A Case Study on Formalizing Algebra in a Module System

Stefania Dumbrava, Fulya Horozal, Kristina Sojakova
Jacobs University
D-28759, Bremen, Germany
s.dumbrava@jacobs-university.de, f.horozal@jacobs-university.de,
k.sojakova@cjacobs-university.de

ABSTRACT

We present a case study on a modular formal representation of algebra in the recently developed module system for the Twelf implementation of the Edinburgh Logical Framework LF. The module system employs signature morphisms as its main primitive concept, which makes it particularly useful to reason about structural translations between mathematical concepts. The mathematical content is encoded in the usual way using LF's higher order abstract syntax and judgments-as-types paradigm, but using the module system to treat all algebraic structures independently. Signature morphisms are used to give an explicit yet simple representation of modular dependency between the algebraic structures. Our results demonstrate the feasibility of comprehensively formalizing large-scale theorems and proofs and thus promise significant future applications.

Categories and Subject Descriptors

I.2.4 [**Knowledge Representation Formalisms and Methods**]: Representation languages

General Terms

Design, Languages, Verification

Keywords

Logical frameworks, Twelf, modularity, encodings, abstract algebra

1. INTRODUCTION

In recent decades, a large amount of mathematics has been formalized in various proof systems, which created large libraries of mechanically verified mathematical knowledge. Several proof systems have developed module systems to manage their large mathematical developments. These module systems often follow the "little theories approach" proposed in [FGT92], in which separate contexts are represented by separate theories. These permit to encapsulate mathematical theories and re-use them in different contexts.

For a long time, Coq [Coq] used a sectioning mechanism that separates its formalizations into different sections to which definitions, theorems and proofs are made local. An ML-like module system for Coq was later implemented (see [Chr03]), which conveniently lets a user define parametrized theories or data structures to be easily used in other formalizations.

Locales ([KW98]) are the modules of Isabelle [Pau94] for efficient theory management that group relevant mathematical theories together. They are similar to the sections in Coq, but provide additional features. For instance, an already existing locale can be opened again to add theorems to its content. Locales are integrated with Isabelle's language for readable proof documents (Isar [Wen99]) and interpreted in the context of proofs.

Recently, a module system for the Twelf implementation [PS99] of the Edinburgh Logical Framework LF [HHP93] was developed (see [RS09]). The Twelf module system employs signatures and signature morphisms as its main primitive concepts. Mathematical theories are formalized in signatures as a collection of constant declarations and constant definitions. The Twelf module system follows the approach of using signature morphisms to represent structural relationships between mathematical theories (see [Far00]). This leads to the notion of signature graphs, which are a simple and scalable means to relate signature to one another.

In this paper, we present a case study on the formalization of mathematical theories from algebra in the Twelf module system. Algebraic structures are particularly suitable for modular representation as there is inherently a large amount of sharing involved, and thus are suitable for an evaluation of a module system. This case study is one of the initial extensive applications of the Twelf module system for mathematical theories (the other application is on the representation of proof and model theory of first-order logic, see [HR09]).

This paper is organized as follows. We present the Twelf system and its module system in Sect. 2. In Sect. 3, we present our case study, in particular, the encoding of algebraic structures in Sec. 3.1 and that of lattices in Sect. 3.2. We summarize our results and discuss future work in Sect. 4.

2. THE TWELF SYSTEM

The Twelf system is an implementation of the logical framework LF designed as a meta-language for the representation of deductive systems. It is a dependent type theory

MLPA CADE 2009, Montreal, Canada

with typed terms and kinded type families.

Kinds:	K	::=	`type` $\mid A \to K$
Type families:	A, B	::=	$a \mid A\ M \mid \Pi_{x:A} B \mid A \to B$
Objects:	M, N	::=	$c \mid x \mid \lambda_{x:A} M \mid M\ N$

Twelf features the dependent product type constructor $\Pi_{x:A} B$ and its introductory axiom, the λ-binder $\lambda_{x:A} M$. As usual, application is written as juxtaposition $M\ N$. $A \to B$ abbreviates $\Pi_{x:A} B$ if x does not occur freely in B. Type families are kinded by kinds, where types are the type families kinded by `type`, and objects are typed by types.

Twelf signatures contain declarations of type- or object-level constants. Constants are declared in the form of declarations $a : K$ or $c : A$, or definitions $a : K = A$ or $c : A = M$. Variables $x : A$ are typed, never kinded.

Signature morphisms define mappings between signatures and come in two flavors: *structures*, which copy and instantiate a signature S into T, and *views*, which translate from a signature S to T. Readers familiar with modular theory development languages such as development graphs ([AHMS99]) will recognize structures as definitional theory morphisms and views as postulated theory morphisms.

A modular Twelf specification is a sequence of signature declarations D_T and view declarations D_v. Signatures are sequences of constant declarations D_c, structure declarations D_s and signature inclusions D_I. Similarly, views are sequences of assignments to constants and structures. These are shown below in terms of Twelf's concrete syntax. Keywords are introduced with `%` and precede all declarations except for constant declarations.

