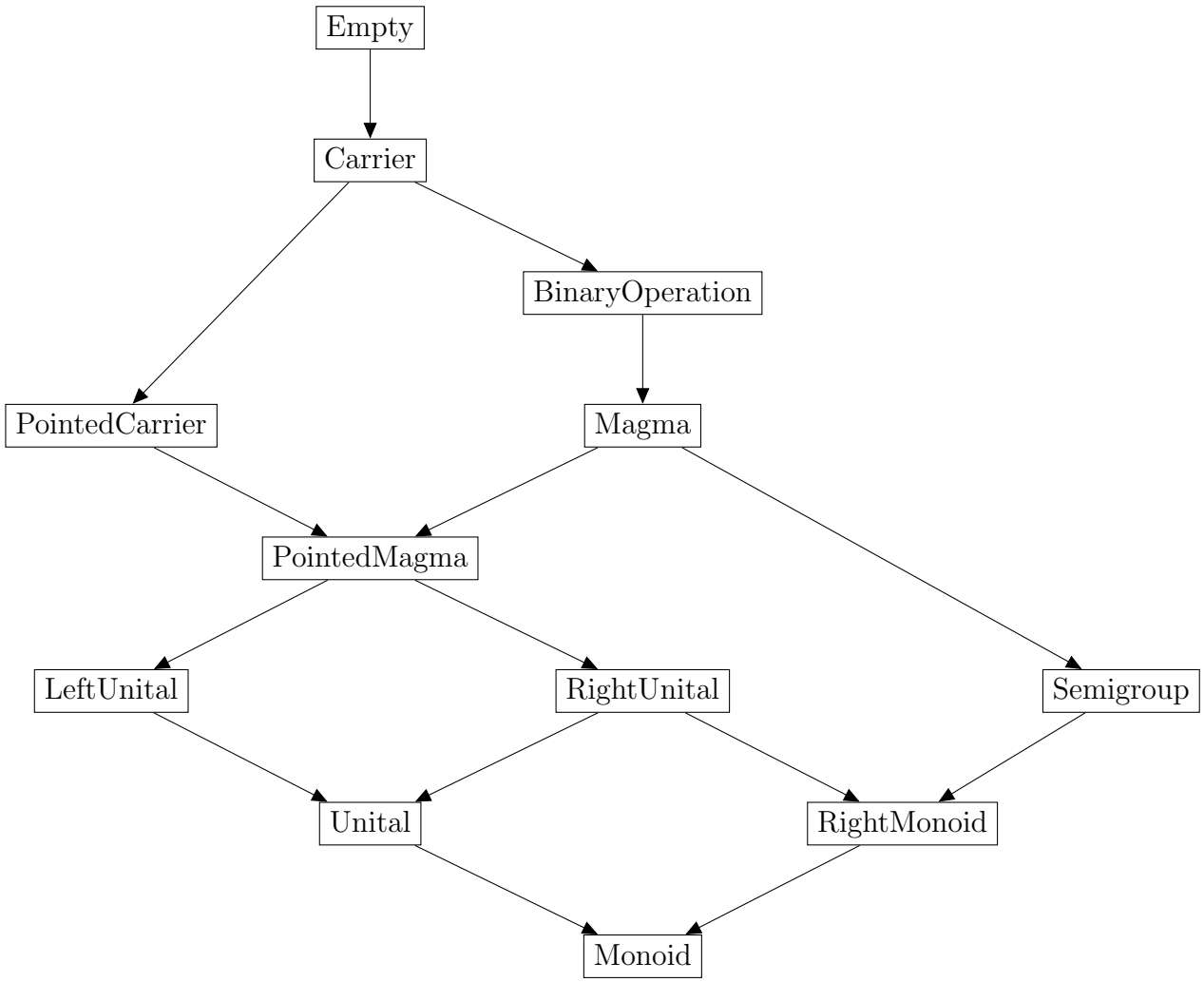

Figure 3.1: The Construction of a Theory of a Monoid

```
Carrier := Empty extended by { U : type; }
```

PointedCarrier is Carrier extended with an element e of the carrier set:

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

Monoid is the combination of Unital and RightMonoid over RightUnital:

```
Monoid := combine Unital, RightMonoid over RightUnital
```

In expanded form, it is:

```
Monoid := Theory {
  U : type;
  * : (U,U) -> U;
  e : U;
  axiom leftIdentity_*_e : leftIdentity((*),e);
  axiom rightIdentity_*_e : rightIdentity((*),e);
  axiom associative_* : associative((*));
}
```

3.3.2 Concrete Theories

As mentioned previously, besides abstract algebra, other concrete theories have also been formalized. For instance, theories of sequences, bit strings, characters and algorithmic complexity, among other things, have been formalized and can be found in [20]. Since the concrete theories are not relevant for the purpose of this thesis, we do not discuss them further.

CHAPTER 4

REIFICATION OF THEORIES AND THEORY INTERPRETATIONS

This chapter aims to explain the motivation and the algebraic construction for reifiying theories as types as well as theory interpretations as elements.

4.1 Motivation

Theories are the building blocks for building the library of formalized mathematics. The true power of theories lies in the fact that since a theory is a specification of a collection of models, facts proven within a theory are applicable to any of its (possibly infinitely many) models.

For instance, suppose we have a theory of a group. Within that theory, we can prove that the identity element is unique. This result is true in any model of the theory, i.e. in any group. Concretely, since $(\mathbb{Z}, +, 0, -)$ is a model of the theory of a group where 0 is the identity element of $+$, we know that 0 is the unique identity element.

Unfortunately, theories have their weakness. A theory allows us to reason about a concept but not about individual elements belonging to that concept. The theory of a group allows us to reason about the concept of a group. It does not, however, allows us to reason about a collection of groups. Specifically, we cannot express such statements as: for all groups G, property X holds in G or there exists a group G in

which property Y holds.

In MSL, theories are represented by theory expressions (see Appendix) and thus cannot be directly used in an expression. In particular, we cannot declare a group element, e.g. $\mathsf{G} : \mathsf{Group}$. Here, Group is a theory expression and hence, it cannot be used as a type.

Furthermore, often we want to reason about a particular group. Suppose $\mathcal{M}$ is a structure having a group structure witnessed by a theory interpretation from *Group* to $\mathcal{M}$.

Recall that a theory interpretation is our tool for transferring theorems from one theory to another theory. However, a theory interpretation is defined by establishing a connection between two theories and is therefore in the meta-language. Hence, theory interpretations are inaccessible to the reasoning process on the expression level. In particular, we cannot express the statement on the expression level like Φ is a theory interpretation from *Group* to $\mathcal{M}$. In other words, reasoning about theory interpretations directly in the object language is not feasible.

Finally, since theory interpretation is the only way to transfer results from one theory to another, the ability to transfer results of *Group* to another concrete group structure $\mathcal{M}$ depends on whether such a theory interpretation can be established or not. As discussed in Chapter 2, theory interpretation is defined between two theories in a very particular way. In particular, $\mathcal{M}$ should reside in a theory and its components should be defined as concepts which should be arranged in a way that a theory interpretation can be defined.

However, it could happen that even though $\mathcal{M}$ does exhibit a group structure, its components may come from different sources, for instance declared as variables, and hence a theory interpretation from *Group* to $\mathcal{M}$ cannot be defined. In this case, one obvious solution is to extend the notion of theory interpretation so that it embodies the above mentioned scenarios. In fact, this technique is implemented in IMPS [11] and is called a *context interpretation*. Nevertheless, we want to keep the definition of theory interpretation simple and solve the problem by reifying theories as types.

4.2 Reification of a Theory as a Type

The previously mentioned weaknesses of theories motivate us to reify theories as types. Given a theory T, we want to reify it as a type whose elements are models of T. Intuitively, a reification of a theory makes it accessible on the expression level.

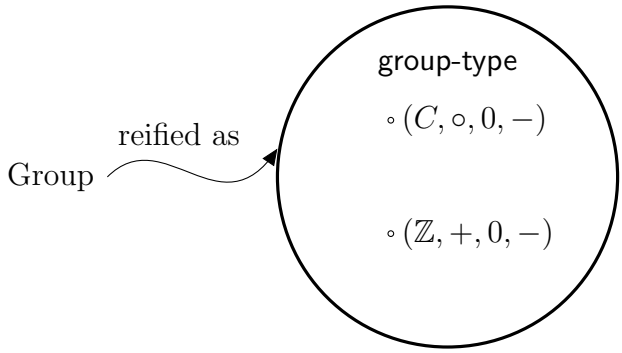

Figure 4.1: Type of Groups

So for instance, reification of the theory of Group above would create a type of groups, called group-type. All mathematical structures exhibiting a group structure would be an element of group-type. In particular, $(\mathbb{Z}, +, 0, -)$, $(C, \circ, 0, -)$ are both elements of group-type. This is graphically depicted in Figure 4.1.

Ultimately, reifying a theory as a type means finding a data type to *represent* a theory. We notice that not all information in a theory is relevant for reification. In fact, only the primitive concepts and axioms of a theory are required constituents of an instance of that concept. Other parts such as theorems etc. are not relevant.

In the example of Group, the concepts define which components a group needs to have, i.e. a carrier set G, a binary operation $*$, an element e, a unary operation $^{-1}$. The axioms dictate what properties these need to have, i.e. $*$ is associative, e is an identity element and $^{-1}$ is an inverse operation.

Since both concepts and axioms constitute the essence of a theory, the data type for representing theories needs to be able to package up both of them.

4.2.1 Reification of a Theory as a Dependent Record Type

A record is the widely used data structure for packaging up various, possibly diverse, data into a single data structure. For instance, various information on employees can be represented by a single record:

```
{ name : string,
  age  : int,
  salary : float
  ...
}
```

At the first attempt, we reify Group as the following record:

```
{ G   : type,
  *   : (G, G) -> G,
  e   : G,
 ^-1  : G -> G
  ...
}
```

However, this is not a valid record since the definition of $*$ depends on the previously defined G. This is what dependent records are good for. A dependent record is similar to a normal record except in a dependent record fields can be dependent on other fields.

De Bruijn was the first one who used a *telescope*, which is linearly dependent record, to package up mathematical structures in his proof development system *Automath* [5].

In a telescope, the type declaration of a field may depend on previously declared fields. As a result, the order of fields matters. So, for instance, (G : type, e : G) is a valid telescope. On the other hand, (e : G, G : **type**) is not a valid telescope. The reason is that, in the later example, the declaration of e requires G to have previously been declared. However, here G is declared after e.

Since a dependent record type has proven to be a suitable data structure for packaging up mathematical structures, we use it as the data structure for representing a reified theory.

The following gives the definition of a dependent record type:

Definition 4.2.1 Formally, a dependent record type is a list of labeled fields $\{l_1 : type_1, \ldots, l_n : type_n\}$ where $n \geq 0$, $type_i$ is type expression and may contain any l_j such that $j < i$.

Using dependent records, the declaration of concepts of a theory can be easily packaged up. For instance, the concepts of `Group` is reified as:

```
{ G   : type,
  *   : (G, G) -> G,
  e   : G,
  ^-1 : G -> G
}
```

We still need to figure out how to deal with axioms. As discussed before, the concepts $G, *, e, ^{-1}$ alone only define the language of a group. The axioms specify the properties they need to have in order for the structure $(G, *, e, ^{-1})$ to be called a group. Since axioms are inseparatable from concepts, we package up the concepts and axioms of a reified theory in the same dependent record. The famous *Curry-Howard correspondence* [24] gives a hint on solving that problem.

In the Curry-Howard correspondence, a formula can be seen as a type whose elements are proofs of that formula. This leads us reify axioms as types of proofs.

For instance, the theory of Group is reified as:

```
group−type =
{ G   : type,
  *   : (G, G) −> G,
  e   : G,
  ^−1 : G −> G,
  associativity_*_   : ProofOf (associative((*)));
  identity_e_  : ProofOf (identity((*),e));
  inverse_−1_  : ProofOf (inverse((*),−1,e));
}
```

Here, ProofOf turns a formula into the corresponding type of proofs of that formula.

In summary, given any theory T, the type of T is represented by a dependent record in which concepts of T are reified to type declarations and axioms are reified to types of proofs.

4.2.2 Elements of a Reified Theory

Having a type representing a theory, we are naturally interested in what its elements are. Obviously, since the type of a theory is a dependent record type, their elements are records. For instance, an element of the type of group group−type is a record representing the integer structure $(\mathbb{Z}, +, 0, -)$:

```
{ G   = ℤ,
  *   = +
  e   = 0,
  ^−1 = −,
```

```
  associativity_*_ = proof_object_associative((+)),
  identity_e_      = proof_object_identity((-),0),
  inverse_-1_      = proof_object_inverse((*),-,0)
}
```

Here, the record contains:

- A type Z.
- A binary operation defined on Z, $+ : (Z, Z) \to Z$.
- An element 0 of Z.
- A unary operation $^{-1}$ defined on Z, $- : Z \to Z$.

Furthermore, the record contains three proof objects being the proofs about the associativity of $+$, identity element of 0 and inverse operation of $-$, respectively.

Generally, an element of the reified type of a theory is a mathematical structure along with the proof objects showing that the components of the structure satisfy the axioms defined in the theory.

4.3 Reification of a Theory Interpretation as an Element

A theory interpretation Φ from a theory T_1 to another theory T_2 is a witness that T_2 has the structure of T_1. We also reify a theory interpretation as a record.

This means that, if a theory T_1 is reified as a type t_1, a theory interpretation from T_1 to another theory T_2 is reified as a member e of t_1. This is graphically depicted in Figure 4.2.

For instance, suppose we have a theory of reals as follows:

```
Real := Theory {
  R : type;
  + : (R,R) -> R;
  0 : R;
  - : R -> R;
  ...
}
```

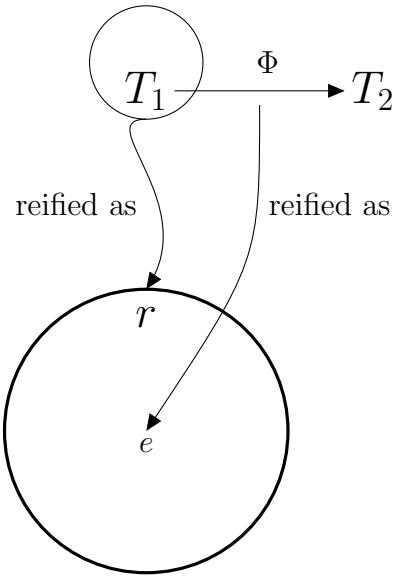

Figure 4.2: Reification of a Theory Interpretation

$\Phi = [G \mapsto \mathbb{R}, * \mapsto +, e \mapsto 0, ^{-1} \mapsto -]$ is a theory interpretation from Group to Real. Φ shows that the model $(\mathbb{R}, +, 0, -)$ of Real has a group structure or equivalently $(\mathbb{R}, +, 0, -)$ is an element of the reified type group−type above. Φ is reified as a record of group−type:

```
{ G   = R,
  *   = +
  e   = 0,
  −1  = −,
  associativity_*_ = proof_object_associative((+)),
  identity_e_      = proof_object_identity((−),0),
  inverse_−1_      = proof_object_inverse((*),−,0)
}
```

4.4 Implementation

We have introduced a type constructor **TypeFrom** to MSL that takes as input a theory and constructs the dependent record type representing that theory. For example, **TypeFrom**(Group) would construct the type group−type. We have generalized MSL's record type mechanism to a dependent record type mechanism.

Reification of theory interpretations has not been implemented yet.

CHAPTER 5

GENERATION OF THEORIES OF HOMOMORPHISMS

Homomorphisms are a very important concept in abstract algebra and model theory. Basically, a homomorphism between two algebraic structures is a structure-preserving mapping between their carrier sets. A homomorphism allows results proved in one mathematical structure to be transfered to another structure, among other things. Consequently, it is very natural to use and reason about homomorphisms within an algebraic setting.

The purpose of this chapter is to introduce two algebraic constructions for automatically deriving the theory of a homomorphism as well as its variants epimorphism, monomorphism and isomorphism from an existing theory. One construction relies on the construction for reifying a theory as a dependent record types via **TypeFrom** (see Chapter 4). The other construction is via calculating the pushout of two theory morphisms (see Chapter 2).

5.1 Motivation

In the current MathScheme Library, a lot of algebraic structures have been formalized as theories. By an algebraic structure, we mean a mathematical structure consisting of one or more carrier sets and operations (functions) defined on them. For example, the library contains the theory of a monoid, the theory of a group and the theory of a

vector space etc. It is expected that more algebraic structures will be formalized and added to the library in the future, both by developers and users.

Eventually, we will need to formalize the notion of a homomorphism for those theories in order to reason about semigroup and group homomorphisms etc. Moreover, in the future whenever a new algebraic structure is formalized as a theory T and added to the library, the corresponding theory of a T-homomorphism needs to be formalized for T in order to reason about T-homomorphisms.

However, instead of repeatedly defining homomorphisms for every single theory, we desire to define an algebraic construction that can automatically derive the notion of a homomorphism based on the structure of the theory. The construction is developed once and for all and can be used to obtain the notion of a homomorphism from an arbitrary theory. This is possible because there is a generic definition of homomorphims.

5.2 Generic Definition of Homomorphism

Even though textbook presentations of homomorphisms vary depending on the concrete structures involved, we can have a generic definition of homomorphisms between two 1-sorted algebraic structures having the same signature as follows:

Definition 5.2.1 Let $\mathcal{M}$ and $\mathcal{N}$ be two 1-sorted algebraic structures having the same signature $\mathcal{S}$. Let M and N be their carrier sets, respectively. $h : M \to N$ is a mapping (function) from the carrier set of $\mathcal{M}$ to the carrier set of $\mathcal{N}$. Furthermore, the following conditions are satisfied;

- For each n-ary function μ ($n \geq 0$) in $\mathcal{S}$, $h(\mu_{\mathcal{M}}(x_1, \ldots, x_n)) = \mu_{\mathcal{N}}(h(x_1), \ldots, h(x_n))$ for all $x_1, \ldots, x_n \in M$.

Then h is called a *homomorphism* between $\mathcal{M}$ and $\mathcal{N}$.

