Diagram Combinators in MMT

Florian Rabe[1,2] and Yasmine Sharoda[3] *

[1] Computer Science, FAU Erlangen-Nürnberg, Germany
[2] LRI, Université Paris Sud, France
florian.rabe@fau.de
[3] Computing and Software, McMaster University, Canada
sharodym@mcmaster.ca

Abstract. Formal libraries, especially large ones, usually employ modularity to build and maintain large theories efficiently. Although the techniques used to achieve modularity vary between systems, most of them can be understood as operations in the category of theories and theory morphisms. This yields a representation of libraries as diagrams in this category, with all theory-forming operations extending the diagram.
However, as libraries grow even bigger, it is starting to become important to build these diagrams modularly as well, i.e., we need languages and tools that support computing entire diagrams at once. A simple example would be to systematically apply the same operation to all theories in a diagram and return both the new diagram and the morphisms that relate the old one to the new one.
In this work, we introduce such diagram combinators as an extension to the MMT language and tool. We provide many important combinators, and our extension allows library developers to define arbitrary new ones. We evaluate our framework by building a library of algebraic theories in an extremely compact way.

1 Introduction and Related Work

Motivation Throughout the development of formal mathematical languages, we can see how when the size of formalizations reached new scales, novel concepts and language features became necessary. In particular, as libraries grow bigger and require extensive maintenance, the elimination of redundancy becomes even more critical.

Originally, only basic declarations such as operators, and axioms were used. Over time *theories* were made explicit in order to name groups of basic declarations. Then calculi were introduced to build larger theories from smaller ones. Using the language of category theory proved particularly useful: *theory morphisms* could be used to relate theories, and operators like *colimits* could be used to compose them. Structured theories [1, 14] introduced, among other operations, unions and translations of theories. Systems like OBJ [7] and most comprehensively CASL [4] further developed these ideas.

* The authors were supported by DFG grant RA-18723-1 OAF and EU grant Horizon 2020 ERI 676541 OpenDreamKit.

Today many major systems support some language features for theory formation such as Isabelle's locale expressions [8] or Coq's module system. Specware [15] is notable for introducing colimits on arbitrary diagrams early on. For maximizing reuse, it is advisable to apply the little theories approach [6] that formalizes the mathematical principle of making every statement in the smallest theory that supports it — even if that means retroactively splitting theories up into smaller ones. This leads to an organization of formal libraries as large diagrams of theories and theory morphisms, in which results can be moved between theories along morphisms.

We believe that one next step in this development will be *diagram operations* that operate not just on individual theories but on entire diagrams. They will allow building structured diagrams by reusing diagram-level structure systematically. For example, one might build a diagram with > 100 theories for the algebraic hierarchy (containing the theories of groups, rings, etc.) and then, in a single operation, systematically duplicate it and add the axiom of finiteness to each theory.

Contribution We develop a formal language for named diagrams of theories and operations on such diagrams. To our knowledge, this is the first time such a language has been developed systematically.

We work in the context of the MMT system [13], which is practical for us because it already provides a logic-independent language for structured theories and morphisms. Moreover, because its kernel is designed to be extensible [12], even disruptively new concepts like ours can be added with relative ease. However, our ideas are not tied to MMT and can be transferred easily to any formalism for diagram of theories.

Our solution is extensible in the sense that users can add new diagram operators, beside the initial suite that we provide.

Related Work Our work is inspired by and generalizes two recent results. Firstly, the DOL language [5] and its implementation in Hets [9] uses named diagrams together with operations for building larger diagrams by merging or removing elements from smaller ones. It operates on names only, i.e., diagrams are formed by listing the names of subdiagrams, theories, and morphisms that should be included or excluded. DOL is developed in the context of ontology languages, but the ideas are logic-independent.

Secondly, the MathScheme project revisited theory combinators in [2]. This work recognized that many theory-formation operators can actually be better understood as operators returning diagrams, e.g., the pushout operator return not only a theory but also three morphisms. It does not use diagrams as input yet, and we believe our generalization brings out its ideas much more clearly.