`Start:`	S	::= $D_T \mid D_v$
`Signatures:`	D_T	::= `%sig T = {`$(D_c \mid D_s \mid D_I)^*$`}.`
`Views:`	D_v	::= `%view` v `:` S `->` T `=` `{`$(c$ `:=` $C \mid$ `%struct` s `:=` $\mu)^*$`}.`
`Inclusions:`	D_I	::= `%include` T`.`
`Constants:`	D_c	::= c `:` C`.` $\mid c$ `:` C `=` C`.`
`Structures:`	D_s	::= `%struct` s `:` S `=` `{`$(c$ `:=` C`.` $\mid$ `%struct` s `:=` μ`.)^*}.`
`Terms:`	C	::= `type` $\mid c \mid x \mid C$ `->` $C \mid C\ C$ $\mid$ `{`x`:`C`}` $C \mid$ `[`x`:`C`]` C
`Morphisms:`	μ	::= $v \mid s \mid \mu\ \mu$

Finally, there are two classes of expressions. Firstly, terms are normal LF expressions where the Π-binder is written with braces `{}` and the λ-binder with square brackets `[]`. Secondly, morphisms are expressions that translate between signatures: Each structure s or view v induces a signature morphism, and the juxtaposition $\mu\ \mu'$ of morphisms represents their diagram-order composition. Morphisms preserve all judgments regarding well-formedness, typing, and equality (see [HST94, RS09] for the preservation results).

A *view* declaration encodes a translation between two signatures. A view occurs on toplevel and gives domain and codomain explicitly. It must instantiate all constants (except those that have definitions) of the domain signature with expressions of the codomain signature. This induces a signature morphism in the obvious way.

A *structure* declaration `%struct` s `:` S `= {}` occurring in the signature T represents an inheritance relation from S to T: It adds a copy of S to T. The copied constants are accessible by qualified names formed by prefixing the structure name. Then the declaration of a structure s induces a signature morphism from the instantiated signature S to its containing signature, which maps every constant c of S to $s.c$. A structure declaration may also carry assignments, which are used to translate S when copying it into T.

```
%sig FOL = {
  o     : type.
  i     : type.
  true  : o.
  false : o.
  ¬ : o -> o.
  ...
}.
%sig FOLEQ = {
  %struct fol : FOL.
  == : fol.i -> fol.i -> fol.o.
}.
```

For example, consider the following signature declarations, which we use for our encodings. The signature `FOL` contains a type o for propositions and a type i for first-order individuals. The terms `truth` and `false` represent the truth values for propositions. `FOL` encodes the logical connectives as expected (e.g., ¬ encodes the unary connective for negation). The signature `FOLEQ` encodes first-order logic with equality by inheriting from `FOL` via a structure called `fol` and adds a symbol for equality.

Then the signature `FOLEQPF` encodes natural deduction style proof rules for the logical connectives and quantifiers in `FOL`. Firstly, it inherits from `FOL` and `FOLEQ` where the instantiation `%struct fol := fol.` in the structure `foleq` works as follows: The left side of the instantiation is a symbol declared in the domain signature – here: `FOLEQ` – and the right side is an expression over the codomain signature – here: `FOLEQPF`. This instantiation has the effect of a sharing declaration: The two structures `fol` and `foleq.fol` inheriting from `FOL` are identified. Secondly, it declares the constant ⊢ as o-indexed type family. This type family exemplifies how logic encodings in LF represent judgments as types and derivations as objects: Objects of type $\vdash A$ represent derivations of the judgment "A is true". Then it declares constants that encode the introduction and elimination rules of connectives and quantifiers (e.g., `notI` and `notE` encode the introduction and elimination rules for negation).

```
%sig FOLEQPF = {
  %struct   fol : FOL.
  %struct foleq : FOLEQ = {%struct fol := fol.}.
  ⊢ : o -> type.
  notI : (⊢ A -> {B} ⊢ B) -> ⊢ (fol.¬ A).
  notE : ⊢ A -> ⊢ (fol.¬ A) -> {B} ⊢ B.
  ...
}.
```

Note that we use implicit arguments: Upper case free variables in declarations are assumed be implicitly Π-bound on the outside. This has the effect of free parameters. For example, in `notE`, the variable `A` is free. The verbal reading of the rule is "For any A, if A is true and $\neg A$ is true, then for all B we have B is true ".

Twelf allows one to omit the qualified names of copied constants in a signature using the keyword `%open`. For instance, in `FOLEQPF`, if we consider the following structure declaration `%struct fol : FOL %open ¬.`, then it is possible to refer to the negation connective simply as ¬. The semantics of `%open` is that if s is a structure from signature S occurring in signature T, then `%open` c, for a constant c of S, is equivalent to defining a new constant c as $s.c$ in T. `%open` can be used for inclusion as well in a similar way.

Twelf computes the semantics of the modular signatures by elaborating them to the non-modular syntax. For example, `FOLEQPF` is equivalent to

```
%sig FOLEQPF2 = {
  fol.o     : type.
  fol.i     : type.
  fol.true  : o.
  fol.false : o.
```

```
  fol.¬ : o -> o.
  ...
  foleq.fol.o    : type  = fol.o.
  foleq.fol.i    : type  = fol.i.
  foleq.fol.true : fol.o = fol.true.
  ...
  ⊢ : fol.o -> type.
  notI : (⊢ A -> {B} ⊢ B) -> ⊢ (fol.¬ A).
  notE : ⊢ (fol.¬ A) -> ⊢ A -> {B} ⊢ B.
}.
```

For simplicity, we will use the non-modular version above as the meta-theory of our Twelf encodings and refer to it simply as `FOL`.