The following gives the definitions of the special cases of a homomorphisms

Definition 5.2.2 A surjective homomorphism is called an *epimorphism*.

Definition 5.2.3 An injective homomorphism is an *monomorphism*.

Definition 5.2.4 A bijective homomorphism is called an *isomorphism*.

Example 5.2.5 Let $(M, *, e, inv)$ and $(M', *', e', inv')$ be two group structures. Let $h : M \to M'$ be a mapping between M and M' satisfying the following conditions:

- $h(x * y) = h(x) *' h(y)$ for all x, y in M.
- $h(inv(x)) = inv'(h(x))$ for all x in M.
- $h(e) = e'$.

□

We can extend the definitions of a homomorphism and its variant to multi-sorted algebraic structures having the same signature as follows:

Definition 5.2.6 Let $\mathcal{M}$ and $\mathcal{N}$ be two n-sorted algebraic structures having the same signature $\mathcal{S}$ $(n \geq 1)$. Let $C_1, \ldots, C_{n-1}, M$ and $C_1, \ldots, C_{n-1}, N$ be their carrier sets, respectively. That means, $\mathcal{M}$ and $\mathcal{N}$ have the same carrier sets $C_1, \ldots, C_{n-1}$ but may differ in one carrier set. $h : M \to N$ is a function between M and N such that:

- For each m-ary function $\mu : (S_1, \ldots, S_m) \to M$ $(m \geq 1)$ in $\mathcal{S}$ whose return type is M, $S_i \in \{C_1, \ldots, C_{n-1}, M\}$, $h(\mu_{\mathcal{M}}(x_1, \ldots, x_m)) = \mu_{\mathcal{N}}(h_1(x_1), \ldots, h_n(x_m)))$ for all $x_1 \in S_1, \ldots, x_m \in S_m$. Here, if $S_i = M$ then $h_i = h$, otherwise if $S_i \neq M$, $h_i = id$ (id is the identity function).
- For each constant c in $\mathcal{S}$, $h(c_{\mathcal{M}}) = c_{\mathcal{N}}$.

For example, in the following we define the notion of a homomorphism between two vector spaces. A vector space is a 2-sorted algebraic structure that contains both fields and vectors.

Since the definition of a vector space is based on the definition of a field, we first review the definition of a field:

Definition 5.2.7 A *field* is a mathematical structure $\mathcal{F} = (F, +, *, 0, 1)$ such that:

- $0 \neq 1$.
- $(F, +, 0)$ is a commutative group.
- $(F - \{0\}, *, 1)$ is a commutative group.

- $a * (b + c) = a * b + a * c$ for all a, b, c in F.

The following is the definition of a vector space:

Definition 5.2.8 A *vector space* is a mathematical structure $\mathcal{V} = (V, +, *, 0)$ over a field $\mathcal{F}$ such that:

- $* : (F, V) \to V$.
- $(V, +, 0)$ is a commutative group.
- $a * (v + w) = a * v + a * w$ for all a in $\mathcal{F}$ and v, w in V.
- $(a + b) * v = a * v + b * v$ for all a, b in $\mathcal{F}$ and v in V.
- $a * (b * v) = (a * b) * v$.

Example 5.2.9 Let $\mathcal{U} = (V_U, +_U, *_U, 0_U)$ and $\mathcal{W} = (V_W, +_W, *_W, 0_W)$ be two vector spaces over the same field $\mathcal{K}$. Let $h : V_U \to V_W$ such that:

- $h(x +_U y) = h(x) +_W h(y)$ for all x, y in V_U.
- $h(a *_U x) = a *_W h(x)$ for all a in F and x in V_U.
- $h(0_U) = 0_W$.

Then h is a homomorphism between $\mathcal{U}$ and $\mathcal{W}$. □

It is noteworthy that in textbooks, a homomorphism between two vector spaces over the same field is usually called a *linear map*. Furthermore, textbook presentations of linear map usually only mention the former two axioms. The last axiom is usually omitted since it can be proved from the former two axioms.

Nonetheless, the definition of a homomorphism of vector spaces derived from the generic definition is equivalent to textbook definitions.

5.3 Constructing a Homomorphism via TypeFrom

The first construction for generating the notion of a homomorphism relies on the reification of a theory as a dependent record type via **TypeFrom** (Chapter 4).

First, we consider 1-sorted theories since this is the simplest case. Given a 1-sorted theory T, using **TypeFrom** we can reify T as a type which allows us to declare T

elements of that type. The notion of a T-homomorphism can be derived by declaring (1) two elements A, B of **TypeFrom**(T), (2) a function h between the carrier sets of A and B and (3) axioms specifying that h preserve the functions and constants in T (as given in Definition 5.2.6).

Example 5.3.1 Suppose we would like to derive the theory of a group homomorphism from the following theory of Group:

```
Group := Theory {
   G   : type;
   *   : (G,G) -> G;
   e   : G;
   inv : G -> G;
   axiom associativity_*_ : associative((*));
   axiom identity_e_      : identity((*),e);
   axiom inverse_inv_     : inverse((*),inv, e);
}
```

The theory of a group homomorphism can be obtained by declaring (1) two group elements A and B of **TypeFrom**(Group), (2) a function $h : A.G \to B.G$ between two carrier sets of A and B and (3) axioms specifying that h preserves the functions and constants $*, inv, e$ of Group. Concretely, the theory of GroupHomomorphism, derived from Group, may look as follows:

```
GroupHomomorphism := Theory {
  type GroupType = TypeFrom(SemiGroup);
  A, B : GroupType;
  h : A.G -> B.G
  axiom : forall x, y : A.G . f(x A.* y) = f(x) B.* f(y);
  axiom : forall x : A.G . f(A.inv(x)) = B.inv(f(x));
  axiom : h(A.e) = B.e;
}
```

□

The theory of a group epimorphism and theory of a group monomorphism are theory extensions (see Chapter 2) of GroupHomomorphism by adding the surjectivity and injectivity axiom, respectively:

```
GroupEpimorphism := GroupHomomorphism extended by {
 axiom : surjective(h);
}
GroupMonomorphism := GroupHomomorphism extended by {
 axiom : injective(h);
}
```

Finally, the theory of a group isomorphism is the theory combination (see Chapter 2) of GroupEpimorphism and GroupMonomorphism over GroupHomomorphism:

```
GroupIsomorphism :=
  combine GroupEpimorphism, GroupMonomorphism
  over GroupHomomorphism
```

Deriving the theory of a homomorphism from a multi-sorted theory is more complicated since it is not obvious on which carrier sets the mapping should be defined. One solution is that we restrict ourselves to defining the notion of a homomorphism for only one particular carrier set while the remaining carrier sets are fixed using Definition 5.2.6.

Example 5.3.2 We would like to derive the theory of a vector space homomorphism from the following 2-sorted theory of VectorSpace. Furthermore, the homomorphism shall be defined for the vector domain V.

```
Field := Theory {
 Concepts
 F : type;
 + : (F,F) -> F;
 * : (F,F) -> F;
 - : F -> F;
 / : F -> F;
 0,1   : F;

 Axioms
  axiom : 0 \= 1;
  axiom associativity_+_ : associative(+);
  axiom identity_+_ : leftIdentity((+), 0)
        and rightIdentity((+), 0);
```

```
  axiom inverse_-_  : inverse((+),(-),0);
  axiom commutativity_+_ : commutative(+);
  axiom associativity_*_ : associative(*);
   axiom identity_1_ : leftIdentity((*),1)
         and rightIdentity((*),1);
  axiom inverse_/_ : forall x : F. (x \= 0)
                     implies (x * (/x) = 1);
  axiom commutativity_*_ : commutative(*);
  axiom distributivity_*_over_+ : distributive((*),(+));
}


VectorSpace := Field extended by {
  V : type;
  +_V : (V, V) -> V;
  *_V : (F, V) -> V;
  0_V : V;
  -_V : V -> V;

  axiom associativity_+_ :
     forall u, v, w : V . u +_V (v +_V w) = (u +_V v) +_V w;
  axiom commutativity_+_V_ :
     forall v, w : V . v +_V w = w +_V v;
  axiom identity_0_V_:
     forall v : V . v +_V 0_V = 0_v +_V v = v;
  axiom inverse     :
     forall v : v +_V (-v) = 0_V;
  axiom distributivity1 :
     forall a : F .
     forall v, w : V . a *_V (v +_V w) = a *_V v +_V a *_V w;
  axiom distributivity2 :
     forall a, b : F . forall v : V .
                       (a+_V b)*_V v = a*_V v +_V b*_V v;
  axiom compatibility    :
     forall a, b : F . forall v : V . a*_V(b*_V v) = (a*_V b)*_V v;
```

```
  axiom identity_1_V_    :
     forall v : V . 1*_V v = v;
}
```

The theory of a vector space homomorphism can be obtained by declaring (1) two vector space elements of **TypeFrom**(VectorSpace), (2) a function $h : A.V \to B.V$ and (3) axioms specifying that the field in A is identical to the field domain in B, i.e. A.F = B.F and that f preserves the functions whose return type is V and constants of type V. Concretely, the theory of VectorSpaceHomomorphism, generated from VectorSpace, may look as follows:

```
VectorSpaceHomomorphism := Theory {
  type VectorSpaceType = TypeFrom(VectorSpace);
  A, B : VectorSpaceType;
  h : A.V -> B.V;
  axiom : A.F = B.F;
  axiom : forall x, y : A.V . h(x A.+ y) = h(x) B.+ h(y);
  axiom : forall a : A.F . x : A.V . h(a A.* x) = a B.* h(x);
  axiom : forall x : A.V . h(A.- x) = B.- h(x);
  axiom : f(A.0_V) = B.0_V;
}
```

□

Based on the two examples above, the construction can be generalized to generate the theory of T-homomorphism from an arbitrary n-sorted theory ($n \geq 1$) with n carrier sets $\{C_1, \ldots, C_{n-1}, V\}$. Assume that the homomorphism shall be defined for the carrier set V of T, the algorithm for generating THomomorphism is as follows:

(1) Initially, THomomorphism is an empty theory.

(2) Add a type declaration **type** TType = **TypeFrom**(T) to THomomorphism.

(3) Add two elements A, B of type TType, i.e. A, B : TType.

(4) Add a mapping $h : A.V \to B.V$ to THomomorphism.

(5) For each carrier set C_i, add an axiom specifying that $A.C_i$ is identical to $B.C_i$, that is $A.C_i = B.C_i$.

(6) For each m-ary function (or constant when $n = 0$) $f : S_1, \ldots, S_m \to V$ ($m \geq 0$, $S_i \in \{C_1, \ldots, C_{n-1}, V\}$) in the signature whose return type is V, add an axiom specifying that h preserves f, i.e. $\forall x_1 : A \,.\, S_1, \ldots, x_m : A.S_m.h(A.f(x_1, \ldots, x_m)) = B.f(h(x_1), \ldots, h(x_m))$.

(7) The resulting THomomorphism is the theory of T-homomorphism of T.

Furthermore, a T-epimorphism (-monomorphism and -isomorphism) can be obtained from THomomorphism as below:

```
TEpimorphism := GroupHomomorphism extended by {
 axiom : surjective(h);
}
GroupMonomorphism := GroupHomomorphism extended by {
 axiom : injective(h);
}
GroupIsomorphism :=
  combine GroupEpimorphism, GroupMonomorphism
  over GroupHomomorphism
```

5.4 Constructing a Homomorphism via Pushout

The other construction for generating the notion of a homomorphism is to calculate the pushout of theory morphisms (Chapter 2).

Again, for the sake of simplicity, we first consider 1-sorted theories. Given a 1-sorted theory T, we would like to construct the theory of a T-homomorphism from it.

The key idea is that from T a new theory called $DoubleT$ is generated that contains two copies of T (both concepts and axioms) in it. The homomorphism is defined as a function from the carrier set of the first T copy to the carrier set of the second T copy along with the structure-preserving axioms.

Example 5.4.1 We would like to derive the theory of a group homomorphism from Group. The theory of a group homomorphism can be generated from Group in two steps:

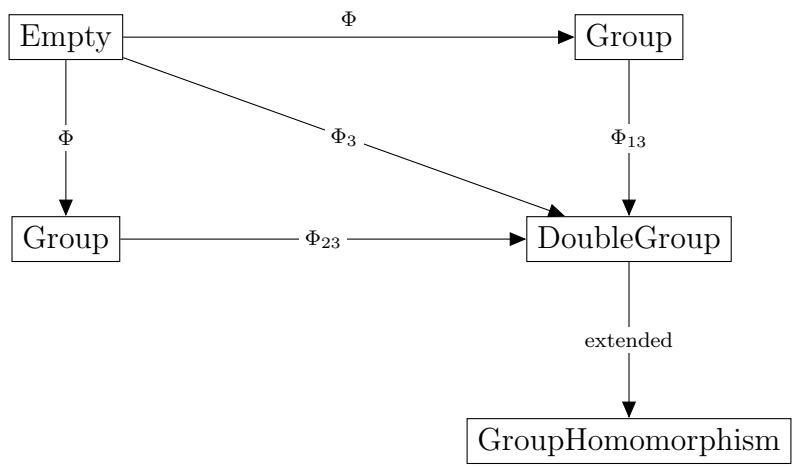

Figure 5.1: Constructing a Group Homomorphism via Pushout

(1) Construct the theory combination DoubleGroup of Group with itself by calculating the pushout of two theory morphisms (Empty, Group, Φ) with itself where Φ is an empty injection. Assume that $(G_1, *1, \mathrm{inv}1, e1)$ and $(G_2, *2, \mathrm{inv}2, e2)$ are the first and second copies of group's concepts in DoubleGroup.

(2) Extend DoubleGroup to THomomorphism by adding (1) a function $h : G_1 \to G_2$ and (2) axioms specifying that h preserve $*, inv, e$ in Group.

This is graphically depicted in Figure 5.1.

GroupHomomorphism may look as follows:

```
GroupHomomorphism := Theory {
  G1 : type;
  G2 : type;
  *1 : (G1, G1) -> G1;
  *2 : (G2, G2) -> G2;
  e1 : G1;
  e2 : G2;
  inv1 : G1 -> G1;
  inv2 : G2 -> G2;
  axiom associativity_*1_ : associative((*1));
  axiom associativity_*2_ : associative((*2));
  axiom identity_e1_      : identity ((*1), e1);
```

```
axiom identity_e2_      : identity ((*2), e2);
axiom inverse_inv1_     : inverse ((*1), inv1, e1);
axiom inverse_inv2_     : inverse ((*2), inv2, e2);

h : G1 -> G2;
axiom : forall x, y : G1 . h(x *1 y) = h(x) *2 h(y);
axiom : forall x : G1 . h(inv1(x)) = inv2(h(x));
axiom : h(e1) = e2;
```

□

Before constructing the theory of a group epimorphism, a group monomorphism and a group isomorphism, we define two theory morphisms (MultiCarrierWithFunc, GroupHomomorphism, Φ_1) and (MultiCarrierWithFunc, MultiCarrierWithSurjectiveFunc, Φ_2) where MultiCarrier, MultiCarrierWithFunc and MultiCarrierWithSurjectiveFunc are theories of two carrier sets, two carrier sets with a function between them and two carrier sets with a surjective function between them, respectively:

```
MultiCarrier := Theory {
  G1, G2 : type;
}
MultiCarrierWithFunc := MultiCarrier extended by {
  h : G1 -> G2;
}
MultiCarrierWithSurjectiveFunc := MultiCarrierWithFunc
  extended by {
    axiom : surjective(h);
}
```

The theory of group epimorphism can be obtained by calculating the pushout of the two theory morphisms mentioned above as graphically depicted in Figure 5.2.

Similarly, the theory of a group monomorphism can be obtained by calculating the pushout of two theory morphisms (MultiCarrierWithFunc, GroupHomomorphism, Φ_1) and (MultiCarrierWithFunc, MultiCarrierWithInjectFunc, Φ_2) where MultiCarrierInjectiveFunc is the theory of two carrier sets with an injective function between them:

```
MultiCarrierWithInjectiveFunc := MultiCarrierWithFunc
```

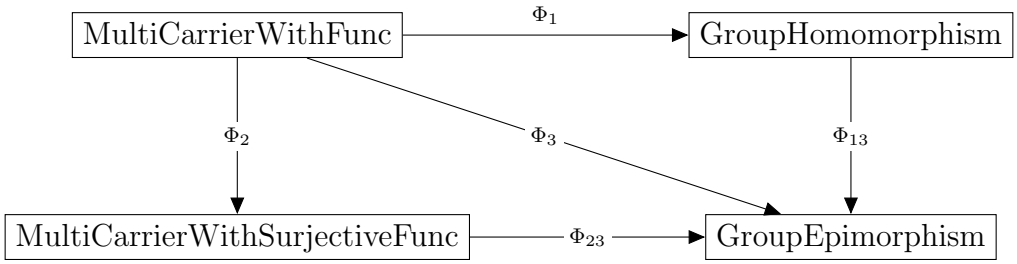

Figure 5.2: Constructing a Group Epimorphism via Pushout

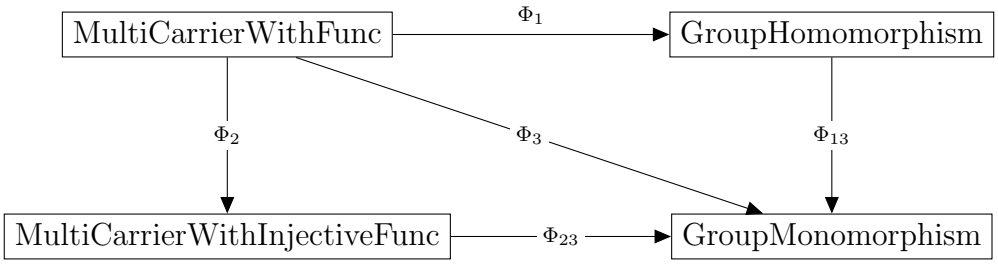

Figure 5.3: Constructing a Group Monomorphism via Pushout

```
  extended by {
    axiom : injective(h);
}
```

In expanded form, it is:

```
MultiCarrierWithInjectiveFunc := Theory {
  G1, G2 : type;
  h : G1 -> G2;
  axiom : injective(h);
}
```

Finally, the theory of a group isomorphism can be obtained by calculating the pushout of (GroupHomomorphism, GroupEpimorphism, Φ_1) and (GroupHomomorphism, GroupMonomorphism, Φ_2) where Φ_1 and Φ_2 are identity injections (Figure 5.4).