Overview The structure of the paper is very simple. In Sect. 2, we recap MMT theories and morphisms. Moreover, we introduce MMT diagrams, a new definition that adapts the standard notion of diagrams from category theory for our purposes. In Sect. 3, we specify our initial library of diagram operators. Some of

these operators are closely related to existing ones, while others are new. Finally in Sect. 4, we describe how diagrams and diagram operators are implemented in MMT.

2 Diagrams

2.1 Theories and Morphisms

The main building blocks of diagrams are theories and morphisms, and we briefly introduce MMT's definitions for them. We refer to [11, 12] for details.

Definition 1 (Theory). *A **theory** is a pair of an optional theory M — the **meta-theory** — and a list of declarations.*

*A **declaration** is of the form $c\,[:A]\,[=t]\,[\#N]$ where*
- *c is a fresh identifier — the **name**,*
- *A is an optional expression — the **type**,*
- *t is an optional expression — the **definiens**,*
- *N is an optional **notation** (which we only need in this paper to make examples legible).*

*Every theory T induces a set of T-**expressions**. The details of how expressions are formed are not essential for our purposes except that they are syntax trees formed from (among other things) references to constants declared in T or its meta-theory M.*

The role of the meta-theory is to make MMT logic-independent. Individual logics L are formalized as theories as well, which declare the logic's primitive operators and typing rules and are then used as the meta-theory of L-theories.

Example 1. The following is an MMT theory declaration that introduces the theory `Magma` with typed first-order logic as the meta-theory:

```
theory Magma : SFOL =
   U : type
   op : U → U → U # 1 ∘ 2
```

The theory `Semigroup` arises by extending `Magma` with the axiom

```
   associativity : ∀ [x,y,z] (x ∘ y) ∘ z ≐ x ∘ (y ∘ z)
```

Definition 2 (Morphism). *Given two theories S and T a **morphism** $S \to T$ consists of an optional meta-morphism and a list of assignments.*

*An **assignment** is of the form $c := t$ for an S-constant c and a T-expressions t, and there must be exactly one assignment for every S-constant.*

*Every morphism $m : S \to T$ induces the **homomorphic extension** $m(-)$ that maps S-expressions to T-expressions. The details depend on the (omitted) definition of expressions, and we only say that $m(E)$ arises by replacing every reference to an S-constant c with the expression t provided by the corresponding assignment in m.*

The role of the meta-morphism is to translate the meta-theory of S (if any) to the meta-theory of T. For our purposes, it is sufficient to assume that we work with a fixed meta-theory and all meta-morphisms are the identity.

We omit the typing conditions on assignments and only mention that t must be such that the homomorphic extension preserves all judgments, e.g., if $\vdash_S t : A$ then $\vdash_T m(t) : m(A)$. This preservation theorem is critical to move definitions and theorems between theories along morphisms in a way that preserves theoremhood.

Example 2. The inclusion morphism from `Magma` to `Semigroup` can be spelled out explicit as

```
Magma2Semigroup : Magma → Semigroup =id_SFOL
    U := U
    op := op
```

Let `AdditiveMagma` be the variant of `Magma` with the operator `op` renamed to `+`. Then the following defines the morphism that performs the renaming:

```
Additive : Magma → AdditiveMagma =id_SFOL
    U := U
    op := +
```

Both morphisms are trivial because they are just renamings. The following morphism, whips flips the arguments of the magma operation is more complex:

```
Flip : Magma → Magma =id_SFOL
    U := U
    op := λ x,y.y ∘ x
```

2.2 Diagrams

Using the straightforward definitions of identity and composition, MMT theories and morphisms form a category `THY`. Category theory defines *diagrams* over `THY` as pairs of a (usually finite) category G and a functor $D : G \to \texttt{THY}$. Here G only defines the graph structure of the diagram, and the actual G-objects are only relevant as distinct labels for the nodes and edges of the diagram. D assigns to each node or edge a theory resp. morphism.