A special case of inheritance relation is the *include declaration*. The declaration `%include` S occurring in T creates an inclusion from S to T. This is similar to a structure declaration but simpler and only possible in certain cases. If a signature is included in multiple ways, all inclusions are identified. Therefore, a symbol c included from S into T is identified uniquely by the name $S..c$.

3. MODULAR REPRESENTATION OF ABSTRACT ALGEBRA AND LATTICES

Our case study on the Twelf module system consists of two major parts: The encoding of abstract algebra and the encoding of lattices (both as algebraic structures in Sect. 3.2.1 and as orderings in Sect. 3.2.2). The two variant of the encodings of lattices can be proven to be equivalent by translating the former encoding to the latter one. We elaborate these translations by presenting brief examples in Sect. 3.2.3. We present both the underlying mathematical theory of the content being encoded and the respective Twelf encodings, and highlight the main features of the Twelf module system as we go along with the presentation of the encodings.

3.1 The Encoding of Algebraic Structures

3.1.1 Algebraic Structures with One Binary Operation

The most basic structure in algebra is a *magma* - a set endowed with a binary operation. Virtually all other structures found in algebra - be it groups, ring, fields, or vector spaces - are built on top of one or more magmas. Fig. 1 illustrates the hierarchy among the signatures encoding the algebraic structures with one binary operation.

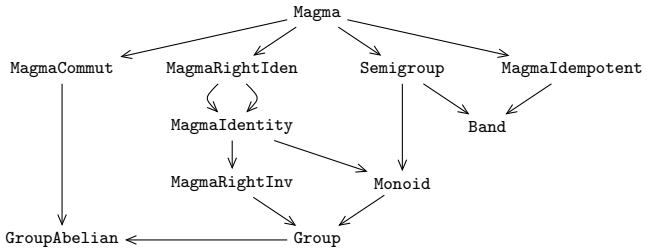

Figure 1: The Signature Graph for Algebraic Structures with One Binary Operation

```
%sig Magma = {
  %include FOL      %open i.
  * : i -> i -> i.  %infix *.
}.
```

In Twelf, we encode magmas in the signature `Magma` where we declare the binary operation $*$. The underlying set of all elements of the magma is precisely the type i of individuals declared the signature `FOL`, which is imported to `Magma` as an inclusion.

Given a magma M with the operation $*$, we may often wish to consider the magma dual to M, i.e. the magma with $*$ replaced by the operation $*'$, where $x *' y = y * x$. This can be easily done by introducing a view from `Magma` to `Magma`, which redefines the operation $*$ accordingly. The term `[x][y] y * x` is a function of the required type $i \to i \to i$, hence the instantiation is valid.

```
%view OppositeMagma : Magma -> Magma = {
  * := [x][y] y * x.
}.
```

```
%sig MagmaCommut = {
  %include FOL          %open ⊢ ∀ ==.
  %struct mag : Magma   %open *.
  commut : ⊢ ∀ [x] ∀ [y] ((x * y) == (y * x)).
}.
```

We can now extend magmas by requiring that the binary operation possesses certain properties. For instance, we define a *commutative magma* in the signature `MagmaCommut` by encapsulating an instance of `Magma` and adding a commutativity axiom.

An instance of `Magma` is imported into `MagmaCommut` via `%struct mag : Magma`, which imports all symbols and axioms declared in `Magma` - in our case just the symbol $*$ - and prefixes all symbol names with the name of the import. In this respect, the `%struct` and `%include` operators behave similarly; however, unlike `%include`, which only imports symbols and axioms, the `%struct` operator also allows us to instatiate imported symbols with existing ones. Furthermore, if two copies of the same signature are imported via a `%struct` they are treated as different; if they are included by means of an `%include` they are considered identical.

We define a *semigroup*, which is an associative magma, and an *idempotent magma* analogous to `MagmaCommut` by extending `Magma` with an axiom for associativity and idempotency, respectively.

```
%sig MagmaRightIden = {
  %include FOL %open i ⊢ ∀ ==.
  %struct mag : Magma %open *.
  e : i.
  iden : ⊢ ∀ [x] ((x * e) == x).
}.
```

Next, we define a magma with an identity element for $*$. To this extent, we first define a magma with a special element e, and add an axiom asserting that e is a right identity for the magma.

If a magma has a left identity element e, then in its dual the element e is a right identity. Thus, we can define a magma with both right and left identity by using duality as follows.

```
%sig MagmaIdentity = {
  %include FOL.
  %struct rid : MagmaRightIden %open e.
  %struct lid : MagmaRightIden = {
    %struct mag := OppositeMagma rid.mag.
                e := e.}.
}.
```

The symbols $*$ and e are imported from `MagmaRightIden` via the structure `rid`, thus we know that e is a right identity for $*$. To assert that e is also a left identity, we use a second instance of `MagmaRightIden`, which we get via the structure `lid`. We do not wish to import any new symbols

this way, but only axioms. For this reason, we introduce two assignments in `lid`. We want the identity element we get via `lid` to be precisely the element e coming from `rid`; therefore we have the (second) assignment `e := e` (the left-hand side is the name of the symbol imported by `lid` while the right-hand side is the element e that already exists in `MagmaIdentity`).