Deriving the theory of a homomorphism from a multi-sorted theory is more complicated but can be done. The key idea is that the notion of a homomorphism is defined for only one carrier set while the remaining carrier sets are fixed. The fixed carrier sets along with their axioms have one copy in the generated theory of homo-

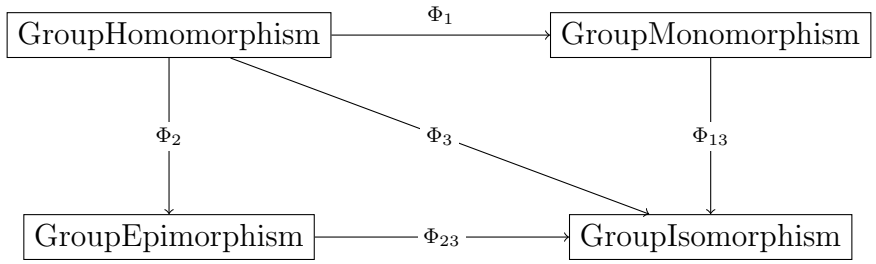

Figure 5.4: Constructing a Group Isomorphism via Pushout

morphism. On the other hand, carrier set, for which the homomorphism is defined, its operations and and its axioms have two copies so that we can define the homomorphism mapping between them.

Example 5.4.2 The theory of a vector space homomorphism can be derived from VectorSpace in two steps:

(1) Construct the theory combination DoubleVectorSpace of VectorSpace with itself by calculating the pushout of the theory morphisms (Field, VectorSpace, Φ) with itself where Φ is an identity injection and Field the theory of a field given previously. Assume that $(V, +, *, 0, 1)$ and $(V', +', *', 0', 1')$ are the first and second copies of vector space's concepts in DoubleVectorSpace.

(2) Extend DoubleVectorSpace to VectorSpaceHomomorphism by adding (1) a function $h : V \to V'$ and (2) axioms specifying that h preserve the vector operations whose return type is V and constants of type V

This is graphically depicted in Figure 5.5.

The theory of homomorphism of vector spaces may look as below in MSL:

```
VectorSpaceHomomorphism := Theory {
 Concepts
  F    : type;
  +    : (F,F) -> F;
  *    : (F,F) -> F;
  -    : F -> F;
  /    : F -> F;
  0,1  : F;
```

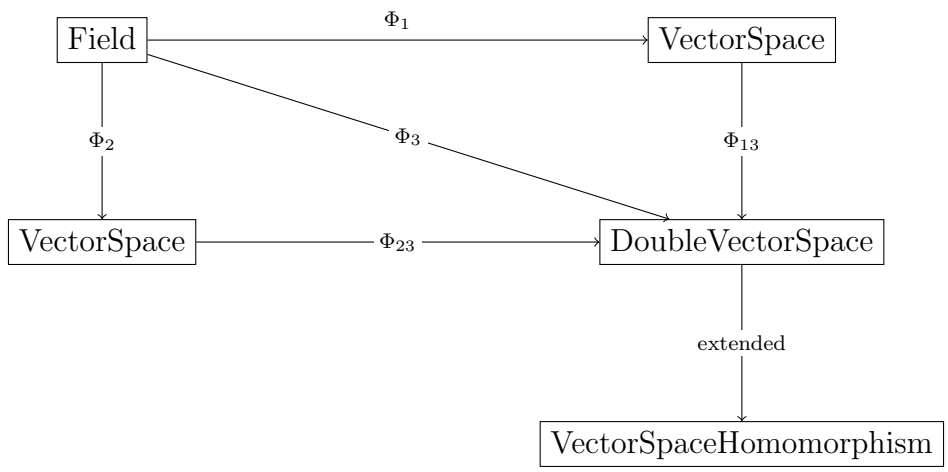

Figure 5.5: Constructing a VectorSpace Homomorphism via Pushout

```
Facts
 axiom : 0 \= 1;
 axiom associativity_+_ : associative(+);
 axiom identity_+_ : leftIdentity((+), 0)
                     and rightIdentity((+), 0);
 axiom inverse_-_  : inverse((+),(-),0);
 axiom commutativity_+_ : commutative(+);
 axiom associativity_*_ : associative(*);
 axiom identity_1_ : leftIdentity((*),1)
                     and rightIdentity((*),1);
 axiom inverse_/_ : forall x : F. (x \= 0)
                    implies (x * (/x) = 1);
 axiom commutativity_*_ : commutative(*);
 axiom distributivity_*_over_+ : distributive((*),(+));

 V : type;
 +_V : (V, V) -> V;
 *_V : (F, V) -> V;
 0_V : V;
 -_V : V -> V;
```

```
V' : type;
+V' : (V, V) -> V;
*V' : (F, V) -> V;
0V' : V;
-V' : V -> V;

axiom associativity_+V_ :
   forall x, y, z : V . x +V (y +V z) = (x +V y) +V z;
axiom associativity_+V'_ :
   forall x, y, z : V . x +V' (y +V' z) = (x +V' y) +V' z;
axiom commutativity_+V_ :
   forall x, y : V . x +V y = y +V x;
axiom commutativity_+V'_ :
   forall x, y : V . x +V' y = y +V' x;
axiom identity_0V_ :
   forall x : V . x +V 0V = 0V +V x = x;
axiom identity_0V'_:
   forall x : V . x +V' 0V' = 0V' +V' x = x;
axiom inverse_-V_ :
   forall x : V . x +V (-V x) = 0V;
axiom inverse_-V'_ :
   forall x : V' . x +V' (-V' x) = 0V';
axiom distributivity1  : forall a : F .
   forall x, y : V . a *V (x +V y) = a *V x +V a *V y;
axiom distributivity1' : forall a : F .
   forall x, y : V . a *V' (x +V' y) = a *V' x +V' a *V' y;
axiom distributivity2  :
   forall a, b : F . forall x : V .
                    (a +V b)*Vx = a*Vx +V b*Vx;
axiom distributivity2' :
   forall a, b : F .
   forall x : V' . (a +V' b) *V' x = a *V' x +V' b *V' x;
axiom compatibility    : forall a, b : F .
   forall x : V  . a *V (b *V x) = (a *V b) *V x;
```

```
  axiom compatibility'  : forall a, b : F .
     forall x : V' . a *V' (b *V' x) = (a *V' b) *V' x;

  h: V -> V';
  axiom : forall x, y : V . h(x +V y) = h(x) +V' h(y);
  axiom : forall a : F . forall x : V . h(a*Vx) = a*V;h(x);
  axiom : forall x : V . h(−Vx) = −V'h(x);
  axiom : h(0) = 0';
}
```

□

Similar to the group construction, the theory of a VectorSpaceEpimorphism and VectorSpaceMonomorphism can be obtained by calculating the pushout of the morphisms, (MultiCarrierWithFunc, VectorSpaceHomomorphism, Φ_1) with (MultiCarrierWithFunc, MultiCarrierWithSurjectiveFunc) and (MultiCarrierWithFunc, VectorSpaceHomomorphism, Φ_1) with (MultiCarrierWithFunc, MultiCarrierWithInjectiveFunc), respectively.

Finally, the theory of a vector space isomorphism can be obtained by calculating the pushout of (VectorSpaceHomomorphism, VectorSpaceEpimorphism, Φ_1) with (VectorSpaceHomomorphism, VectorSpaceMonomorphism, Φ_2).

The construction can be generalized to generate the theory of a T-homomorphism (epimorphism, monomorphism and isomorphism) from an arbitrary n-sorted theory T ($n \geq 1$). Let $\{C_1, \ldots, C_{n-1}, M\}$ be T's carrier sets. Moreover, the homomorphism shall be defined for M. The construction algorithm is the following:

(1) Construct the theory combination DoubleT of T with itself over the theory FixedTheory by calculating the pushout of two theory morphisms (FixedTheory, T, Φ_1) and (FixedTheory, T, Φ_2) where Φ_1 and Φ_2 are theory injections.

(2) Extend DoubleT to THomomorphism with (1) a function $h : M \to M'$ and (2) axioms specifying that h preserve the operations whose return type is M.

This is graphically depicted in Figure 5.6. Here, FixedTheory contains the part that is fixed. If T is a 1-sorted theory (as in the example of group), FixedTheory is the empty theory. Otherwise, if T is multi-sorted theory (at least 2-sorted), FixedTheory contains the carrier sets, their operations and the specifying axioms that are common

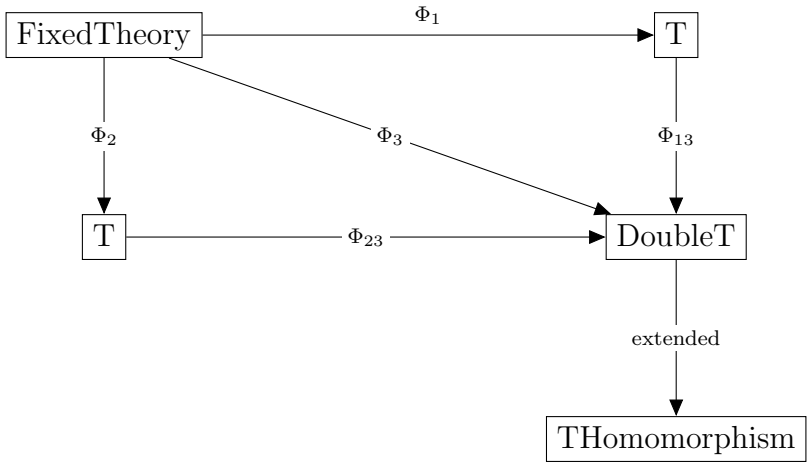

Figure 5.6: Constructing a T-Homomorphism via Pushout

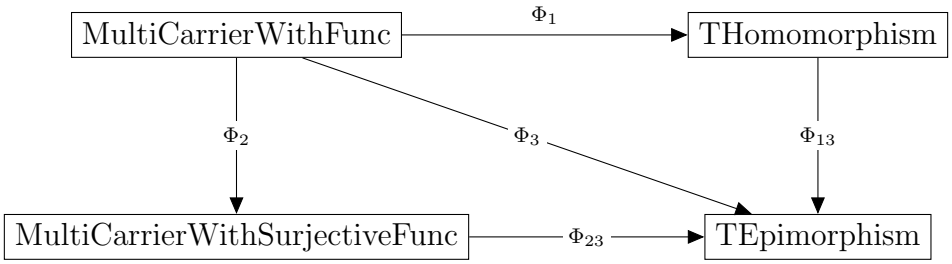

Figure 5.7: Constructing a T-Epimorphism via Pushout

part of the two theories between which the homomorphism is being defined. As seen previously, in the example of the vector space, FixedTheory is the theory of Field.

The theory of T-epimorphism can be obtained from THomomorphism by calculating the pushout of (MultiCarrierWithFunc, THomomorphism, Φ_1) and (MultiCarrierWithFunc, MultiCarrierWithSurjectiveFunc, Φ_2) as graphically illustrated in Figure 5.7.

Similarly, the theory of T-monomorphism can be obtained by calculating the pushout of two theory morphisms (MultiCarrierWithFunc, THomomorphism, Φ_1) and (MultiCarrierWithFunc, MultiCarrierWithInjectFunc, Φ_2). Figure 5.8 graphically shows this.

Finally, the theory of isomorphism of T can be obtained by calculating the pushout of (THomomorphism, TEpimorphism, Φ_1) and (TMonomorphism, TIsomorphism, Φ_2).

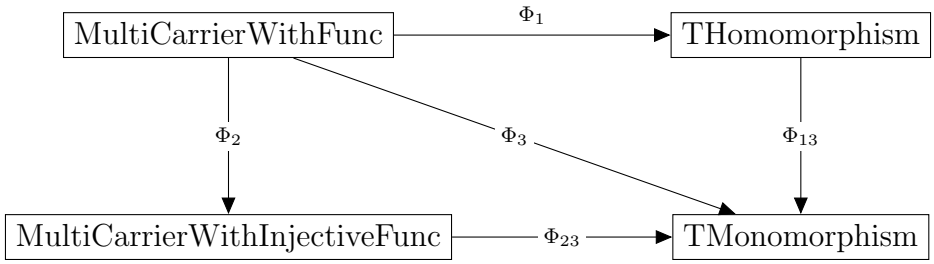

Figure 5.8: Constructing a T-Monomorphism via Pushout

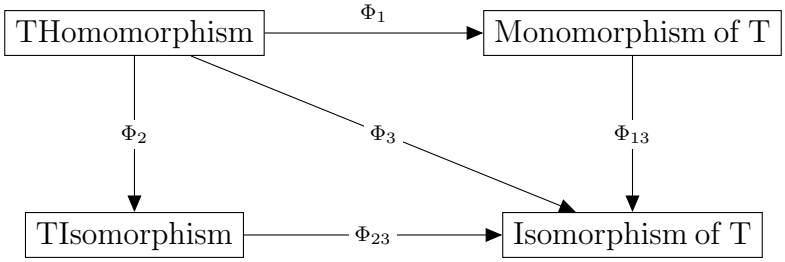

Figure 5.9: Constructing a T-Isomorphism via Pushout

5.5 Comparison of the Two Constructions

Both constructions discussed previously generate the theory of a T-homomorphism (as well as epimorphisms, monomorphisms and isomorphisms) from an arbitrary n-sorted theory. At first glance, the construction using **TypeFrom** produces more compact theories than the construction using pushout. Nevertheless, this is because most of the complexity is handled by **TypeFrom** which, when expanded, is a dependent record containing all the concepts and proof objects of the input theory.

The downside of the construction using pushout is that the generated theory could be enormous since there are two copies of the part for which the homomorphism is defined. However, the advantage is that if the theory morphisms are fully supported the calculation of the theory morphism (FixedTheory, T, Φ) with itself produces not only the theory DoubleT but other theory morphisms as well. These can be stored in the system and reused in other contexts.

CHAPTER 6

GENERATION OF THEORIES OF SUBSTRUCTURES AND SUBMODELS

Similar to homomorphism, substructure and submodel are also important concepts in abstract algebra and model theory. In an algebraic setting, it is often the case that attention is paid to a particularly interesting subset of elements of a carrier set in a certain structure. Moreover, things become more interesting and straighforward if the functions of the structure also work on the subset.

The purpose of this chapter is to discuss the algebraic constructions that have been developed for generating the theory of a substructure and the theory of a submodel from an existing theory. Similar to generating a homomorphism (see Chapter 5), two constructions, one using **TypeFrom** and one using pushout, can be used to generate the notions of a substructure and a submodel.

6.1 Motivation for the Generation of a Substructure and a Submodel

Similar to the motivation for generating a homomorphism, instead of manually defining the notion of a sub-T-structure and of a sub-T-model for a theory T, we desire to automatically derive it instead. Since our aim is a method for an automatic derivation, we are interested in the generic definition of substructures and submodels.

6.2 Generic Definition of a Substructure and a Submodel

The following gives the definition for the notion of a substructure in terms of two 1-sorted algebraic structures:

Definition 6.2.1 Let $\mathcal{M}$ and $\mathcal{N}$ be two 1-sorted algebraic structures having the same signature. Furthermore, let M and N be their carrier sets, respectively. $\mathcal{M}$ is said to be a *substructure* of $\mathcal{N}$, if

- M is nonempty and M is a subset of N.
- For every n-ary function f in the signature ($n \geq 1$), $f_{\mathcal{M}} = f_{\mathcal{N}} \mid M^n$. That is, $f_{\mathcal{M}}$ is the restriction of $f_{\mathcal{N}}$ on M.

Or equivalently:

Definition 6.2.2 Let $\mathcal{M}$ and $\mathcal{N}$ be two 1-sorted algebraic structures having the same signature. Furthermore, let M and N be their carrier sets, respectively. $\mathcal{M}$ is said to be a *substructure* of $\mathcal{N}$, if

- M is a subset of N.
- For every n-ary function (or constant when $n = 0$) f in the signature ($n \geq 0$), $f_{\mathcal{M}} = f_{\mathcal{N}} \mid M^n$. That is, $f_{\mathcal{M}}$ is the restriction of $f_{\mathcal{N}}$ on M.

Based on that, the following is the definition of a submodel:

Definition 6.2.3 Let T be a 1-sorted theory capturing one or a collection of algebraic structures. $\mathcal{M}$ and $\mathcal{N}$ are two 1-sorted algebraic structures having the same signature. $\mathcal{M}$ is said to be a *submodel* of $\mathcal{N}$, if

- $\mathcal{M}$ is a substructure of $\mathcal{N}$.
- $\mathcal{M}$ and $\mathcal{N}$ are both models of T.

Example 6.2.4 Let T be the theory of a group (its definition in MSL is given in Section 5.3 in Chapter 5). Let $\mathcal{N} = (\mathbb{N}, +_N, 0_N)$ and $\mathcal{Z} = (\mathbb{Z}, +_Z, 0_Z)$ be a structure of natural numbers and a structure of integers, respectively. Clearly, $\mathcal{N}$ is a substructure of $\mathcal{Z}$ because:

- $\mathbb{N}$ is nonempty and $\mathbb{N} \subseteq \mathbb{Z}$.
- $+_N = +_Z \mid \mathbb{N}$.

However, $\mathcal{N}$ does not form a submodel of $\mathcal{Z}$ because, unlike $\mathcal{Z}$, $\mathcal{N}$ is not a group. $\square$

We extend the definitions of substructures and submodels above to multi-sorted algebraic structures. In this case, the notion of a substructure is defined for a particular carrier set while the remaining carrier sets are fixed. That means, the remaining carrier sets of the two structures are identical.