However, for an implementation in a formal system, this abstract definition is not optimal, and we adapt it slightly. Moreover, we allow each diagram to have a *distinguished* node, and we allow morphisms to be *implicit*. These are novel variations of how diagrams are usually defined that will be very helpful later on:

Definition 3 (Diagrams). *A **diagram** D consists of*

- *a list of **nodes**, each written as* $\mathrm{Node}(l, D(l))$ *where*
  - *l is an identifier — the **label***
  - *$D(l)$ is a theory*
- *a list of **edges**, each written as* $\mathrm{Edge}(l, d \xrightarrow{i} c, D(l))$ *where*

  - *l is an identifier — the **label***
  - *d and c are labels of nodes — the **domain** and **codomain***
  - *$D(l)$ is a morphism $D(d) \to D(c)$*
  - *i is a boolean flag that may or may not be present — indicating whether the edge is **implicit***
- *an optional label $D^{\texttt{dist}}$ of a node — the **distinguished** node*

The labels of nodes and edges must be unique, and we write (as already used above) $D(l)$ for the theory/morphism with label l.

Every node may be the codomain of at most one implicit edge, and we write $f \xrightarrow{\vec{e}} t$ if e is the list of implicit edges forming the path from node f to node t. Here $\vec{e}$ is uniquely determined if it exists.

Remark 1 (Distinguished Nodes). In practice, diagram operations are often used to define a specific theory. The pushout, as we show in example 3, can be seen as a map between diagrams. But often we want to identify the resulting diagram with the new theory it contains. This is the purpose of using a distinguished node: if the pushout operator marks the new node as distinguished, we can later refer to the diagram as a whole when we mean that node.

Remark 2 (Implicit Edges). Implicit MMT morphisms were introduced in [10] as a generalization of inclusion morphisms. The main idea is that the collection of all implicit morphisms forms a commutative sub-diagram of the (not necessarily commutative) diagram of all theories and morphisms named in a user's development. Identity and inclusions morphisms are always implicit, and so is the composition of implicit morphisms. This has the effect that for any two theories S and T there may be at most one implicit morphism $S \to T$, which MMT can infer automatically. MMT uses for a number of implicit coercions that simplify concrete syntax, e.g., to allow S-expressions in T-contexts.

When implementing diagram operators, it will be particularly useful to conveniently refer to the implicit morphisms into the distinguished node.

In general, the same theory/morphism may occur at multiple nodes/edges of the same diagram with different labels. However, very often we have the following situation: a node/edge refers simply to a previously named theory/morphism, which only occurs once in the diagram. If a theory/morphism has not been previously named or occurs multiple times, we have to generate labels for them.

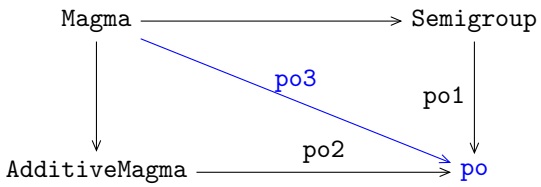
Example 3 (Diagram). The morphism `Magma2Semigroup` from Ex. 2 can be seen as a diagram D_1 with

- Node($\texttt{Magma}, (\texttt{SFOL}, U : \ldots, \texttt{op} : \ldots)$)
- Node($\texttt{Semigroup}, \texttt{SFOL}, U : \ldots, \texttt{op} : \ldots, \texttt{associative} : \ldots$)
- Edge($\texttt{Magma2Semigroup}, \texttt{Magma} \xrightarrow{i} \texttt{Semigroup}, id_{\texttt{SFOL}}, U := U, op := op$)

where we have omitted the types for brevity. We made this edge implicit because we want it to be inferred whenever a morphism $\texttt{Magma} \to \texttt{Semigroup}$ is needed so that we do not have to write `Magma2Semigroup` explicitly.

Let D_2 be the corresponding diagram for `AdditiveMagma`. We can now already indicate the usefulness of diagram operators: if we had a diagram operator

to glue D_1 and D_2 together into a span, and another diagram operator to build the pushout of a span, we could simply define `AdditiveSemigroup` as the resulting pushout diagram:



Here, we draw only labels for each node/edge. The labels `po`, `po1`, `po2`, and `po3` refer to nodes/edges added by the pushout operator.

Of course, we want `AdditiveSemigroup` to be the theory at node `po`, not the entire diagram. Therefore, the pushout operator should return a diagram in which `po` is the distinguished node. That way, if we use `AdditiveSemigroup` as a theory, the system can infer which theory is meant. Similarly, we mark the morphism `po3` as the implicit edge of the morphism, so whenever we use `AdditiveSemigroup` as a morphism, the system infers `po3`. Above and in the sequel, we follow the convention of drawing distinguished nodes and implicit arrows in blue.