Consider `%struct mag := OppositeMagma rid.mag`, a structure assignment, which establishes that the operation $*$ imported by `lid` is the dual operation of $*$ that comes via `rid`. The assignment automatically instantiates all symbols in the left-hand side structure by the corresponding symbols in the right-hand side one. Now the structure `lid` gives us an axiom asserting that e is a right identity for the dual of the magma imported by `rid`. Via the instantiations, the original axiom `iden : ⊢ ∀ [x] ((x * e) == x)` of `MagmaRightIden` maps to `iden : ⊢ ∀ [x] ((([x] [y] y * x) x e) == x)`, which gets evaluated to `iden : ⊢ ∀ [x] ((e * x) == x)` as desired.

Now we are ready to define a *monoid*, which is a semigroup with an identity element.

```
%sig Monoid = {
  %include FOL.
  %struct sg : Semigroup.
  %struct miden : MagmaIdentity
                = {%struct rid.mag := sg.mag.}.
}.
```

Therefore, we define the signature `Monoid` by importing from `Semigroup` and `MagmaIdentity`, where we instantiate the underlying magma of `MagmaIdentity`, with the underlying magma of `Semigroup` in order to assert that the operation $*$ that comes from both signatures is the same. This is done by the structure assignment `%struct rid.mag := sg.mag` in `miden`. In a similar fashion we define a *band*, which is an idempotent semigroup.

```
%sig MagmaRightInv = {
  %include FOL %open i ⊢ ∀ ==.
  %struct id : MagmaIdentity %open * e.
  inv : i -> i.
  inverse : ⊢ ∀ [x] ((x * (inv x)) == e).
}.
```

Our next goal is to define a *group*, probably the most recognized algebraic structure with a single operation. A group is a monoid where every element has an inverse. Similarly as with identities, we first define a magma in the signature `MagmaRightInv`, where every element has a right inverse. For this we extend `MagmaIdentity`, since talking about inverses only make sense in a structure with (at least a one-sided) identity. We declare a new symbol `inv` for the inverse operation and add an axiom asserting that for each x, the inverse of x is its right inverse.

Now we could proceed to define a magma where every element has an inverse, much like we did with identities. However, it turns out this is not necessary - for a definition of a group we can likewise assume every element has a right inverse and from this it follows that each element necessarily has an inverse. This is easy to see and we leave it to the reader. Hence, we have the following formalization of groups.

```
%sig Group = {
  %include FOL.
  %struct mon  : Monoid %open * e.
  %struct minv : MagmaRightInv =
    {%struct id := mon.miden.} %open inv.
}.
```

As usual, we identify the two copies of the identity operation coming from the signatures `Monoid` and `MagmaRightInv`, by introducing the assignment `%struct id := mon.miden`.

Abelian groups are groups in which the operation $*$ is commutative, and we often write $+, 0, -$ instead of $*, e$ and the inverse operation. Therefore, we define the signature `GroupAbelian` by importing from `Group` and `MagmaCommut`. When importing from `MagmaCommut`, we instantiate the structure `mag` with the structure that inherits the underlying magma of the group imported into `GroupAbelian` in order to assert that the operation $*$ in that comes along `g` coincides with the one that comes along `mc`.

```
%sig GroupAbelian = {
  %include FOL.
  %struct g  : Group %open * e inv.
  %struct mc : MagmaCommut =
                    {%struct mag := g.mon.sg.mag.}.
  + = [x][y] x * y.  %infix +.
  0 = e.
  - = inv.
}.
```

3.1.2 Algebraic Structures with Two Binary Operations

Now we consider algebraic structures with two binary operations, usually denoted as $+$ and $*$. Fig. 2 illustrates the modular dependency between the signatures encoding the algebraic structures with two binary operations.

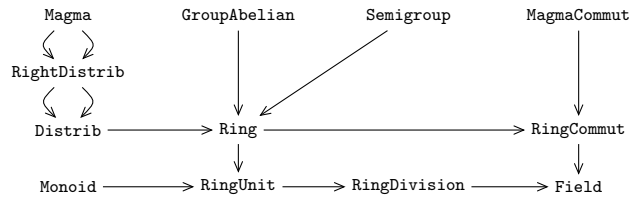

Figure 2: The Signature Graph for Algebraic Structures with Two Binary Operations

```
%sig RightDistrib = {
  %include FOL %open ⊢ ∀ ==.
  %struct add : Magma.
  %struct mul : Magma.
  + = [x][y] x add.* y. %infix +.
  * = [x][y] x mul.* y. %infix *.
  dist : ⊢ ∀ [x] (∀ [y] (∀ [z]
  (((x + y) * z) == ((x * z) + (y * z))))).
}.
```

First, we encode an algebraic structure where the right-distributive law holds, i.e. for all x, y, z, we have $(x+y)*z = x*z+y*z$. For that we need two binary operations, which we get via importing `Magma` twice (by the structures `mag1` and `mag2`), and we denote the binary operations we get via `mag1` and `mag2` as $+$ and $*$, respectively.