Definition 6.2.5 Let $\mathcal{M}$ and $\mathcal{N}$ be two n-sorted algebraic structures having the same signature ($n \geq 1$). Let $C_1, \ldots, C_{n-1}, M$ and $C_1, \ldots, C_{n-1}, N$ be their carrier sets, respectively. Moreover, the following conditions are satisfied:

- $M \subseteq N$.
- For each m-ary function (or constant when $n = 0$) $f : (S_1, \ldots, S_m) \to M$ in the signature whose return type is M where $m \geq 1$ and $S_i \in \{C_1, \ldots, C_{n-1}, M\}$, $f_{\mathcal{M}} = f_{\mathcal{N}} \mid M^m$ for all $x_1 \in S_1, \ldots, x_m \in S_m$. In other words, $\forall x_1 \in S_1, \ldots, x_m \in S_m \,.\, f_{\mathcal{M}}(x_1, \ldots, x_m) = f_{\mathcal{N}}(x_1, \ldots, x_m)$.

Then $\mathcal{M}$ is a *substructure* of $\mathcal{N}$.
Based on that, the following is the definition of a submodel:

Definition 6.2.6 Let T be an n-sorted theory. $\mathcal{M}$ and $\mathcal{N}$ are two n-sorted algebraic structures having the same signature. Let $C_1, \ldots, C_{n-1}, M$ and $C_1, \ldots, C_{n-1}, N$ be their carrier sets, respectively. $\mathcal{M}$ is said to be a *submodel* of $\mathcal{N}$, if

- $\mathcal{M}$ is a substructure of $\mathcal{N}$.
- $\mathcal{M}$ and $\mathcal{M}$ are both models of T.

Example 6.2.7 Let $\mathcal{W} = (V_W, +_W, *_W, 0_W)$ be a vector space over field K (the definition of a vector space is given in Chapter 5). Let $\mathcal{U} = (V_U, +_U, *_U, 0_U)$ be a structure of the same signature of $\mathcal{W}$ over the same field K. Furthermore, the following conditions are satisfied:

- V_U is nonempty and $V_U \subseteq V_W$.
- $+_U = +_W \mid V_U^2$.

- $*_U = *_W \mid V_U^2$.

Then $\mathcal{U}$ is a substructure of $\mathcal{W}$. If $\mathcal{U}$ is also a vector space itself, $\mathcal{U}$ is a submodel of $\mathcal{W}$. $\square$

It is worth mentioning that textbooks normally define only the notion of a submodel of a vector space and refer to it as a *vector subspace* or *linear subspace*. Moreover, its definition is different than the definition based on the generic definition given here.

A textbook's definition would say that $\mathcal{U}$ is a vector subspace of $\mathcal{W}$ if $\mathcal{W}$ is a vector space itself or, equivalently, if the following conditions are satisfied:

- $0_U \in V_W$.
- $\forall a, b : K \,.\, \forall x, y : V_W \,.\, a * u + b * v \in V_W$.
- $\forall a : K \,.\, \forall x : V_W \,.\, a * x \in V_W$.

Nevertheless, our definition for vector subspace derived from the generic definition of submodels is equivalent to these textbook's definitions.

6.3 Constructing a Theory of a Substructure and a Submodel via TypeFrom

Similar to generating a homomorphism (see Chapter 5), the first construction for generating the notion of a substructure relies on the reification of types as dependent record types, via **TypeFrom**, as introduced in Chapter 4.

First, we consider 1-sorted theories since this is the simplest case. Given a 1-sorted theory T, using **TypeFrom** T can be reified as a type which allows for declaring a T element of that type. Based on the structure of T, the notion of a sub-T-structure can be derived by declaring a new carrier set and axioms specifying that (1) the new carrier set is a nonempty subset of the carrier set of T and (2) the functions of T are closed on this new carrier set.

Having the notion of a sub-T-structure, the notion of a sub-T-model can be obtained by extending it with an axiom specifying that the substructure is a model of T.

Example 6.3.1 We would like to derive the theory of a group substructure (subgroup) from the theory of a Group.

The theory of a group substructure can be obtained by declaring a group element A being an element of **TypeFrom**(Group), a new carrier set V and axioms specifying that (1) V is a subset of the carrier set of A, (2) the group's operations are closed on V and (3) the group's constant is in V. Concretely, the theory of GroupSubstructure, generated from Group, may look as follows:

```
GroupSubstructure := Theory {
  type GroupType = TypeFrom(Group);
  A : GroupType;
  V : type;
  axiom : V <: A.U;
  axiom : defined-in(A.e, V);
  axiom : forall x, y : V . defined-in(x A.* y, V);
  axiom : forall x : V . defined-in(A.inv(x), V);
}
```

Here, we use **TypeFrom** to construct the type of groups so that we can declare a group element A : GroupType. The axioms specify that the new type V is a nonempty subset of the carrier of A, e is in A and the functions in A are closed on V.

The notion of a group submodel can be obtained from GroupSubstructure by adding an axiom saying that V along with the group operations and constant in A form a group:

```
GroupSubmodel := GroupSubstructure extended by {
  axiom : exists p1, p2, p3 : Proof .
          {G = V,
           * = A.*,
           e = A.e,
           inv = A.inv,
           associativity_*_ = p1,
           identity_e_      = p2,
           inverse_inv_     = p3} in GroupType;
}
```

Here, the axiom specifies that the record consisting of the new carrier set V and group operations and constant of A is an element of **TypeFrom**(Group) or, equivalently, form

a group. □

Unfortunately, there is problem here. Since, it is a well-known fact that any group substructure is also a group submodel, the last axiom in GroupSubmodel turns out to be redundant. This shows one of the weaknesses of the approach.

Deriving the theory of a substructure and the theory of a submodel from a multi-sorted theory is more complicated since it is not obvious on which carrier sets the notion of substructures and submodels should be defined. One solution is that we restrict ourselves to defining the notion of a substructure and submodel for only one particular carrier set while the remaining carrier sets are fixed using Definition 6.4 and Definition 6.4.

The key idea is that in this scenario, the notion of a substructure is defined for only one particular carrier set while the remaining carrier sets are fixed.

Example 6.3.2 We would like to derive the theory of a vector space substructure from the 2-sorted theory of VectorSpace as introduced in Chapter 5. Furthermore, the substructure shall be defined for the vector domain V.

The generation works almost the same as with the group discussed previously. The theory of a vector space substructure can be obtained by declaring a vector space A of **TypeFrom**(VectorSpace), a new carrier set W and axioms specifying that (1) W is a nonempty subset of the vector domain V of A and (2) the vector functions of A are closed on W. Concretely, the theory of VectorspaceSubstructure, generated from VectorSpace, may look as follows:

```
VectorSpaceSubstructure := Theory {
  type VectorSpaceType = TypeFrom(VectorSpace);
  A : VectorSpaceType;
  W : type;
  axiom : (W <: A.V);
  axiom : defined-in (A.0, W);
  axiom : forall x, y : W . defined-in(x A.+ y, W);
  axiom : forall a : A.F, x : W . defined-in(a A.* x, W);
  axiom : forall x : W . defined-in(A.- x, W);
}
```

In VectorspaceSubstructure, the most interesting part is the axiom specifying that A.∗ is closed W:

```
axiom : forall a : A.F, x : W . defined-in(a A.* x, W);
```

Recall that in the original theory VectorSpace, $*$ has following declaration:

```
* : (F, V) -> V;
```

which is defined on two carrier sets F and V. Since the substructure is being defined for the vector domain V and the field domain is fixed, the scalar a is quantified over A.F.

Similar to the group example above, the notion of a vector space submodel can be obtained from VectorSpaceSubstructure by adding an axiom saying that V together with the vector's operations and constants in A form a vector space:

```
VectorSpaceSubmodel := VectorSpaceSubstructure extended by {
  axiom : exists p1, p2, p3, p4, p5, p6, p7, p8 : Proof .
          {V = W,
           + = A.+,
           * = A.*,
           0 = A.0,
           associativity_+_ = p1,
           commutativity_+_ = p2,
           identity_0_      = p3 ,
           inverse          = p4,
           distributivity1  = p5,
           distributivity2  = p6,
           compatibility    = p7,
           identity_1_      = p8 } in VectorSpaceType;
}
```

□

Based on the two examples above, the construction can be generalized to generate the theory of a sub-T-structure TSubStructure from any input n-sorted theory T ($n \geq 1$). Assume that the substructure shall be defined over the carrier set V of T, the algorithm for generating TSubstructure is as follows:

(1) Initially, TSubstructure is an empty theory.

(2) Add a type declaration **type** TType = **TypeFrom**(T) to TSubstructure.

(3) Add a type declaration W : **type** to TSubstructure.

(4) Add the axiom specifying that V is not empty and V is a subset of A.V, i.e. (W <: A.V).

(5) For each n-ary function (or constant when $n = 0$) $f : (C_1, \ldots, C_n) \to V$ $(n \geq 0)$ of T whose return type is V, generate an axiom specifying that $A.f$ is closed on W, i.e. $\forall x_1 : S_1, \ldots, x_n : S_n$.defined-in$(f(x_1, \ldots, x_n), W)$ where $S_i = W$ if $C_i = V$ and $S_i = A.C_i$ otherwise.

The theory of sub-T-model TSubmodel can be obtained by adding to TSubstructure an axiom: $\exists p_1, \ldots, p_n$: Proof . $\{V = W, f_1 = A.f_1, \ldots, f_m = A.f_m, ax_1 = p_1 \ldots, ax_n = p_n\}$ in TypeFrom(T) where f_i are functions or constants in T and ax_i are the names of the axioms in T.

6.4 Constructing a Theory of a Substructure Via Pushout

Also similar to generating a homomorphism in Chapter 5, the second construction for generating the notion of a substructure is via generating the pushout of theory morphisms (as discussed in Chapter 2).

Again, we first discuss the generation method for 1-sorted theories since this is the simplest scenario. Given a 1-sorted theory T, we would like to generate the theory of a sub-T-structure from it.

The key idea is that we define a new theory called DoubleT that contains two copies of the concepts of T and one copy of the axioms of T in it. This can be done by calculating the pushout of two theory morphisms (Empty, T, Φ) and (Empty, TSignature, Φ) where TSignature contains only the concepts of T, that is T without its axioms.

Then we extend DoubleT to TSubstructure by adding axioms specifying that (1) the first carrier set of the first copy of T is a nonempty subset of the second carrier set of the second copy of T and (2) the functions of the first copy of T are restrictions of the corresponding functions of the second copy of T to the first carrier set of T.

Example 6.4.1 We would like to derive the theory of a group substructure from the theory of a Group. The theory of a group substructure can be generated from Group in two steps:

(1) Construct the theory combination DoubleGroup by calculating the pushout of two theory morphisms (Empty, GroupSignature, Φ_1) and (Empty, Group, Φ_2) where Φ_1 and Φ_2 are empty injections and GroupSignature is Group without its axioms as shown below. This is graphically depicted in Figure 6.1. DoubleGroup contains two copies of Group's concepts and one copy of Group's axioms. Assume that $(G_1, *1, \mathrm{inv}1, e1)$ and $(G_2, *2, \mathrm{inv}2, e2)$ are the first and second copies of group's concepts in DoubleGroup.

(2) Extend DoubleGroup to GroupSubstructure by adding axioms specifying that (1) G_1 is a subset of G_2 and the functions $*1$, $inv1$ of the first group copy are restrictions of the functions $*1$, $inv2$ of the second group copy, respectively.

The following is GroupSignature:

```
GroupSignature := Theory {
   G   : type;
   *   : (G,G) -> G;
   e   : G;
   inv : G -> G;
}
```

The theory of a group substructure may look as below:

```
GroupSubstructure := Theory {
  G1 : type;
  G2 : type;
  *1 : (G1, G1) -> G1;
  *2 : (G2, G2) -> G2;
  e1 : G1;
  e2 : G2;
  inv1 : G1 -> G1;
  inv2 : G2 -> G2;

  axiom : associativity_*2_ : associative((*2));
  axiom : identity_e2_      : identity ((*2), e2);
  axiom : inverse_inv2_     : inverse ((*2), inv2, e2);

  axiom : (G1 <: G2);
```

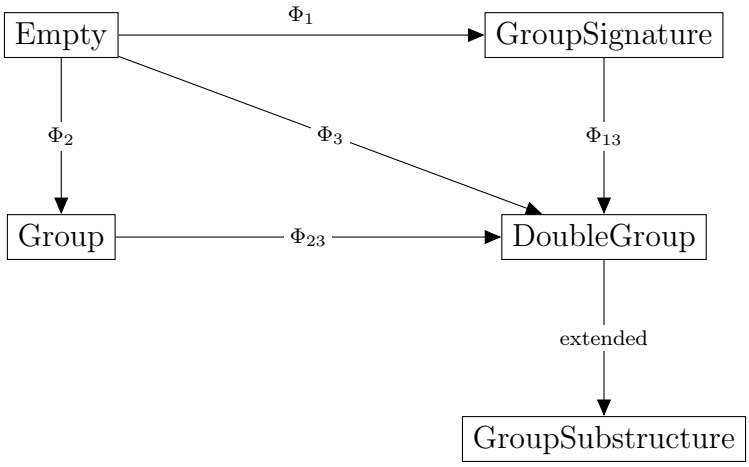

Figure 6.1: Constructing a Group Substructure via Pushout

```
axiom : e1 = e2;
axiom : forall x, y : G1 . x *1 y = x *2 y;
axiom : forall x : G1 . inv1(x) = inv2(x);
```

Notice that GroupSubstructure contains only the concepts but not the axioms of the first copy of Group.

The theory of a group submodel can be obtained from GroupSubstructure by adding an axiom specifying that the first copy of Group's concepts forms a group:

```
GroupSubmodel := GroupSubstructure extended by {
  axiom : exists p1, p2, p3 : Proof .
            {G   = G1,
             *   = *1,
             e   = e1,
             inv = inv1,
             associativity_*_ = p1,
             identity_e_      = p2,
             inverse_inv_     = p3} in GroupType;
            }
```

□

Similar to the first construction using **TypeFrom**, deriving the theory of a substructure from a multi-sorted theory is more complicated but can also be done using

Definition and Definition . The key idea is that the notion of a substructure is defined for only one carrier set while the remaining carrier sets are fixed. The fixed carrier sets along with their axioms have one copy in the generated theory of substructure. On the other hand, the carrier set, for which the substructure is defined, and its functions have two copies. Moreover, there is one copy of the axioms defining the carrier set. This way, it is possible to define the substructure relationship.

Example 6.4.2 We would like to generate the theory of vector space substructure from the theory of the 2-sorted theory of a VectorSpace. Furthermore, the substructure shall be defined for the vector domain V.

Similar to the theory of group, the construction of the theory of a vector space substructure can be done in two steps:

(1) Construct the theory combination DoubleVectorSpace by calculating the pushout of the two theory morphisms (Field, VectorSpaceSignature, Φ_1)) with (Field, VectorSpace, Φ_1)) where Φ_1 and Φ_2 are identity injections and VectorSpaceSignature is VectorSpace without its vector axioms. Assume that $(V, +, *, 0, 1, -)$ and $(V', +', *', 0', 1', -')$ are the first and second copies of vector space's concepts in DoubleVectorSpace.

(2) Extend DoubleVectorSpace to VectorSpaceSubstructure by adding axioms specifying that (1) V is a nonempty subset of V' and (2) the functions of the first vector space copy $+, *, -$ are restrictions of the functions of the second vector space copy $+', *', -'$, respectively.

This is graphically depicted in Figure 6.2.

VectorSpaceSubstructure may look as below:

```
VectorSpaceSubstructure := Theory {

  F    : type;
  +    : (F,F) -> F;
  *    : (F,F) -> F;
  -    : F -> F;
  /    : F -> F;
  0,1  : F;

  axiom associativity_+_ : associative(+);
```

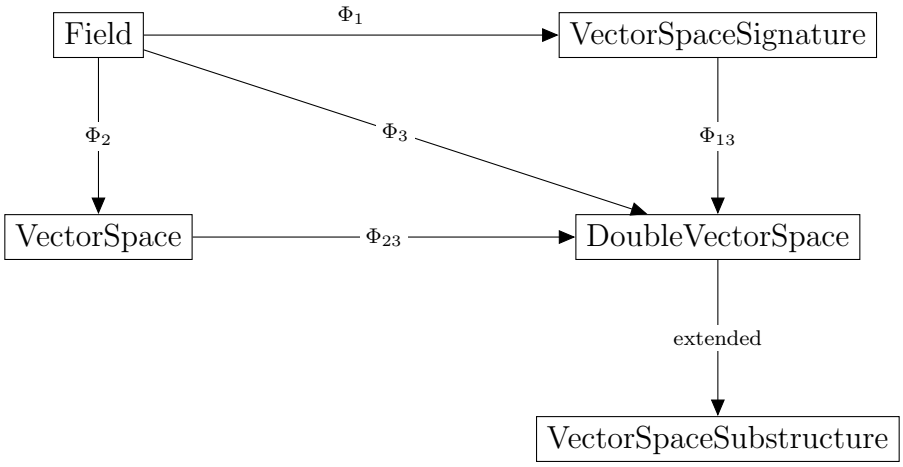

Figure 6.2: Constructing a Vector Space Substructure via Pushout

```
axiom identity_+_ : leftIdentity((+), 0)
                    and rightIdentity((+), 0);
axiom inverse_−_ : inverse((+),(−),0);
axiom commutativity_+_ : commutative(+);
axiom associativity_*_ : associative(*);
axiom identity_1_ : leftIdentity((*),1)
                    and rightIdentity((*),1);
axiom inverse_/_ : inverse((*),(/),1);
axiom commutativity_*_ : commutative(*);
axiom distributivity_*_over_+ : distributive((*),(+));

V : type;
+_V : (V, V) -> V;
*_V : (F, V) -> V;
0_V : V;
−_V : V -> V;

V' : type;
+_V' : (V, V) -> V;
*_V' : (F, V) -> V;
0_V' : V;
```

```
  −V′ : V -> V;

  axiom associativity_+V′_ :
        forall x, y, z : V .
        x +V′ (y +V′ z) = (x +V′ y) +V′ z;
  axiom commutativity_+V′_ :
        forall x, y : V . x +V′ y = y +V′ x;
  axiom identity_0V′_ :
        forall x : V . x +V′ 0V′ = 0V′ +V′ x = x;
  axiom inverse_−V′_ :
        forall x : V' . x +V′ (−V′ x) = 0V′;
  axiom distributivity1  : forall a : F .
        forall x, y : V .
        a *V′ (x +V′ y) = a *V′ x +V′ a *V′ y;
  axiom distributivity2  :
        forall a, b : F .
        forall x : V . (a +V′ b)*V′x = a*V′x +V′ b*V′x;
    axiom compatibility   : forall a, b : F .
        forall x : V  . a *V′ (b *V′ x) = (a *V′ b) *V′ x;

  axiom : (V <: V');
  axiom : 0 = 0';
  axiom : forall x, y : V . x +V y = x +V′ y;
  axiom : forall a : F . forall x : V . a *V x = a *V′ x;
  axiom : forall x : V . −V x = −V′ x;
}
```

□

Based on the two examples above, we can generalize the construction to generate the theory of sub-T-structure from any input n-sorted theory T ($n \geq 1$). Let $C_1, \ldots, C_{n-1}, M$ be its carrier sets. Assume that the substructure shall be defined for M. The algorithm for generating the theory of sub-T-structure is as follows:

(1) Construct the theory combination DoubleT by calculating the pushout of the two theory morphisms (F, TSignature, Φ_1)) with (F, T, Φ_1)) (Figure 6.3). Here, Φ_1 and Φ_2 are identity injections and F contains all the carrier sets $C_1, \ldots, C_{n-1}$,

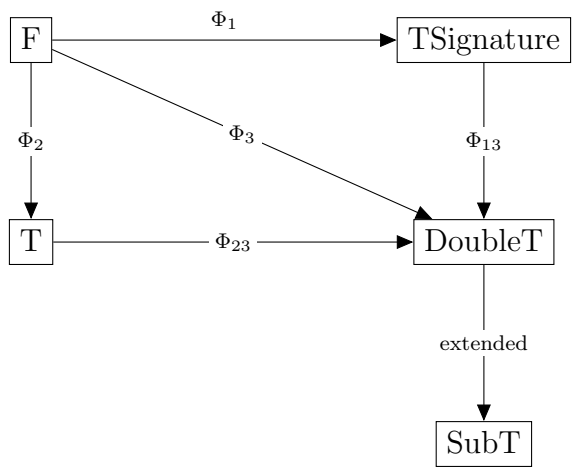

Figure 6.3: Constructing a Sub-T-Structure via Pushout

their operations and defining axioms. Additionally, TSignature is T without its axioms. Assume that M_1 and M_2 are the first and second copies of M in DoubleT.

(2) Extend DoubleT to TSubstructure by adding axioms specifying that (1) M_1 is a nonempty subset of M_2, (2) the functions of the first T copy are restrictions of the functions of the second T copy, respectively and (3) the constants of the first T copy are equal to the constants of the second T copy.

(3) The resulting TSubstructure is the theory of sub-T-structure of T.

Moreover, the theory of a T-submodel can be obtained by extending TSubstructure with an axiom specifying that there exists proof objects showing that TSubstructure is a model of T, i.e. an element of **TypeFrom**(T).

6.5 Comparison of the Two Constructions

The comparison is essentially the same as with constructions of homomorphisms (Section 5.5 of Chapter 5).

Both constructions discussed previously generate the theory of sub-T-structure from an arbitrary n-sorted theory. At first glance, the approach using **TypeFrom** (Section 6.3) produces more compact theories than the approach using pushout (Section

6.4). Nevertheless, this is because most of the complexity is handled by **TypeFrom** which, when expanded, is a dependent record containing all the concepts and proof objects of the input theory.

The downside of the construction using pushout is that the generated theory could be enormous since there are two copies of the part for which the substructure is defined. However, the advantage is that if the theory morphisms are fully supported the calculation of the theory morphism (F, T, Φ) with itself produce not only the theory DoubleT but also theory morphisms Φ_{13}, Φ_{23} and Φ_3 which can be stored and reused in other contexts.

CHAPTER 7

THEORY SYNTAX REPRESENTATION AND OTHER SYNTACTIC MACHINERY

As we have discussed previously, the notion of a biform theory is used to merge axiomatic and algorithmic theories. The facts of a biform theory may specify transformers that are functions manipulating the syntax of expressions. As a result, as opposed to axiomatic theories, reasoning in a biform theory also embodies reasoning about syntactic expressions. Reasoning about syntax requires that the syntactic machinery must be available.

The purpose of this chapter is to introduce several algebraic constructions we have developed to generate syntactic machinery from existing theories. In particular, we introduce a construction for reifying the term algebra of a theory as an inductive data type. We also discuss other potentially useful syntactic functions such as the length function of syntactic expressions.

The starting point of the work presented in this chapter is Dr. O'Connor's idea of automatically deriving a term algebra from the structure of a 1-sorted theory. The idea has been implemented in the current MathScheme implementation.

7.1 Motivation

When defining a biform theory, we usually start with the axiomatic part because the axiomatic part captures the essence of the mathematical concept we are trying to formalize. Then, we may want to reason about the syntax within the theory and thus define the syntactic machinery for it. Lots of syntactic machinery turns out to be automatically generable from the axiomatic part.

To illustrate this, let us take a look at an example inspired by the example of the theory of Bit initially introduced by Dr. Carette. Suppose we are defining the theory of booleans. First, we define an inductive data type B with two elements true and false. Then we define boolean functions such as conjunction, disjunction and negation etc. :

```
Bool := Theory {
  Inductive B
    | true  : B
    | false : B
  ;
  and : (B, B) -> B;
  or  : (B, B) -> B;
  not : B -> B;
  ...
}
```

Here we do not list the defining axioms for **and**, **or** and **not** etc. since they are not relevant to our discussion.

The theory Bool above is purely axiomatic because it does not contain any transformers. Now, we could add transformers that manipulate the syntax of boolean expressions. Examples of syntactic boolean expressions are |^true^|, |^ false ^|, |^**not**(**and**(true,false)))^| etc. (|^^| is an MSL's ASCII notation for quotation).

In particular, we could add a transformer called simplify that simplifies boolean expressions while preserving their semantics:

```
Bool := Theory {
  ...
  simplify : BoolExpr -> BoolExpr;
}
```

Associated with simplify is a program that implements the simplification algorithm. For instance, simplify reduces **not**(**and**(true,false))) to true.

Here, BoolExpr is the type of all syntactic boolean expressions that can be constructed. That means, |^ true ^|, |^ false ^|, |^**not**(**and**(true,false)))^| etc. are elements of BoolExpr.

So we need BoolExpr, the type of syntactic boolean expressions, in order to be able to declare the simplify function. We would like to generate BoolExpr based on the definition of Bool.

Furthermore, within the theory, we may also want to define statements about simplify and reason about it. In particular, we want to express that simplify preserves the semantics of input argument and the simplified expression is shorter than the input expression.

This could be illustrated by the following extended version of Bool:

```
Bool := Theory {
  Inductive B
    | true  : B
    | false : B
  ;
  and : (B, B) -> B;
  or  : (B, B) -> B;
  not : B -> B;
  simplify : BoolExpr -> BoolExpr;
  length : BoolExpr -> Nat;

  axiom : forall e : BoolExpr .
          [| e |]_B = [| simplify(e) |]_B;
  theorem : forall e : BoolExpr .
          length(simplify(e)) <= length(e);
}
```

Here, we assume that there is a theory of naturals Nat with a total order $\leq$ defined on it.

The specification of simplify requires the existence of a length function that calculates the length of a given syntactic expression. Ideally, the user does not have to define length but it is already predefined. length is another example of a useful

syntactic function that can be potentially predefined as well as generated.

In short, for specifying and reasoning about transformers, we have to manually define the type of syntactic expressions and other syntactic machinery. We aim to reduce this burden by predefining and generating as much syntactic machinery as we can from the information extracted from a theory.

7.2 Definition of the Term Algebra of a Theory

In the later sections, we will use the concept of the term algebra of a theory and therefore define it in this section. First, we review the well-known notion of the term algebra of a language. Then, we define the notion of the term algebra of a theory from it.

Definition 7.2.1 (Term Algebra of a Language) Given a language L with a set of constants and function symbols and a set of variables V, the term algebra (also called Herbrand universe) of L over V is the set Term of all terms that can be constructed from L. Term can be defined by inductive definition as below:

(1) For each constant symbol $c \in L, c \in \mathsf{Term}$.

(2) For each function symbol $f \in L$ of arity n, $f(t_1, \ldots, t_n) \in \mathsf{Term}$ where t_i are terms.

(3) Term contains only elements defined in (1) and (2).

Based on that, the following is the definition of the term algebra of a theory:

Definition 7.2.2 (Term Algebra of a Theory) Given a theory $T = (L, \Gamma)$ in which L contains only constant symbols and function symbols, the term algebra of T is the term algebra of L.

Example 7.2.3 Suppose we have a theory T whose languages consist of a constant symbol c and function symbol f:

```
T := Theory {
  U : type;
  c : U;
  f : U -> U;
}
```

The term algebra of T consists of $c, f(c), f(f(c)), f(f(f(c)))$, etc. $\square$

We notice that the term algebra of a theory only depends on its language but not on its axioms.

7.3 Syntax Framework

Discussing a system that directly supports syntactic reasoning tends to be very confusing. The reason is that it is often very hard to recognize what belongs to semantics and syntax. This motivated Dr. Farmer and Pouya Larjani to formulate the idea of a syntax framework [12]. In this framework, semantic and syntactic constituents are clearly distinguished. The goal of the framework is to provide a formal setting to discuss systems with syntax reasoning support. In this section, we give a brief explanation of the notion of a syntax framework. Then, we show how Chiron and MSL can be regarded as a syntax framework. All formal definitions are taken directly from [12].

7.3.1 Definition of a Syntax Framework

Let a *formal language* be a set of expressions each having a unique mathematically precise syntactic structure.

Definition 7.3.1 (Interpreted Language) An *interpreted language* is a triple $I = (L, D_{\text{sem}}, V_{\text{sem}})$ where:

- L is a formal language.
- D_{sem} is a nonempty domain (set) of *semantic values*.
- $V_{\text{sem}} : L \to D_{\text{sem}}$ is a total function, called a *semantic valuation function*, that assigns each expression $e \in L$ a semantic value $V_{\text{sem(e)}} \in D_{\text{sem}}$.

Intuitively, an interpreted language is a formal language with a function that maps each expression to a semantic value.

Example 7.3.2 Let us consider the example of propositional logic. The triple $(\mathsf{Prop}, \mathbb{B}, V)$ is an interpreted language where:

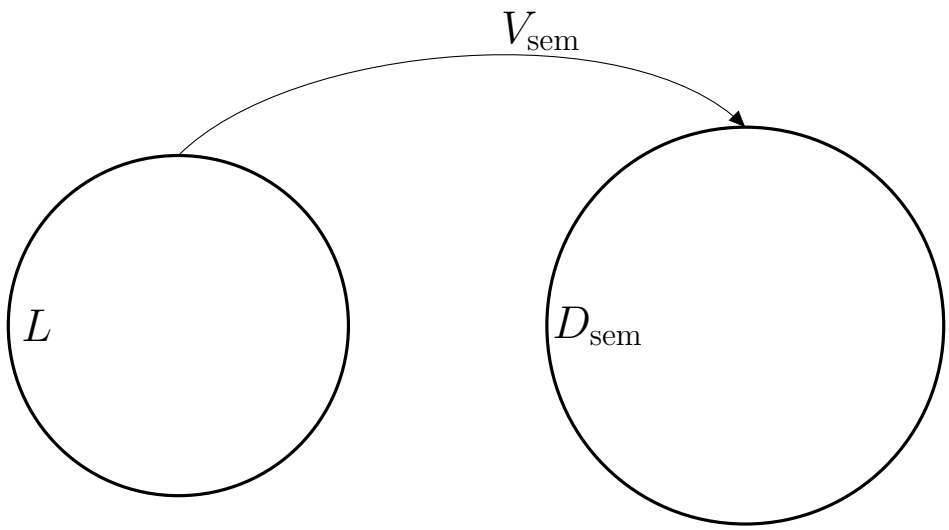

Figure 7.1: An Interpreted Language

- Prop is the set of all propositions,
- $\mathbb{B}$ is the set of truth values, $\mathbb{B} = \{\mathbb{T}, \mathbb{F}\}$,
- V assigns each proposition a truth value defined by inductive definition on the structure of Prop at a given assignment σ of propositional variables to truth values,

Figure 7.1 graphically depicts an interpreted language. □

Definition 7.3.3 (Syntax Representation) Let L be a formal language. A *syntax representation* of L is a pair $R = (D_{\text{syn}}, V_{\text{syn}})$ where:

- D_{syn} is a nonempty domain (set) of syntactic values. Each member of D_{syn} represents a syntactic structure.
- $V_{\text{syn}} : L \to D_{\text{syn}}$ is an injective total function, called a *syntactic valuation function*, that assigns each expression $e \in L$ a syntactic value $V_{\text{syn}}(e) \in D_{\text{syn}}$ such that $V_{\text{syn}}(e)$ represents the syntactic structure of e.

Figure 7.2 graphically depicts a syntax representation.

Intuitively, a syntax representation of a formal language assigns to each expression of the language a syntactic meaning. Unlike the semantic meaning of an expression

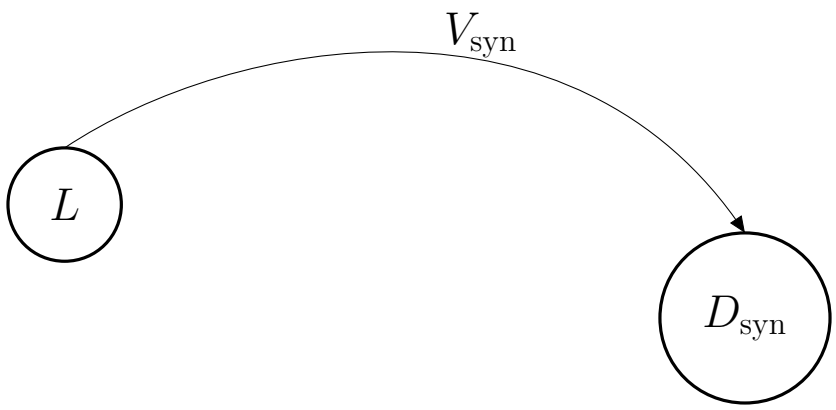

Figure 7.2: A Syntax Representation

which represents the value the expression denotes, the syntactic meaning of an expression represents the structure of the expression.

Hence, an expression may have two meanings:

- The semantic meaning which is the value the expression denotes.
- The syntactic meaning which is the structure of the expression.

In the example of propositional logic, the most straightforward way to represent the syntactic structure of propositions is to use strings. In particular, the pair $R = (\mathsf{String}, \mathsf{toString})$ is a syntax representation of Prop where:

- String is the set of all strings.
- $\mathsf{toString} : \mathsf{Prop} \to \mathsf{String}$ maps a proposition to the string representing it, e.g. it maps the proposition $T \wedge T$ to "$T \wedge T$".

We can also have other syntax representions for Prop. A more structured way is to represent the structure of propositions is to use parse trees. The pair $R = (\mathsf{PropParseTree}, \mathsf{parse})$ is also a syntax representation of Prop where:

- $\mathsf{PropParseTree}$ is the set of parse trees that can be constructed from the grammar of propositions.
- $\mathsf{parse} : \mathsf{Prop} \to \mathsf{PropParseTree}$ maps a proposition to the parse tree representing its structure.

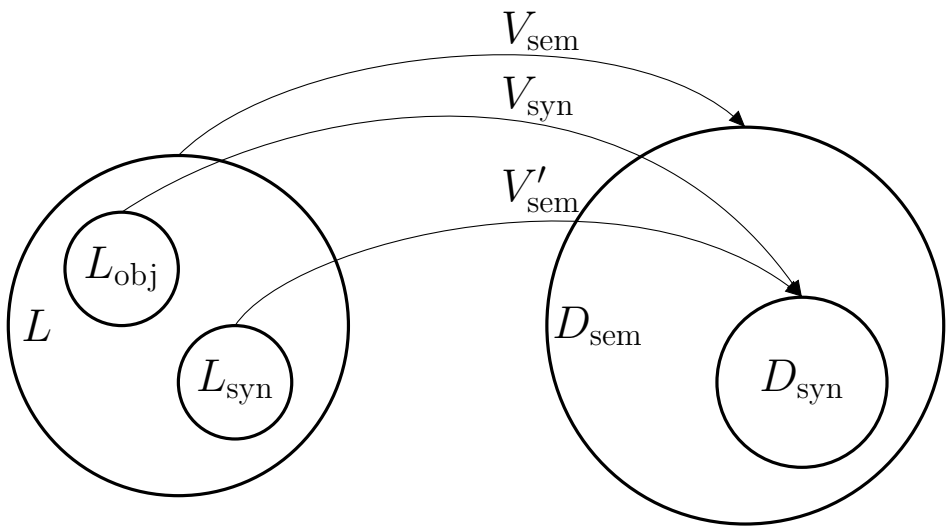

Figure 7.3: A Syntax Language

It is worth mentioning that as opposed to the semantic function V_{sem} defined in interpreted language, the syntactic valuation function V_{syn} needs to be injective. The reason is that two syntactically different expressions may denote the same semantic value but have different syntactic values. For instance, both propositions $T \vee F$ and $F \vee T$ denote T, i.e. have the same semantic value $V_{\text{sem}}(T \vee F) = V_{\text{sem}}(F \vee T) = T$. However, syntactically, they are not the same, i.e. they have different syntactic values. If we were using strings to denote syntactic values of propositions, then $V_{\text{sem}}(T \vee F) =$ "$T \vee F$" $\neq V_{\text{sem}}(F \vee T) =$ "$F \vee T$". Similarly, if we were using parse trees, then the parse tree of $T \vee F$ is different from the parse tree of $F \vee T$.

Definition 7.3.4 (Syntax Language) Let $R = (D_{\text{syn},\text{V}_{\text{syn}}})$ be a syntax representation of a formal language L_{obj}. A *syntax language* for R is a pair (L_{syn}, I) where:

- $I = (L, D_{\text{sem}}, V_{\text{sem}})$ is an interpreted language.
- $L_{\text{obj}} \subseteq L, L_{\text{syn}} \subseteq L$ and $D_{\text{syn}} \subseteq D_{\text{sem}}$.
- V_{sem} restricted to L_{syn} is a total function $V'_{\text{sem}} : L_{\text{syn}} \to D_{\text{syn}}$.