We can obtain a left-distributive algebraic structure from `RightDistrib` if we have a magma and its dual. Thus, we can give the following encoding of a distributive structure in the signature `Distrib`. The magmas imported by the structures `rdis` and `ldis` should coincide, since they define the same $+$ operation. The second magmas should be dual to each other since we want to invert the order of arguments in $*$.

```
%sig Distrib = {
  %include FOL.
  %struct rdis : RightDistrib.
  %struct ldis : RightDistrib =
          {%struct add := rdis.add.
```

```
          %struct mul := OppositeMagma rdis.mul.}.
}.

%sig Ring = {
  %include FOL.
  %struct ga  : GroupAbelian.
  %struct sg  : Semigroup.
  %struct dis : Distrib = {
    %struct rdis.add := ga.g.mon.sg.mag.
    %struct rdis.mul := sg.mag.}.
}.
```

A *ring* is a set S together with two operations $+$ and $*$ and an element 0 such that $(S, +, 0)$ is an Abelian group, $(S, *)$ is a semigroup, and $(S, +, *)$ is a distributive structure. We can easily encode this as follows.

We encode a *commutative ring*, where the operation $*$ is commutative, by extending `Ring` with the properties of `MagmaCommut`. Similarly, we encode a *ring with unity*, where we have an identity element for $*$, which is commonly denoted as 1, by extending `Ring` with `Monoid`. We encode these two algebraic structures in the signatures `RingCommut` and `RingUnit`, whose contents are omitted here.

A *division ring* is a ring with unity, where each non-zero element has an inverse. The natural way to encode this would be to say that a ring $(S, +, 0, -, *, 1)$ is a division ring if the set $S \setminus \{0\}$ with the operations $*$ and 1 forms a group. Since we use FOL as our meta-logic, we cannot talk about subsets of the set of all individuals (i.e. terms of type i) as FOL does not have subsorting. Hence, we need to add separate axioms for the existence of inverses, which are given by the terms `invLeft` and `invRight`.

```
%sig RingDivision = {
  %include FOL          %open i ⊢ ∀ ⇒ != ==.
  %struct ru : RingUnit %open 0 * 1.
  inv : i -> i.
  invLeft  : ⊢ ∀ [x] ((x != 0) ⇒
                      ((x * (inv x)) == 1)).
  invRight : ⊢ ∀ [x] ((x != 0) ⇒
                      (((inv x) * x) == 1)).
}.

%sig Field = {
  %include FOL.
  %struct rd : RingDivision.
  %struct mc : RingCommut = {
    %struct r := rd.ru.r.}.
}.
```

Finally, we define a *field* as a commutative division ring: When importing `RingCommut` we instantiate the structure `r` with the underlying ring of the imported division ring `rd`, to assert that the operations $+, 0, -, *$ imported via `rd` coincide with the ones imported via `mc`. These encodings can be found in [SR09].

3.2 The Encoding of Lattices

3.2.1 Lattices as Algebraic Structures

A *lattice* is an algebra $\langle L; \cap, \cup \rangle$, where L is a non-empty set, and $\cap$ and $\cup$ are binary operations on L. We will call these operations *meet* and *join*, respectively. Also, both operations have to satisfy idempotency, commutativity, associativity and the following absorbtion laws: $a \cap (a \cup b) = a$ and $a \cup (a \cap b) = a$.

We encode the theory of lattices modularly using the algebraic structures from Sect. 3.1 as illustrated in Fig. 3.

A *semilattice* is an algebra whose operation is idempotent, commutative and associative. A lattice is thus defined to contain two semilattices connected by the above-defined

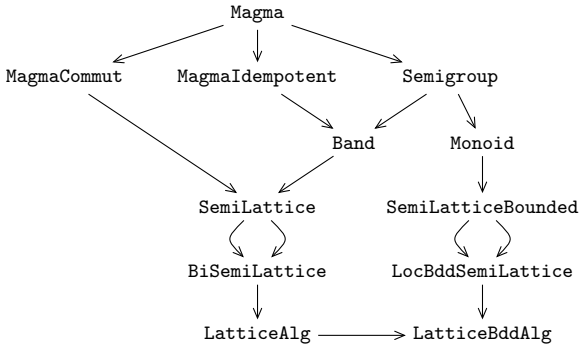

Figure 3: The Signature Graph for Lattices as Algebras

absorbtion laws. Given that the underlying operations of these semilattices are $\cap$ and $\cup$, they are therefore called *meet semilattice* and *join semilattice*, respectively.

```
%sig SemiLattice = {
  %include FOL.
  %struct mc : MagmaCommut.
  %struct bd : Band = {
    %struct sg.mag := mc.mag.}.
}.
```

We encode semilattices in the signature `SemiLattice` as a union of other algebraic structures, whose joint intrinsic axioms establish all of its defining properties. `SemiLattice` inherits commutativity from a commutative magma via the structure `mc`, associativity and idempotency from a band via the structure `bd`. We enforce with a structure assignment that `MagmaCommut` and `Band` share a common magma.