Figure 7.3 graphically depicts a syntax language.

Finally, the following is the definition of a syntax framework:

Definition 7.3.5 (Syntax Framework) Let $I = (L, D_{\text{sem}}, V_{\text{sem}})$ be an interpreted language and L_{obj} be a sublanguage of L. A *syntax framework* for (L_{obj}, I) is a tuple $F = (D_{\text{syn}}, V_{\text{syn}}, L_{\text{syn}}, Q, E)$ where:

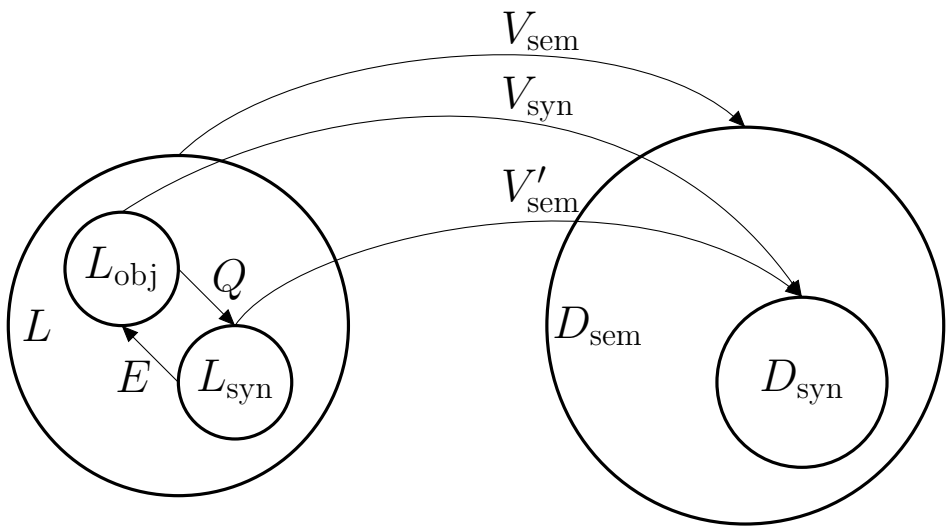

Figure 7.4: A Syntax Framework

(1) $R = (D_{\text{syn}}, V_{\text{syn}})$ is a syntax representation of L_{obj}.

(2) (L_{syn}, I) is syntax language for R.

(3) $Q : L_{\text{obj}} \to L_{\text{syn}}$ is an injective, total function, called a *quotation function*, such that:

Quotation Axiom. For all $e \in L_{\text{obj}}$,

$$V_{\text{sem}}(Q(e)) = V_{\text{syn}}(e).$$

(4) $E : L_{\text{syn}} \to L_{\text{obj}}$ is a (possibly partial) function, called an *evaluation function*, such that:

Evaluation Axiom. For all $e \in L_{\text{syn}}$,

$$V_{\text{sem}}(E(e)) = V_{\text{sem}}(V_{\text{syn}}^{-1}(V_{\text{sem}}(e)))$$

whenever $E(e)$ is defined. □

In Figure 7.3, L_{syn} is the language representing the syntactic structure of the expressions in L_{obj}. The quotation function Q maps an expression in L_{obj} to its syntactic structure in L_{syn}. Conversely, the evaluation function E maps a syntactic structure in L_{syn} to an expression in L_{obj} representing the value that the structure denotes.

7.3.2 Chiron as a Syntax Framework

The paper [12] shows how Chiron can be regarded as a syntax framework as follows:

Let L be a language of Chiron, $\mathcal{E}_L$ be the set of expressions in L, M be a standard model for L, D_M be the set of values in M, V be the valuation function in M, and φ be an assignment into M. Then $I = (\mathcal{E}_L, D_M, V_\varphi)$ is an interpreted language.

D_M includes certain sets called *constructions* that are isomorphic to the syntactic structures of the expressions in $\mathcal{E}_L$. H is a function in M that maps each expression in $\mathcal{E}_L$ to a construction representing it. Let D_{syn} be the range of H and $\mathcal{T}_{\text{syn}}$ be the set of terms a such that $V_\varphi(a) \in D_{\text{syn}}$. For $e \in \mathcal{E}_L$, define $Q(e) = (\mathsf{quote}, e)$. For $a \in \mathcal{T}_{\text{syn}}$, define $E(a)$ as follows:

(1) If $V_\varphi(a)$ is a construction that represents a type and $H^{-1}(V_\varphi(a))$ is eval-free, then $E(a) = (\mathsf{eval}, a, \mathsf{type})$.

(2) If $V_\varphi(a)$ is a construction that represents a term, $H^{-1}(V_\varphi(a))$ is eval-free, and $V_\varphi(H^{-1}(V_\varphi(a))) \neq \bot$, then $E(a) = (\mathsf{eval}, a, \mathsf{C})$.

(3) If $V_\varphi(a)$ is a construction that represents a formula and $H^{-1}(V_\varphi(a))$ is eval-free, then $E(a) = (\mathsf{eval}, a, \mathsf{formula})$.

(4) Otherwise, $E(a)$ is undefined.

Then $F = (D_{\text{syn}}, H, \mathcal{T}_{\text{syn}}, Q, E)$ is a syntax framework for $(\mathcal{E}_L, I)$. This is graphically depicted in Figure 7.5.

As depicted in Figure 7.5, in Chiron, the object language L_{obj} is identified with the language L.

7.3.3 The MathScheme Language as a Syntax Framework

Let L_{MSL} be the set of expressions of MSL. Furthermore, let $T : L_{\text{MSL}} \to \epsilon_{\text{L}}$ be the translation function that translates each MSL expression into a Chiron expression. This is graphically shown in Figure 7.6. If we translate all MSL expressions into Chiron, working in MSL is the same as working in Chiron.

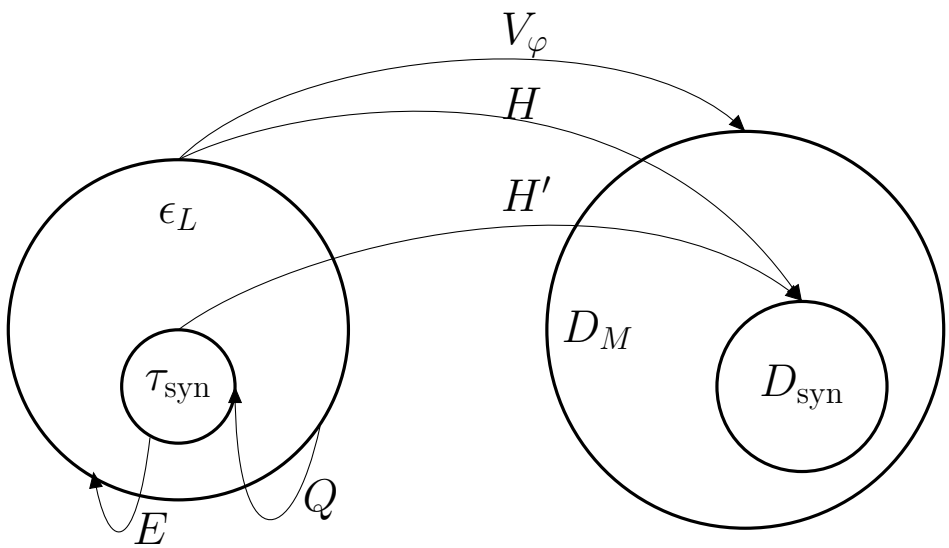

Figure 7.5: Chiron As A Syntax Framework

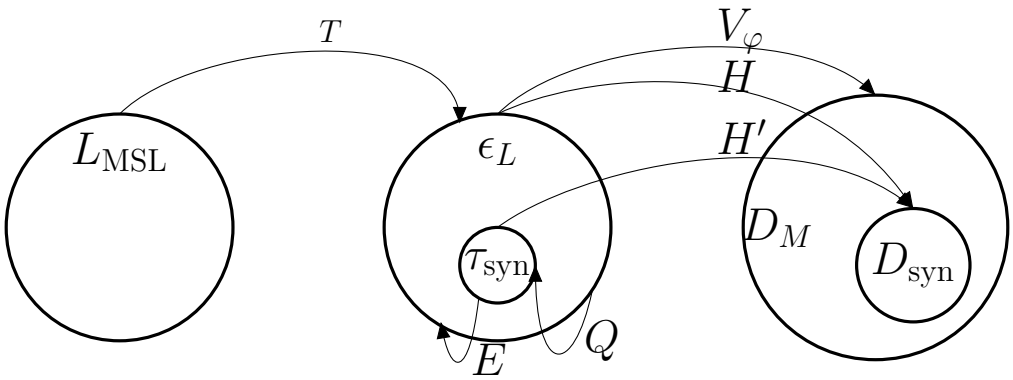

Figure 7.6: MathScheme Language As A Syntax Framework

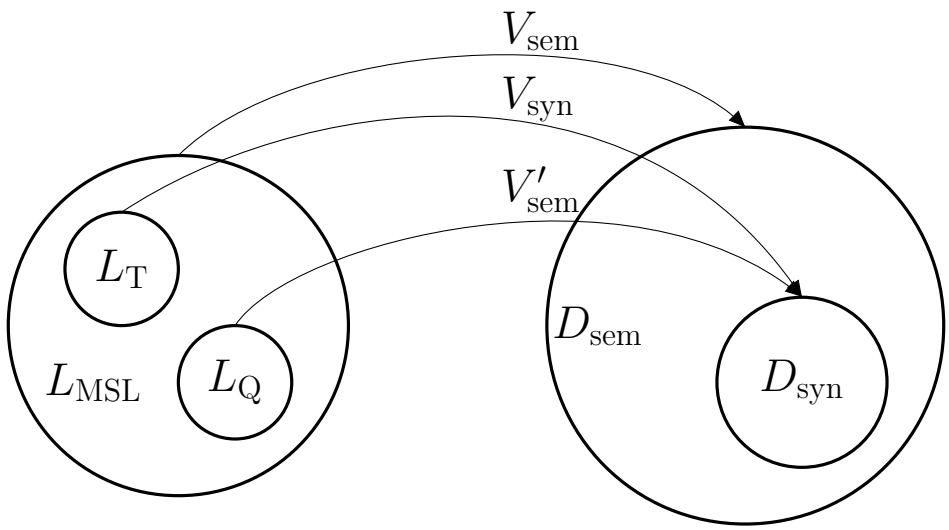

Figure 7.7: Theories in the MathScheme Language

7.4 Reification of the Term Algebra of a Theory as an Inductive Data Type

Let T be a theory formalized in MSL. The concepts of T induce a set of expressions L_T that can be constructed from them which is precisely the term algebra of T. We call the set of all quotations of expressions in L_T L_{Q}, the *quotations set* of T. Furthermore, the set D_{syn} contains all constructions representing expressions of the term algebra L_T. Figure 7.7 graphically illustrates this.

For example, if T is the theory of Bool above, then:

- The term algebra L_{T} contains true, false , **and**(true,false), **and**(**not**(true),or(false,true)) etc.
- The quotations set L_Q contains |^true^|, |^ false ^|, |^**and**(true,false)^|, |^**and**(**not**(true),or(false,true))^| etc.

It is obvious that if a new theory T' is defined as a theory extension of T by adding more concepts, then L_T and L_Q are subsets of L'_T and L'_Q, respectively where L'_T and L'_Q are the term algebra and the quotations set of T'.

Both the term algebra L_T and the quotations set L_Q are expressed in the meta-language of MSL. Moreover, they are isomorphic. Technically, we can reason about the syntactic expressions of T using the quotations in L_{Q}. However, both L_T and

L_{Q} themselves are concepts of the meta-language and cannot be used while reasoning inside MSL.

We can obtain the type of expressions of T by reifying its term algebra L_T. The Reification of L_T is identical to representing D_{syn} on the syntactic level. Dr. O'Connor initially introduced the idea of reifying the term algebra of a 1-sorted theory as an inductive data type. Using the analysis illustrated in Figure 7.7, his idea corresponds to reifying L_T as an inductive data type.

The biggest advantage of reifying L_T as an inductive data type is that an inductive data type is well-structured, as opposed to (say) strings.

The reification algorithm for a 1-sorted theory is simple and can be described as follows:

- Each constant c of the theory becomes an 0-ary data constructor of the inductive data type.
- Each function of arity n ($n > 0$) becomes an n-ary data constructor of the inductive data type.

Example 7.4.1 The term algebra of Bool above, when reified, becomes the following inductive data type:

```
BoolTerm = data X .
    | true  : X
    | false : X
    | and   : (X,X) -> X
    | or    : (X,X) -> X
    | not   : X -> X
```

□

7.4.1 Linking the Reified Term Algebra with the Quotations Set

We see that the term algebra L_T and the quotations set L_Q of a theory are isomorphic but separate sets. Even though the reified inductive data type of the term algebra does capture the term algebra of the theory, it does not represent the quotation sets. Reasoning about the elements of the reified term algebra is not reasoning about

quotations. Consequently, we need to connect the elements of the inductive data type representing the term algebra L_T and the elements of the quotations set L_Q.

One pragmatic way of doing this is to have a transformer that converts an element of the reified inductive data type into the corresponding quotation. The implementation of this transformer is straightforward because we only need to traverse the structure of an input element of the inductive data type and recursively convert it into a quotation. In the example of Bool, the transformer would return |^true^| for true of BoolTerm and |^**and**(true,false)^| for **and**(true,false) of BoolTerm.

7.4.2 The Term Algebra of a Multi-sorted Theory

The construction for reifying the term algebra of a theory as an inductive data type discussed previously only works for 1-sorted theories. However, the construction can be easily extended to multi-sorted theories.

Before introducing the reification algorithm, we take a look at the example of how we can reify the term algebras of the two-sorted theory of a vector space (see Chapter 5 for the definition of a vector space in MSL). The term algebra for the field part can be reified as the following inductive data type:

```
type FieldTerm =
 data X . | + : (X, X) -> X
          | * : (X,X) -> X
          | - : X -> X
          | / : X -> X
          | 0 : X
          | 1 : X
```

However, besides the field, syntactic expressions of the vector space can be built up from vectors as well. Moreover, vectors can built up not only from other vectors but also from field and vector elements as witnessed by $*_V$. Here, $*_V$ takes a field element and a vector element and returns a vector element. That means, the inductive data type of the term algebra of vectors depends on the definition of the term algebra of fields.

We can reify the term algebra of vectors as the following inductive data type:

```
type VectorTerm =
 data Y . | +V : (Y, Y) -> Y
```

```
| *_V  : (FieldTerm, Y) -> Y
| 0_V  : Y
| -_V  : Y -> Y
```

Here, the definition of VectorTerm depends on the definition of FieldTerm.

In general, suppose we want to generate the term algebras from an n-sorted theory T with $n \geq 1$ carrier sets $C_1, \ldots, C_n$. The algorithm for reifiying the term algebras of T is the following:

- For each carrier set C_i, create an inductive data type C_iTerm with bound variable X_i.
- For each constant of type C_i of T, add a 0-ary to C_iTerm.
- For each n-ary function of T whose return type is C_i, add an n-ary data constructor to C_iTerm. Moreover, for each parameter of the function, if its type is C_j, the corresponding parameter type of the data constructor is X_j. Otherwise if it is C_k where $k \neq j$, the parameter type is C_kTerm.

Occasionally, it may happen that we are only interested in term algebras of a certain subset of carrier sets, especially when the theory has a lot of carrier sets. We modify the algorithm above so that it is possible to specify the carrier sets for which the term algebras should be reified.

Suppose we want to generate the term algebras from an n-sorted theory T with $n \geq 1$ carrier sets $C = C_1, \ldots, C_n$. The algorithm for reifiying the term algebras of T for the carrier sets $S = \{S_1, \ldots, S_l\} \subseteq C$ is the following:

- For each carrier set C_i in S , create an inductive data type C_iTerm with bound variable X_i.
- For each constant of type C_i of T, add an 0-ary to C_iTerm.
- For each n-ary function of T whose return type is C_i and whose the types of all parameters are in S, add an n-ary data constructor to C_iTerm. Moreover, for each parameter of the function, if its type is C_j, the corresponding parameter type of the data constructor is X_j. Otherwise if it is C_k where $k \neq j$, the parameter type is C_kTerm.

Example 7.4.2 Suppose we reify the term algebra of VectorSpace for only the carrier set V, the inductive data type is as follows:

```
type VectorTerm =
 data Y . | +_V : (Y, Y) -> Y
          | 0_V : Y
          | -_V : Y -> Y
```

Here, the data constructor $*_V$: (FieldTerm, Y) −> Y is not included because the signature of the original function $*_V$: (F, V) −> V contains F which is not in the set of carrier sets we are interested in. □

7.4.3 Implementation

In Dr. O'Connor's implementation, a term algebra builder called & takes a 1-sorted theory as input and returns an inductive data type representing the term algebra of the theory.

For instance, Bool is expanded to:

```
type BoolTerm = data X .
   | true  : X
   | false : X
   | and   : (X,X) -> X
   | or    : (X,X) -> X
   | not   : X -> X
```

Due to time limit, the reification algorithm for multi-sorted theories has not been implemented. The following gives several suggestion for future implementation:

- The term algebra builder & should be changed to TermAlgebraFrom to be more suggestive. TermAlgebraFrom should take as input a theory T as well as the carrier sets of T whose term algebras we would like to reify.
- The result of TermAlgebraFrom is a set of inductive data types, each of which represents the term algebra of the theory over each input carrier set specified as parameter of TermAlgebraFrom.

7.5 Useful Syntactic Functions

As we said before, the good thing about capturing the term algebra in an inductive data type is that an inductive data type is very well-structured. That allows us to easily define functions that are useful for reasoning about the syntax of expressions.

Length of syntactic expressions is a representative example of a useful tool for reasoning about syntax. We would like to have a length function that takes any term algebra in the form of an inductive data type and returns a natural number representing length of the input.

We know that it is possible to define the length function on any inductive data type recursively based on the data constructors. Concretely, the length function defined on an inductive data type can be systematically defined by pattern matching on each data constructor of the inductive data type:

- length of a constant is 1.
- length of an n-ary data constructor $f(e_1, \ldots, e_n) = 1 + \mathsf{length}(e_1) + \ldots + \mathsf{length}(e_n)$.

It is interesting to remark that in the world of functional programming, length is a *catamorphism* [21] for the inductive data type.

Example 7.5.1 The length function for BoolTerm can be defined as below:

```
BoolExt := Bool extended by {
   type BoolTerm = TermAlgebraFrom(Bool);
   simplify : BoolTerm -> BoolTerm;
   length : BoolTerm -> Nat;
   length x = case x of {
      | true   -> 1
      | false  -> 1
      | and (y z) -> 1 + length y + length z
      | or (y z) -> 1 + length y + length z
      | not y -> 1 + length y
   }
   axiom : forall e : BoolTerm .
           [| e |]_B = [| simplify(e) |]_B;
   theorem : forall e : BoolTerm .
           length(simplify(e)) <= length(e);
}
```

□

In the future, other useful syntactic functions should be identified and generated. The example of length may serve as guidance for this purpose.

7.6 Theory of Syntax

We have seen so far that, lots of useful syntactic machinery can be generated which significantly reduces the burden on the user. That would be very nice if we could define all this machinery in a global *theory of syntax*. This way, whenever we want to reason about syntax, we simply use this theory of syntax. Since term algebra is reified as inductive data type, it would be reasonable to define a *theory of an inductive data type* that contains knowledge and reasoning about inductive data types. These can be reused within the context of the theory of syntax. However, this is outside of the scope of this thesis.

CHAPTER 8

CONCLUSION AND FUTURE WORK

The thesis explains the major techniques for constructing the MathScheme Library. Moreover, the thesis discusses several algebraic constructions we have developed for leveraging the information from existing theories to generate new useful machinery. In particular, we have shown how to reify a theory as a dependent record type and a theory interpretation as a dependent record. We have also explained a method for generating a theory of a homomorphism (as well as epimorphism, monomorphism, isomorphism) and a theory of a substructure from an input theory. Finally, we have shown the technique for reifying the term algebra of a theory as an inductive data type as well as other useful syntactic machinery such as the length function of expressions.

Defining algebraic constructions that can automatically generate new information for the library of formalized mathematics from existing theories is a very powerful idea. That way, we can maximally reuse information to alleviate the burden on the user of having to manually defining various machinery. The developed constructions described in this thesis show the feasibility of the idea and and are ready to be implemented in the MathScheme Library.

Based on the work of this thesis, the following work could be done in the future:

- Since the MathScheme Project shifted from a focus in theories to a focus in theory morphisms, the current MathScheme implementation needs to be overhauled to support theory morphisms. Especially, the MathScheme implementation should support the theory morphism's operations introduced in Chapter 2.

- With the exception of reification of theories as dependent record types, other generation methods described in this thesis have not been implemented yet. They need to be implemented as soon as the MathScheme implementation fully supports theory morphisms.

- We should figure out more useful machinery that can potentially be automatically generated and develop techniques for generating it. The methods described here could serve as model for that purpose.

ACKNOWLEDGMENTS

It is an honor for me to express my gratitude to the people who have directly or indirectly helped and supported me through my research as well as made my time being a graduate student one of the best experiences of my life.

First and foremost, I owe my deepest gratitude to my supervisor, Dr. William M. Farmer, for guiding me through my research and academic life with great competency and patience. I could not wish for a better or friendlier supervisor. Thanks a lot for everything, Bill!

I have been greatly aided by Dr. Jacques Carette and post-doctoral researcher Dr. Russell O'Connor, team members of the MathScheme Project, while working on my research topic. Without Dr. Carette and Dr. O'Connor's explanations of the MathScheme Library and MathScheme implementation as well as lots of creative ideas in many MathScheme meetings and email exchanges, this thesis would not have been completed. Thus, I would like to offer my sincerest thank for their support.

I am deeply grateful to Dr. William M. Farmer, Dr. Wolfram Kahl, Dr. Ridha Khedri and Dr. Jeffery Zucker for their superb graduate courses that helped me improve my understanding of computer science and complete this thesis. My supervisor's CAS760 course on "Logics for Practical Use" provided the background knowledge for understanding various applications of formal logics and the MathScheme project. In particular, Chapter 2 would have not been completed without the knowledge from the course. Dr. Kahl's CAS743 course on "Functional Programming" and CAS706 "Programming Languages" were particularly helpful for me to understand functional programming and the semantics of programming languages. Dr. Khedri's CAS707

course had a part about model theory whose knowledge was helpful for me to complete Chapter 5 and Chapter 6. Dr. Zucker's CAS701 course on "Logic and Discrete Mathematics in Software Engineering" and reading course on the "Theory of Computability' provided me with the background to understand more advanced topics'.

Many friends have made my graduate life an unforgettable experience. Notably, friends from The Philosophy of Computer Science study group: Gordon J.Uszkay, Marc Bender, Pouya Larjani and Valentin Cassano. Our formal discussions on computer science in the meetings and informal discussions in the Phoenix afterward have always been a lot of fun. Marc has really inspired me with his passion for computer science. Thanks to Gordon for sharing his experience of computer science, life and Canadian culture. He has been a very inspirational person. I also greatly enjoyed going to the gym with Akbar Abdiraxmanov, Husam Ibrahim, and Pouya Larjani.

I would like to show my gratitude to Dr. Peter Fleischer, Dr. Bernhard Hollunder, Dr. Friedbert Kaspar, Dr. Berthold Laschinger, Mrs. Brigitte Minderlein from Furtwangen University in Germany for kindly recommending me to the McMaster graduate school.

My final thanks goes to my loving family including my parents and sister back in Vietnam. Thanks to my sister for being such a great sister. Thanks to mom for her love and for teaching me how important education is. Thanks to dad for inspiring me to study abroad and teaching me "a+b=b+a" while I was still in kindergarten. Now, I could express your teaching in a slightly more formal way: "+ is commutative".

APPENDIX : MATHSCHEME LANGUAGE

MathScheme Language (MSL) is an exceedingly rich high-level specification language built on top of Chiron for specifying and relating biform theories. As the time of this writing, MSL is continuously being extended, modified and improved. Nevertheless, its core features are pretty stable. This appendix explains the language features of MSL.

H.1 Conventions

In explaining MSL, the following conventions will be used:

H.1.1 Terminals

Terminals are enclosed in quotation mark, e.g. "0", "A".

H.1.2 Nonterminals

A nonterminal is written in lower case, e.g. <expr>.

H.1.3 Options

An option is represented through square brackets, e.g. [ ].

H.1.4 Alternatives

Alternatives are expressed by a vertical |, e.g. (<letter> | <digit>).

H.1.5 Repetitions

There are two kinds of repetitions. A repetition that must occur at least once is represented by $*$, e.g. <digit>$*$. A repetition that may not occur is represented by $+$, e.g. <digit>$+$.

H.1.6 Comments

Comments are put between (@* This is a comment *@)

H.1.7 Identifiers and Operators

Identifiers are used to name various kinds of concepts in MSL such as theory names, variable names etc. Operators are usual mathematical operators such as $+, -$ etc.

```
<oper>   ::= <symbol>+{<digit>}
<ident>  ::= (<letter> | <digit> | #)
             (<letter> | <digit> | <symbol>)*
<letter> ::= a..z | A..Z
<digit>  ::= 0..9
<symbol> ::= + | * | < | > | ^ | / | \ | # |
             _ | @ | - | ~ | \ | =
```

H.2 Expression

```
<expr> ::= <expr> and <expr>
         | <expr> or <expr>
         | <expr> implies <expr>
         | not <expr>
         | <relation_expr>
         | <expr> in <full_type>
         | <quantifier>
```

```
          | <atom>
          | <application_expr>
          | <oper_expr>
          | <expr> . <ident>
          | lambda <var_spec> . <expr>
          | case <expr> of <cases>
          | if ( <expr> , <expr> , <expr> )
<relation_expr> ::= <expr> = <expr>
<application_expr> ::= atom atom
                     | application_expr atom
<var_spec> ::= <ident_list> : <full_type>
```

Expressions are either terms or formulas. However, MSL's grammar does not differentiate between them. In particular, an expression is one of the following constructs:

- A conjunction of two formulas.
- A disjunction of two formulas.
- A negation of a formula.
- An equality of two expressions.
- A membership relationship of an expression in a type.
- A quantifier.
- An atom.
- A function application.
- An operator expression.
- A lambda abstraction.
- A case expression.
- An if-conditional expression.

A formula can be formed using the usual logical operators: conjunction $\wedge$, disjunction $\vee$, negation $\neg$. Equality checking of two expressions, e.g. e1 = e2, and the membership checking, e.g. 0 **in** nat, are also formulas.

Moreover, MSL provides support for universal quantifier $\forall$ and existential quantifier $\exists$. The production rule for quantifiers is as below:

```
<quantifier> ::= <forall_expr>
               | <exists_expr>
<forall_expr> ::= forall <var_spec>  . <expr>
<exists_expr) ::= exists <var_spec>  . <expr>
                | exists! <var_spec> . <expr>
```

Beside the standard existential quantifier, there is also a unique existential quantifier **exists** !. This can be used to write such statements as "there exists a unique x such that...".

H.2.1 Term

With the exception of atoms that can be either formulas or terms, function applications, operator expressions, constructor selectors, lambda abstraction, case expressions and if-conditional expressions are terms.

The following sections describe them.

H.2.2 Atom

```
<atom> ::= <ident>
         | ( <oper> )
         | <p_strict_expr_list>
         | b_record_list
         | [# <full_type> #] . <ident>
         | ( <expr> )
         | |_ <expr> _|
         | |^ <expr> ^|
         | [[ <expr> ]]_ <full_type>
```

An atom is a formula or a term. It can be either one of the following constructs:

- An identifier.

- An operator.
- A tuple.
- A record.
- A constructor selector of an inductive data type.
- A bracketed expression.
- A marked expression.
- Quotation.
- Term evaluation (to a certain type).

H.2.3 Identifier

An atom can be denoted by an identifier.

H.2.4 Operator

A bracketed operator name is an atom. For instance, (>).

H.2.5 Tuple

```
<atom> ::=
         | ...
         | <p_strict_expr_list>
         | ...
<p_strict_expr_list> ::= ( <expr> , <expr_list> )
<expr_list> ::= <expr>
              | <expr> , <expr_list>
```

A tuple is an itom. It has two or more expressions. For instance, (e_1, e_2, e_3) is a 3-tuple.

H.2.6 Record

```
<atom> ::=
        | ...
        | <b_record_list>
        | ...
<b_record_list> ::= { [record_list] }
<record_list> ::= <ident> = <expr>
                | <oper>  = <expr>
                | <ident> = <expr> , <record_list>
                | <oper>  = <expr> , <record_list>
```

A record is an atom. Each record is an instance of the record type and denoted by a (possibily empty) list of labeled expressions. For instance, {r = 5.0,imag = 2.0}.

H.2.7 Constructor Selector

A constructor selector is an atom. It selects one of the data constructors defined in an inductive data type. For instance, Suppose we have an inductive data type of Peano Arithmetic

```
data X . | zero : X | suc : X -> X
```

Then

```
[data X . | zero : X | suc : X -> X].suc
```

will return the suc data constructor.

H.2.8 Bracketed Expression

An expression enclosed by brackets is an atom. For instance, (e).

H.2.9 Marked Expression

A marked expression an atom. It is used in conjunction with quasiquotation to represent an expression within a quotation that should be evaluated. For instance, $\lceil f\ (\lfloor 2+3 \rfloor) \rceil$ evaluates to $\lceil f\ 5 \rceil$.

H.2.10 Quotation

A quotation is an atom. For instance, given $0 : nat$, |^ 0 ^| denotes the syntactic expression that, when evaluated, denotes the natural number 0.

H.2.11 Term evaluation

A term evaluation is an atom. Term evaluation evaluates a syntactic expression to a term of a certain type. For instance, [[|^ 0 ^|]] _nat evaluates to the natural number 0.

H.2.12 Function Application

```
<expr> ::=
          | ...
          | <application_expr>
          | ...

<application_expr> ::= atom atom
                     | application_expr atom
```

A function application is a term. It is the application of a function to some arguments. For instance, f x, f (x − y) z are function applications.

H.2.13 Operator Expressions

```
<expr> ::=
          | ...
          | <oper_expr>
          | ...
<oper_expr> ::= <expr> inline_oper <atom>
<inline_oper> ::= <oper>
                | <ioper>
<ioper> ::= ` (symbol | goodchars)+
```

An operator expression is a term and is defined by the production rule oper_expr. It represents an expression constructed by applying operators written in infix format. For instance, e ∗ x is an operator expression.

H.2.14 Function Abstraction

```
<expr> ::=
          | ...
          | lambda <var_spec> . <expr>
```

```
          | ...
```

A function abstraction is a term. It is defined using lambda abstraction as usual in lambda calculus. For instance, **lambda** x : Nat. 2∗x.

H.2.15 Case Expression

```
<expr> ::=
          | ...
          | case <expr> of <cases>
          | ...
<cases> ::= { [ case_list ] }
<case_list> ::= <case>
              | case <case_list>
<case> ::= \| <pattern> -> <expr>
<pattern> ::= <base_pat>
            | ( <pattern> )
            | <pattern> <pattern>
            | ( <pattern> , <pattern_list>  )
            | {}
            | { <pat_record_list> }
<base_pat> ::= <ident>
             | <oper>
             | _
<pat_record_list> ::= <ident> = <pattern>
                    | <oper>  = <pattern>
                    | <ident> = pattern , <pat_record_list>
                    | <oper>  = pattern , <pat_record_list>
```

A case expression is a term. Case expressions in MSL are syntactically and semantically almost identical to those in OCaml. A case expression matches an expression with a list of cases.

Each case is of the form pattern −> expression. If the pattern is matched, the result of the entire case expression is the expression following the arrow.

A pattern can be:

- A base pattern that is either an identifier or an operator name.
- A bracketed pattern.

- A function application pattern, e.g. suc 0.
- A tuple pattern, e.g. (e1, e2).
- An empty record pattern {}.
- A non-empty record pattern. For instance, {r = r1, img = r2}.

H.2.16 Definite And Indefinite Description

```
<atom> ::=
         | ...
         | <quantifier>
         | ...
<quantifier> ::=
                | ...
                | <iota_expr>
                | <epsilon_expr>
<iota_expr>     ::= iota <ident> : <full_type> . <expr>
<epsilon_expr> ::= epsilon <ident> : <full_type> . <expr>
```

A definite description is a term and is defined by the production rule iota_expr. It denotes the unique element that satisfies a condition. For instance, iota x : $\mathbb{R}$. $x^3 = -27$ denotes the unique real number x such that $x^3 = -27$ which is -3.

An indefinite description is a term and is defined by the production rule epsilon. It denotes some element that satisfies a condition. For instance, iota$x : \mathbb{R} . x^2 = 4$ denotes some real number x such that $x^2 = 4$ which can be either 2 or -2.

H.3 Type Expression

```
<full_type> ::= type
              | <full_type> -> <full_type>
              | ( <ident> : <full_type> )
              | <typeapp>
              | type_plus ( <tplus> )
              | { <record_field_list> }
              | ( <full_type> )
              | ( <type_seq1> )
```

```
                | ( data <ident> . <typespec>  )
                | power <ident>
                | power ( <full_type> )
                | & <ident>
```

Currently, a type expression is one of the following constructs:

- The word **type**.
- A function type.
- A dependent function type.
- A type application.
- A sum type.
- A record type.
- A tuple type.
- An inductive data type.
- A power type.
- A type of term algebra of a theory.

It is worth noting that if we want to introduce a new type expression construct, such as **TypeFrom** (Chapter 4), it will be first introduced here.

H.3.1 Function Type

```
<full_type> ::=  ...
                | <full_type> -> <full_type>
                |  ...
```

A function type is a type expression and is denoted by the arrow → as usual. For instance, nat -> nat is a function type from naturals to naturals.

H.3.2 Dependent Function Type

```
<full_type> ::= ...
              | ( <ident> : <full_type> )
              | ...
```

A dependent function type $\bigwedge x : \alpha.\beta$ is a type expression. It is a generalization of function types described above. In the normal function type $\alpha \to \beta$, β is independent of α. On the other hand, in a dependent function type $\bigwedge x : \alpha.\beta$, β may depend on x, i.e x may occur freely in β.

For instance, in the the following code, (n : Array (n)) is a dependent functon type. It takes a natural number n and returns the type of arrays of length n.

```
Nat : type;
Array : Nat -> type;
a1 : ( n : Array (n) );
```

H.3.3 Type Applications

```
<full_type> ::= ...
              | <typeapp>
              | ...
<typeapp> ::= <ident>
            | <oper>
            | lift <atom>
            | <ident> <full_type>
```

A type application is a type expression. **lift** <atom>+ **lift** what the parser otherwise considers to be an expression to instead be interpreted as a type, e.g. operator names. Moreover, complete expressions can be entered as types as well.

H.3.4 Sum Type

```
<full_type> ::= ...
              | type_plus ( <tplus> )
              | ...
<tplus> ::= <full_type>
          | <full_type> , <tplus>
```

A sum type is a type expression. **type_plus** takes one or more types and creates a sum type from them. It corresponds to disjoint unions in mathematics and the | in

OCaml. For instance, in the example below, **type_plus** (Nat, Real) is a sum type of Nat and Real.

```
n : type_plus ( Nat, Real )
```

H.3.5 Record Type

```
<full_type> ::= ...
              | { <record_field_list> }
              | ...
<record_field_list> ::= <field_sig>
                      | <field_sig> , <record_field_list>
<field_sig> ::= <ident> : <full_type>
```

A record type is a type expression and is defined by a list of record fields enclosed in curly brackets. For instance, $\{r : \mathbb{R}, \mathrm{img} : \mathbb{R}\}$ is a record type.

H.3.6 Tuple Type

```
<full_type> ::| ...
              | ( <type_seq1> )
              | ...
<type_seq1> ::= <full_type> , <type_seq0>
<type_seq0> ::= <full_type>
              | <full_type> , <type_seq0>
```

A tuple type is a type expression and defined by a list of types enclosed in brackets. There are at least two types in that list. For instance, (nat, nat, nat) is a 3-tuple of naturals.