Extending the concept, we can introduce a *bisemilattice* as the pair consisting of a meet and a join semilattice, i.e an algebra with the operations $\cap$ and $\cup$, both of which satisfy the three above-mentioned properties. Thus, it follows naturally that a lattice can be considered as a bisemilattice that satisfies the absorption laws.

```
%sig BiSemiLattice = {
  %include FOL.
  %struct meet : SemiLattice.
  %struct join : SemiLattice.
  ∩ = [x][y] x meet.bd.sg.mag.* y.
                          %infix ∩.
  ∪ = [x][y] x join.bd.sg.mag.* y.
                          %infix ∪.
}.
```

The meet and join operation of `BiSemiLattice` are defined using two copies of the $*$ operation of `Magma`, imported along the structures `meet.bd.sg.mag` and `join.bd.sg.mag`, respectively.

Finally, we encode lattices in the signature `LatticeAlg` by importing a `BiSemiLattice` and asserting the absorption laws additionally.

```
%sig LatticeAlg = {
  %include FOL                          %open ⊢ ∧ ∀ ==.
  %struct bisemlat: BiSemiLattice %open ∩ ∪.
  absorbtion: ⊢ ∀ [x] ∀ [y] ((x ∩ (x ∪ y)) == x)
                           ∧ ((x ∪ (x ∩ y)) == x).
}.
```

We encode *bounded* semilattices and lattices analogous to the encodings of semilattices and lattices by extending them with greatest and least elements.

3.2.2 Lattices as Ordered Sets

Here we represent lattices from a set ordering perspective. The signature graph for the encoding of lattices as orderings is given in Fig. 4.

Let us introduce the notion of a *poset* $\langle L; \leq \rangle$, which is defined as a set equipped with a partial ordering $\leq$, i.e satisfying reflexivity, anti-symmetry and transitivity. A lattice is then a poset $\langle L; \leq \rangle$ such that, for every pair of elements $a, b \in L$, there exist $inf\{a, b\}$ and $sup\{a, b\}$. A meet (join) semilattice is a poset $\langle L; \leq_\cap \rangle$ ($\langle L; \leq_\cup \rangle$), such that, for all $a, b \in L$, there exists $inf\{a, b\}$ ($sup\{a, b\}$).

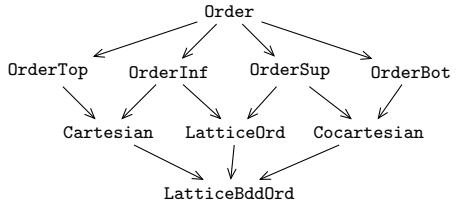

Figure 4: The Signature Graph for Lattices as Orderings

First, we encode orderings in the signature `Order` by introducing a binary operation $\leq$ and its partial ordering properties. $\leq$ takes two arguments of type i and returns a truth-value, which encodes whether the first element is smaller or equal to the second.

```
%sig Order = {
  %include FOL %open i o ⊢ ∧ ⇒ ∀ ==.
  ≤ : i -> i -> o.  %infix ≤.
  refl    : ⊢ ∀ [x] (x ≤ x).
  antisym : ⊢ ∀ [x] (∀ [y]
              ((x ≤ y) ∧ (y ≤ x) ⇒ (x == y))).
  trans   : ⊢ ∀ [x] (∀ [y] (∀ [z]
              ((x ≤ y) ∧ (y ≤ z) ⇒ (x ≤ z)))).
}.
```

We encode a meet semilattice in the signature `OrderInf` as an ordering with a binary infimum operation. First, we import from `Order` and add the infimum axiom, `ax_inf`. Then we introduce a binary operation `Inf` whose definition is not yet given. Next, we define a predicate `is_infimum_of` that takes three arguments, namely two elements x and y, and one of their lower bounds l, and checks that for every other lower bound z of x and y whether l is actually the greatest one, i.e., $z \leq l$. Finally, we can assert the axiom `ax_inf`, which encodes that for any elements x and y, the term `Inf x y` is the infimum of x and y.

```
%sig OrderInf = {
  %include FOL %open i o ⊢ ∧ ⇒ ∀.
  %struct ord : Order %open ≤.
  Inf : i -> i -> i.
  is_infimum_of : i -> i -> i -> o = [x][y][l]
     ((l ≤ x) ∧ (l ≤ y)) ∧
     (∀ ([z] ((z ≤ x) ∧ (z ≤ y) => (z ≤ l)))).
  ax_inf : ⊢ ∀ [x] ∀ [y] is_infimum_of x y (Inf x y).
}.
```

Analogously, the signature `OrderSup` encodes an ordering with a binary supremum operation.

```
%sig OrderSup = {
  %include FOL %open i o ⊢ ∧ ⇒ ∀.
  %struct ord : Order %open ≤.
  Sup: i -> i -> i.
  is_supremum_of: i -> i -> i -> o = [x][y][u]
    ((x ≤ u) ∧ (y ≤ u)) ∧
    (∀ ([z] ((x ≤ z ∧ y ≤ z) => u ≤ z))).
  ax_sup: ⊢ ∀ [x] ∀ [y] is_supremum_of x y (Sup x y).
}.
```

Notice that we can make the infimum and the supremum operation equivalent on the dual order. Therefore, we define the view `Opp` below, which interprets `Order` in itself where its binary operation $\leq$ is mapped to the term `[x][y] (y ≤ x)`, i.e., the dual of $\leq$. In other words, it reflects the fact that the ordering properties are maintained when the order of the arguments of $\leq$ is switched. Therefore, we have to prove that reflexivity, anti-symmetry and transitivity hold for the dual of $\leq$. More specifically, we need to map the terms `refl`, `antisym` and `trans` in `Order` to respective proof terms.

```
%view Opp : Order -> Order = {
  leq     := [x][y] (y ≤ x).
  refl    := refl.
  antisym := FOL..forallI [x] FOL..forallI [y]
             FOL..impI [p] (FOL..impE (FOL..forallE y
             (FOL..forallE x antisym))(FOL..andI
             (FOL..andEr p) (FOL..andEl p))).
  trans   := FOL..forallI [x] FOL..forallI [y]
             FOL..forallI [p] FOL..impI [u] (FOL..impE
             (FOL..forallE x (FOL..forallE y
             (FOL..forallE p trans))) (FOL..andI
             (FOL..andEr u)(FOL..andEl u))).
}.
```

First, `refl` is trivially mapped to itself since an opposite ordering has no effect when the two arguments of $\leq$ are equal. Second, `antisym` on this inverse order is obtained by applying modus ponens to the corresponding property in `Order` and to a new clause resulting from the latter by inverse elimination of the hypothesis conjuncts. The proof term for transitivity follows similarly. Currently, the implementation of the Twelf module system does not allow one to omit qualified names within views, which makes even simple proofs look rather ghastly.

We define lattices as orderings in the signature `LatticeOrd` by extending the notions of meet and join semilattices to that of two orderings with the infimum and supremum operator, respectively. Thus we import from `OrderInf` and `OrderSup` and add the condition that the corresponding orderings are actually equivalent, which we elaborate in Sect. 3.2.3.

```
%sig LatticeOrd = {
  %include FOL %open ⊢ ∀ == <=>.
  %struct inf : OrderInf.
  %struct sup : OrderSup.
  ax_leq : ⊢ ∀ [x] ∀ [y]
             ((x inf.ord.leq y) <=> (x sup.ord.leq y)).
}.
```

The remaining signatures in Fig. 4 are `OrderTop` and `OrderBot`, which encode orderings with greatest (top) and smallest (bottom) element, respectively; `Cartesian`, which encodes an ordering with a top element and an infimum operation; `Cocartesian`, which encodes an ordering with a bottom element and a supremum operation; and finally, `LatticeBddOrd`, which encodes an order-based bounded lattice. We omit their definitions here.

3.2.3 Equivalence of the Two Definitions

We illustrate how the two approaches in the definition of lattices are equivalent by translating one definition to the other one by means of views. A view $v : S \to T$ from a signature S to a signature T maps every symbol s in S to an expression t in T, where the typing relation is preserved. The homomorphic extension of v maps all S-expressions (i.e., terms, types and kinds) to T-expressions. This means that a term $s : A$ in S is translated to a term $v(s) : v(A)$ in T.

Fig. 5 presents some of the views we have established between the signatures corresponding to the order-based and algebraic lattices (the full encoding can be found in [DR09]). The views are indicated with double arrows and

local names of the structures are given. These views are `Opp` from Sect. 3.2.2, `OppSup` and `OppInf`, which map an ordering with an supremum operation to an ordering with an infimum operation and vice-versa; `OrdSL` and `SLOrd`, which map an ordering with an infimum operation to an algebraic semilattice and vice-versa; and `LatOrdAlg` and `LatAlgOrd` which map an order-based lattice to its algebraic counterpart and vice-versa.

We briefly explain the views `OrdSL` and `LatOrdAlg`. To understand the views, consider that the following are equivalent in `LatticeAlg` (see, e.g., [DP02]): $a \cap b = a$ and $a \cup b = b$. The equivalence between `LatticeOrd` and `LatticeAlg` interprets $a \leq b$ as this condition, and $\cap$ and $\cup$ as `Inf` and `Sup`.

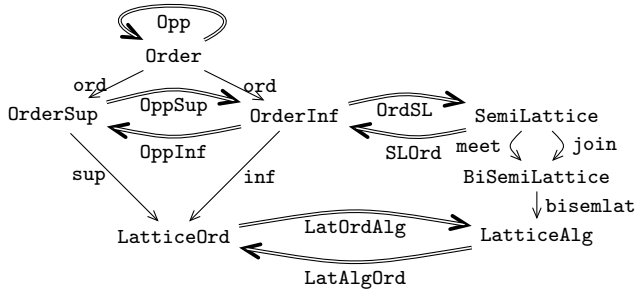

Figure 5: A Simplified Signature Graph for Views between Order-Based and Algebraic Lattices

To conclude the view analysis, we present the view establishing the top-level mapping from an order-based lattice to an algebraic lattice. We initially had the version of the signature `LatticeOrd` on the right, where we shared the structures `inf.ord` and `sup.ord` by making them definitionally equal.

```
%sig LatticeOrd = {
  %include FOL.
  %struct inf : OrderInf.
  %struct sup : OrderSup = {%struct ord := inf.ord.}.
}.
```