H.3.7 Inductive Data Type

```
<full_type> ::= | ...
                | ( data <ident> . <typespec>  )
                | ...
<typespec> ::= | <field_sig>
               | <field_sig> \| <typespec>
<field_sig> ::= <ident> : <full_type>
```

An inductive data type is a type expression. It consists of a bound variable and a list of data constructors. For instance, **data** X . 0 : X | suc : X –> X is an inductive data type of Peano Arithmetic. in detail.

H.3.8 Power Type

```
<full_type> ::| ...
              | power <ident>
              | power ( <full_type> )
              | ...
```

A power type is a type expression. A power type of a type t is the type whose elements are subtypes of t. For instance, **power** $\mathbb{R}$ is the power type of $\mathbb{R}$ whose elements are subtypes of $\mathbb{R}$.

H.3.9 Type of Term Algebras of a Theory

```
<full_type> ::| ...
              | & <ident>
              | ...
```

A type of term algebra of a biform theory is a type expression. In the current implementation, & is a term algebra builder. It takes a biform theory as input and creates an inductive data type containing all syntactic expressions that can be constructed using the constants and functions from the input biform theory. For instance, &Nat.

H.4 Concepts

```
<concept_declaration>  ::=
   (Concept | concept | Concepts | concepts) <tident_list>;
<tident_list> ::= tident
                | tident <tident_list>
<tident> ::= <ident_list> : full_type
           | ( <ident> )  : full_type
<ident_list> ::= <ident>
               | <ident> <ident_list>
```

Concepts are used to represent mathematical concepts in a biform theory. A concept can be a list of types, functions of declared types or constants of declared

types etc. For instance, in the theory of a group, the carrier set G, the binary operator $*$, the neutral element e and the inverse function inv should be defined as concepts:

```
Group := Theory {
  Concepts
    G   : type;
    *   : (G, G) -> G;
    e   : G;
    inv : G -> G;
    ...
}
```

H.5 Facts

```
<single_fact_list> ::= <single_fact> ;
                     | <single_fact> ; <single_fact_list>
<single_fact> ::= axiom <ident> := <expr>
                | axiom <expr>
                | axiom : <expr>
                | axiom <ident> : <expr>
                | theorem <ident> := <expr>
                | theorem <expr>
                | theorem : <expr>
                | theorem <ident> : <expr>
```

A fact is either an axiom or a theorem being a logical statement over the concepts declared within the same biform theory. For instance, the three group axioms: $*$ is associative, e is the neutral element and inv is the inverse can be defined as facts of Group as follows:

```
Group := Theory {
  Concepts
    G   : type;
    *   : (G, G) -> G;
    e   : G;
    inv : G -> G;
  Facts
```

```
axiom: forall x, y, z: G . (x * y) * z = x * (y * z);
axiom: forall x : G . (x * e = x) and (e * x = x);
axiom: forall x : G . x * (inv(x)) = e;  }
```

H.6 Declaration

```
<declaration> ::= <single_declaration> ; <declaration>
                | <single_declaration> ;
                | <single_declaration>
<single_declaration> ::= <typ_declaration>
                       | <type_defn>
                       | <func_defn_declaration>
                       | <axiom_declaration>
                       | <induct_declaration>
                       | <concept_declaration>
                       | <var_declaration>
                       | <fact_declaration>
                       | <defnblock_declaration>
```

A declaration is one of the following constructs:

- A type declaration.
- A type definition.
- A function definition declaration.
- An axiom declaration.
- An inductive data type declaration.
- A concept declaration.
- A variable declaration.
- A fact declaration.
- A definition block declaration.

H.6.1 Type Declaration

```
<single_declaration> ::=
                        | <typ_declaration>
                        | ...
<typ_declaration> ::= <tident>
```

A type declaration assigns to a list of identifiers a type. According to the production rule, a type declaration (typ_declaration) is a type identifier (tident) which is an entry in a concept as described in section H.4. This means, we can declare types within or outside of a concept. In the later case, type declaration is parsed to the TypeDecl data constructor using the typ_declaration rule.

For instance, in Group above, we can move the declarations of G and $*$ out of **Concepts**:

```
Group := Theory {
  Concepts
    e    : G;
    inv  : G -> G;
   ;
    G : type;
    * : (G, G) -> G;
  Facts
    axiom: forall x, y, z: G . (x * y) * z = x * (y * z);
    axiom: forall x : G . (x * e = x) and (e * x = x);
    axiom: forall x : G . x * (inv(x)) = e;
}
```

Then G and $*$ are parsed to TypeDecl internal representations.

H.6.2 Type Definition

```
<single_declaration> ::=
                        | <type_defn>
                        | ...
<type_defn> ::= type <ident> = <top_full_type>
              | <ident> : <full_type>
<top_full_type> ::= <full_type>
```

```
| data <ident> . <typespec>
```

A top level type (top_full_type) is either a full type or an inline inductive data type. A type definition is either a type synonym,
i.e. **type** <ident> = <top_full_type>, or a declaration that an identifier is of a certain type, i.e. **data** <ident> . <typespec>

For instance,

```
type Nat = data X . zero : X | suc : X -> X;
```

Here, Nat is a type synonym for an inline inductive data type containing two data constructor zero and suc.

H.6.3 Function Definition Declaration

```
<func_defn_declaration> ::= <func_defn>
<func_defn> ::= <param_decl> = <expr>
<param_decl> ::= <ident>
               | <ident> <p_ident_list>
               | <ident> <inline_oper> <ident>
<p_ident_list> ::= ( <ident_list> )
```

A function definition declaration is a function definition. The defined function may have no argument e.g. f or a list of arguments e.g. add m n written in prefix. It may also be defined as an infix operator e.g. $+m\,n$.

For instance, in the following PeanoArithmetic theory, add is defined as a binary function in prefix notation.

```
PeanoArithmetic := Theory {
        Inductive Nat
         | zero : Nat
         | suc  : Nat -> Nat
        ;
        add : (Nat, Nat) -> Nat;
        add (m, n) = case n of {
                        | zero -> m
                        | suc p -> suc (add (m, p))
                                                    };
}
```

We can also use the infix notation of addition + instead:

```
PeanoArithmetic := Theory {
        Inductive Nat
         | zero  : Nat
         | suc   : Nat -> Nat
        ;
        + : (Nat, Nat) -> Nat;
        m + n = case n of {
                    | zero -> m
                    | suc p -> suc (add (m, p))
                                   };
}
```

H.6.4 Axiom Declaration

```
<axiom_declaration> ::= <single_fact>
```

An axiom declaration is simply a single fact as described in section H.5.

H.6.5 Inductive Data Type Declaration

```
<induct_declaration> ::= Inductive <ident> <typespec>
                       | Inductive <ident> \| <typespec>
```

An inductive data type declaration **Inductive** is a shorthand for declaring an inductive data type.

For instance, in the PeanoArithmetic theory above, **Inductive** Nat is used as a shorthand to define an inductive data type plus the declarations of the functions corresponding to the data constructors.

For instance, the definition

```
Inductive Nat
   | zero  : Nat
   | suc   : Nat -> Nat
```

is a shorthand for the following definition in expanded form:

```
type Nat = data X. zero : X | suc : X -> X;
zero : Nat;
zero = [# Nat #].zero;
```

```
suc : Nat -> Nat;
suc = [# Nat #].suc;
```

H.6.6 Concept Declaration

A concept declaration is a concept as described in section H.4.

H.6.7 Variable Declaration

```
<var_declaration> ::= (variable | variables) <tident_list2>
<tident_list2> ::= <tident>
                 | <tident> , <tident_list2>
<tident> ::= <ident_list> : <full_type>
           | <pident> : <full_type>
<pident> ::= ( <ident> )
```

A variable declaration declares a list of variables to be of a certain type.

For instance, we can declare three variables one, two, three of type Nat as below:

```
PeanoArithmetic := Theory {
        Inductive Nat
         | zero : Nat
         | suc  : Nat -> Nat
        ;
        ...
        variables one, two, three : Nat;
```

H.6.8 Fact Declaration

```
<fact_declaration> ::= Fact <single_fact_list>
```

A fact declaration is a list of facts which is described in section H.5.

H.6.9 Definition Block Declaration

```
<defnblock_declaration> ::= definition <func_defn_list>
<func_defn_list> ::= <func_defn> ;
                   | <func_defn> ; <func_defn_list>
```

A definition block declaration is a list of function definitions described above.

H.7 Theory Expression

```
<thy_expr> ::= <ident>
             | Theory {}
             | Theory { declaration }
             | ( <thy_expr> )
             | <thy_extension>
             | Theory ( <top_type> ) { <declaration> }
             | <ident> ( <ident> )
             | <thy_expr> [ <id_eq_list> ]
             | (combine | combines) <thy_expr_list> along <thy_expr>
```

A theory expression is one of the following constructs:

- An identifier referring to a theory name.
- An empty theory.
- A basic theory containing declarations in it.
- A bracketed theory expression.
- A biform extension.
- A parameterized theory.
- A theory application.
- A theory renaming.
- A theory combination.

H.7.1 Theory Name

A theory name is a theory expression. It shall refer to an already declared theory.

H.7.2 Empty Theory

An empty theory is a theory expression. It contains no declarations in it.

For instance, the Empty theory is defined as below

```
Empty := Theory {}
```

H.7.3 Theory Extension

```
<thy_expr> ::=
              | ...
              | <thy_extension>
              | ...
<thy_extension> ::= <thy_expr> extended { <declaration> }
                  | <thy_expr> extended by { <declaration> }
                  | <thy_expr> extended conservatively by
                                        { <declaration> }
```

A theory extension is a theory expression. It is the theory resulting from enhancing a theory with further declarations. Declarations are explained in section H.6.

For instance, the following Carrier theory is a theory extension of the Empty theory mentioned previoulsy by adding a type declaration U:

```
Carrier := Empty extended by
{
   U : type
}
```

H.7.4 Parameterized Theory

```
<thy_expr> ::=
              | ...
              | Theory ( <top_type> ) { <declaration> }
              | ...
<top_type> ::= <thyident>
              | <thyident> , <top_type>
<thy_expr> extended { <declaration> }
```

A parameterized theory (also called functor) is a theory expression. However, it is currently not implemented yet.

H.7.5 Theory Application

```
<thy_expr> ::=
              | ...
              | <ident> ( <ident> )
```

```
                | ...
```

A theory application is a theory expression. It is of the form **Theory**(NamedArrow)

H.7.6 Theory Renaming

```
<thy_expr> ::=
                | ...
                | <thy_expr> [ <id_eq_list> ]
                | ...
<id_eq_list> ::= <id_eq>
                  | <id_eq> , <id_eq_list>
<id_eq> ::= <io> = <io>
          | <io> |-> <io>
<io> ::= <ident>
       | <oper>
```

A theory renaming is a theory expression. It is the theory resulting from renaming identifiers and operators in the source theory.

For instance, let BinaryRelation be the theory of binary relation:

```
BinaryRelation := Carrier extended by
{
   R : (U, U)?
}
```

R : (U, U)? is a a just short form for R : (U, U) -> Bool.

Then, an OrderRelation is a binary relation where the more specialized symbol <= is used instead of R. In other words, OrderRelation is the theory resulting from renaming R to <= in BinaryRelation as follows:

```
OrderRelation := BinaryRelation[ R |-> <= ]
```

H.7.7 Theory Combination

```
<thy_expr> ::=
                | ...
                | (combine | combines) <thy_expr_list>
                  (along | over) <thy_expr>
                | ...
```

```
<thy_expr_list> ::= thy_expr
                  | thy_expr , <thy_expr_list>
```

A theory combination is a theory expression. It is the theory resulting from combining a list of theories, represented by thy_expr_list , over a theory, represented by thy_expr.

For instance, let ReflexiveOrderRelation be the theory of order relation being reflexive:

```
ReflexiveOrderRelation := OrderRelation extended by
{
   axiom forall x : U . x <= x
}
```

and ReflexiveOrderRelation be the theory of order relation being transitive:

```
TransitiveOrderRelation := OrderRelation extended by
{
   axiom forall x, y, z : U . (x <= y and y <= z implies x <= z)
}
```

The theory of Preorder can be defined by combining ReflexiveOrderRelation and TransitiveOrderRelation over OrderRelation as below:

```
Preorder := combine ReflexiveOrderRelation, TransitiveOrderRelation
            over OrderRelation
```

In the expanded form, Preorder would look as follows:

```
Preorder := Theory
{
   U  : type;
   <= : (U, U)?;
   axiom forall x : U . x <= x;
   axiom forall x, y, z : U . (x <= y and y <= z implies x <= z);
}
```

H.8 Theory Declaration

A theory declaration is one of the following constructs:

- A declaration of a theory identifier.

- A property declaration.
- An injection.
- A theory instance of a parameterized theory.

```
<theory_declaration> ::= <ident> := <thy_expr>
                       | property <param_decl> := expr ;
                       | <ident> := <inject>
                       | <ident> := instance <ident> of <ident>
                                    via [<id_eq_list>]
```

H.8.1 Declaration of Theory Identifier

An identifier can be declared to refer to a theory expression.

For instance, in the example of Group biform theory above:

```
Group := Theory {
    ...
}
```

Group is an identifier that refers to the theory declared via the keyword **Theory**.

H.8.2 Property

Properties are similar to macros in programming languages. They provide us with a convenient construct to write down such properties as associativity and communitivity while defining biform theories.

For instance,

```
property leftAction(**,++) :=
    forall x: leftDomain((**)). forall y : rightDomain((**)) .
    forall z : rightDomain((++)).
        (x ** y) ++ z = x ++ (y ++ z);
property associative(**) := leftAction((**),(**));
```

Here, leftAction and associative are properties. Whenever we want to say that a certain binary operator $*$ is associative, we can use associative $(*)$ instead of the more verbose form **forall** x, y, z ... The expander will expand properties to their verbose form.

H.8.3 Injection

An injection is a special kind theory morphisms (Chapter 2). We can name injections since they are useful for creating instances. MSL support for theory injections will be extended in the future.

H.8.4 Theory Instance of Parameterized Theory

Parameterized theory is currently not supported yet.

BIBLIOGRAPHY

[1] O. Caml. Home page at http://caml.inria.fr/.

[2] J. Carette and W. M. Farmer, "High-level theories," in *Intelligent Computer Mathematics* (A. Autexier and et al., eds.), vol. 5144 of *Lecture Notes in Computer Science*, pp. 232–245, Springer-Verlag, 2008.

[3] J. Carette, W. M. Farmer, F. Jeremic, V. Maccio, R. O'Connor, and Q. M. Tran, "The mathscheme library: Some preliminary experiments," the 2011 conference on intelligent computer mathematics, University of Bologna, Italy, 2011. Forthcoming.

[4] J. Carette, W. M. Farmer, and R. O'Connor, "Mathscheme: Project description," in *Lecture Notes in Computer Science* (J. H. Davenport, W. M. Farmer, F. Rabe, and J. Urban, eds.), vol. 6824, (Bertinoro, Italy), pp. 287–288, Springer-Verlag, July 2011.

[5] N. G. de Bruijn, "A survey of the project AUTOMATH," in *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* (J. P. Seldin and J. R. Hindley, eds.), pp. 579–606, Academic Press, 1980.

[6] W. M. Farmer, "Theory interpretation in simple type theory," in *Higher-Order Algebra, Logic, and Term Rewriting* (J. H. et al., ed.), vol. 816 of *Lecture Notes in Computer Science*, pp. 96–123, Springer-Verlag, 1994.

[7] W. M. Farmer, "A proposal for the development of an interactive mathematics laboratory for mathematics education," in *CADE-17 Workshop on Deduction Systems for Mathematics Education* (E. Melis, ed.), pp. 20–25, 2000.

[8] W. M. Farmer, "Biform theories in Chiron," in *Towards Mechanized Mathematical Assistants* (M. Kauers, M. Kerber, R. R. Miner, and W. Windsteiger, eds.), vol. 4573 of *Lecture Notes in Computer Science*, pp. 66–79, Springer-Verlag, 2007.

[9] W. M. Farmer, "Chiron: A set theory with types, undefinedness, quotation, and evaluation," SQRL Report No. 38, McMaster University, 2007. Revised 2011.

[10] W. M. Farmer, "Modules for a large library of formalized mathematics," in *AMS Special Session on Formal Mathematics for Mathematicians: Developing Large Repositories of Advanced Mathematics*, 2011.

[11] 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.

[12] W. M. Farmer and P. Larjani, "Frameworks for reasoning about syntax that utilize quotation and evaluation," tech. rep., McMaster University, 2011. preprint, 33 pp.

[13] W. M. Farmer and M. von Mohrenschildt, "An overview of a Formal Framework for Managing Mathematics," *Annals of Mathematics and Artificial Intelligence*, vol. 38, pp. 165–191, 2003.

[14] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "Little theories," in *Automated Deduction | CADE-11, volume 607 of Lecture Notes in Computer Science*, pp. 567–581, Springer-Verlag, 1992.

[15] C. L. for Practical Use. http://hygelac.cas.mcmaster.ca/courses/CAS-760-10/slides/01-review-logic.pdf.

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

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

[18] MathScheme. Home page at http://www.cas.mcmaster.ca/research/mathscheme/.

[19] MathSchemeRepository. Formalizations of Abstract Algebra at /trunk/doc/biform-theories/Algebra.

[20] MathSchemeRepository. Formalizations of Concrete Theories at /trunk/doc/biform-theories/machine.

[21] E. Meijer, M. Fokkinga, and R. Paterson, "Functional programming with bananas, lenses, envelopes and barbed wire," pp. 124–144, Springer-Verlag, 1991.

[22] L. C. Paulson, *Isabelle: A Generic Theorem Prover*. Springer, 1994.

[23] P. Rudnicki, "An overview of the MIZAR project," tech. rep., Department of Computing Science, University of Alberta, 1992.

[24] M. H. B. Srensen and P. Urzyczyn, "Lectures on the curry-howard isomorphism," July 2006.

[25] A. N. Whitehead and B. Russell, *Principia Mathematica*. Cambridge University Press, 1910–13. Paperback version to section *56 published in 1964.

[26] J. Xu, *Mei A Module System for Mechanized Mathematics Systems*. PhD thesis, McMaster University, 2008.