However, this proved to be difficult later when establishing the view `LatOrdAlg` shown below. Mathematically, the view should map the following: (1) `inf.ord` to $x \cap y = x$, and (2) `sup.ord` to $y \cup x = y$, which can be shown to be equivalent in `LatticeAlg`. However, since these are not definitionally equal, the view is rejected by Twelf. Therefore, we used the version of `LatticeOrd` as in Sect. 3.2.2, i.e., only asserting that the two orderings are equivalent – recall `ax_leq` in `LatticeOrd` – instead of making them definitionally equal.

```
%view LatOrdAlg : LatticeOrd -> LatticeAlg = {
 %struct inf := OrdSL bisemlat.meet.
 %struct sup := OppSup OrdSL bisemlat.join.
 ax_leq := ...
}.
```

`LatOrdAlg` interprets the constants of `LatticeOrd` in the signature `LatticeAlg` as follows. (Fig. 5 should help the reader to follow the morphism compositions we will explain below.) The structure `inf`, which imports `OrderInf` into `LatticeOrd` needs to be mapped to a morphism, which interprets `OrderInf` in `LatticeAlg`. We, therefore, map `inf` to the composition of the view `OrdSL`, which interprets `OrderInf` in `SemiLattice`, and the structure `bisemlat.meet`, which imports the content of `SemiLattice` into `LatticeAlg`. Thus, the interpretation of the structure `sup` is not only analogous to that of `inf`, but even re-uses the interpretation of `inf` by means of the view `OppSup`. Finally, we map `ax_leq` to a proof term that proves that the conditions (1) and (2) above are equivalent.

4. CONCLUSION

We have demonstrated that informal mathematical content can be given a precise formal representation in the Twelf module system. The modular structure in the encoding is particularly natural with respect to how informal mathematical reasoning is usually performed. Moreover, signature morphisms allow for the re-use of individual concepts encoded in separate signatures when defining new ones.

Structures are especially useful to handle multiple inheritance of the same signature. However, they often create lengthly prefixes of symbols (e.g., `ga.g.mon.sg.mag` in the signature `Ring`). While the opening of names is one way to avoid prefixes, further improvements can be done for structure names: The number of morphisms of a certain type is often quite limited, and all these morphisms can be found by a path search in the signature graph. We propose to devise an inference mechanism in Twelf that infers the possible targets of an unknown structure reference and chooses the best matching one according to some name resolution algorithm. Such a mechanism would allow the user to write down simplified structure names.

One significant problem we encountered in our case study is due to the fact that consistency conditions in morphisms are checked against Twelf's definitional equality. That precludes the justification of views by appealing to provable equality (or in our case: equivalence) of expressions. The latter would clearly be desirable but very difficult to realize in a logical framework like LF where any equality relation going beyond the built-in definitional equality is represented by a user-declared symbol that is not treated differently from any other symbol.

5. REFERENCES

[AHMS99] S. Autexier, D. Hutter, H. Mantel, and A. Schairer. Towards an Evolutionary Formal Software-Development Using CASL. In D. Bert, C. Choppy, and P. Mosses, editors, *WADT*, volume 1827 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 1999.

[Chr03] Jacek Chrzszcz. Implementing modules in the Coq system, 2003.

[Coq] The Coq Proof Assistant. `http://coq.inria.fr/`.

[DP02] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge Univ. Press, 2002.

[DR09] Stefania Dumbrava and Florian Rabe. The Twelf Encoding of Lattices and Orderings, 2009. `https://svn.kwarc.info/repos/twelf/order/`.

[Far00] William Farmer. An infrastructure for intertheory reasoning. In David McAllester, editor, *Automated Deduction – CADE-17*, number 1831 in LNAI, pages 115–131. Springer Verlag, 2000.

[FGT92] William Farmer, Josuah Guttman, and Xavier Thayer. Little theories. In D. Kapur, editor, *Proceedings of the 11th Conference on Automated Deduction*, volume 607 of *LNCS*,

pages 467–581, Saratoga Springs, NY, USA, 1992. Springer Verlag.

[HHP93] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[HR09] F. Horozal and F. Rabe. Representing Model Theory in a Type-Theorical Logical Framework. Submitted, 2009.

[HST94] R. Harper, D. Sannella, and A. Tarlecki. Structured presentations and logic representations. *Annals of Pure and Applied Logic*, 67:113–160, 1994.

[KW98] Florian Kammüller and Markus Wenzel. Locales: A sectioning concept for Isabelle. In *Theorem Proving in Higher Order Logics (TPHOLs Š99), LNCS 1690*, pages 149–165. Springer, 1998.

[Pau94] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[PS99] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. *Lecture Notes in Computer Science*, 1632:202–206, 1999.

[RS09] F. Rabe and C. Schürmann. A Practical Module System for LF. Submitted, see `http://kwarc.info/frabe/Research/lf.pdf`, 2009.

[SR09] Kristina Sojakova and Florian Rabe. The Twelf Encoding of Basic Algebraic Structures, 2009. `https://svn.kwarc.info/repos/twelf/algebra/`.

[Wen99] Markus Wenzel. Isar - a generic interpretative approach to readable formal proof documents. In *TPHOLs*, pages 167–184, 1999.