3 Operations on Diagrams

There is a wide range of possible operations on diagrams. Our realization in MMT is extensible: as we describe in Sect. 4, we built a general framework in which users can implement new operators. In this section, we specify the individual operators that we are implementing within this framework. In Sect. 3.1, we introduce operators based on formation combinators like extend or pushout. In Sect. 3.2, we introduce basic operators for building diagrams by simply aggregating previously defined theories and morphisms. This includes operators for set-theoretic operations like unions. In Sect. 3.3, we combine the previous operators to introduce many theories at once.

All of the above operators can be defined for an arbitrary meta-theory. In Sect. 3.4, we give an example of an advanced, meta-theory-specific operator. To simplify the presentation, from now on, we assume that unless mentioned otherwise all theories have the same fixed meta-theory so that we can omit it from the notation.

3.1 Theory Formation Operators

Extension Consider the definition of `Semigroup` in example 1. We can define `Semigroup` more easily using a theory operator for extension:

$$\texttt{Semigroup} := \texttt{Magma EXTEND}\,\{\,\texttt{associativity}\colon \cdots\}$$

The central idea of [2] is that the combinator should create not just the theory `Semigroup` but also the morphism `Magma2Semigroup`. While `Magma2Semigroup` is trivial, more general combinators (in particular colimits) would introduce more and more complex new morphisms. Notably, these morphisms are often computed anyway when evaluating a combinator; thus, it would be wasteful to throw them away after building the theory.

We can capture this elegantly by defining extension as a diagram operator: We define $D\,\texttt{EXTEND}\,\{\Sigma\}$, where D is a diagram and Σ is a list of declarations, as the diagram consisting of the following components:

- all nodes and arrows in D.
- a new node labeled `pres` with all declarations in $D(D^{\texttt{dist}})$ and Σ.
- a new edge $\text{Edge}(\texttt{extend}, D^{\texttt{dist}} \xrightarrow{i} \texttt{pres}, id)$ where id maps every constant to itself,
- `pres` is the distinguished node.

Example 4. We build some extensions of `Magma` that we will use later on. We also recover `Magma` as an extension of the theory `Carrier`.

```
Carrier := Empty EXTEND {U: type}
Magma := Carrier EXTEND
         {op : U → U → U # 1 ∘ 2}
Semigroup := Magma EXTEND
 {associative_op : ∀ [x,y,z] (x ∘ y) ∘ z ≐ x ∘ (y ∘ z)}
Commutative := Magma EXTEND
 {commutative_op : ∀ [x,y] x ∘ y ≐ y ∘ x}
Idempotent := Magma EXTEND
 {idempotent_op : ∀ [x] x ∘ x ≐ x }
```

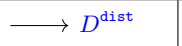
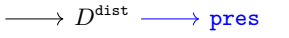
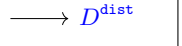
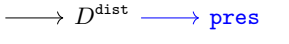
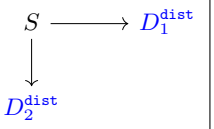
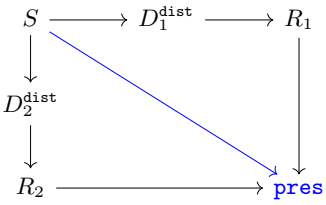
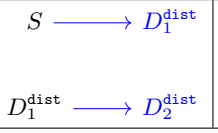
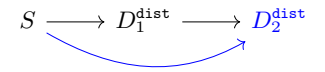
In the sequel, we give a few more examples how theory-forming operators can be captured elegantly as diagram operators. Our presentation and notation roughly follows [2] and Table 1 gives an overview.

Renaming We define $D\,\texttt{RENAME}\,\{c \rightsquigarrow c'\}$, where D is a diagram and c and c' are names, as the diagram consisting of the following components:

- all nodes and arrows in D,
- a new node `pres` which contains all declarations in $D(D^{\texttt{dist}})$ with every occurrence of c replaced by the expression c',
- a new edge $\text{Edge}(\texttt{rename}, D^{\texttt{dist}} \xrightarrow{i} \texttt{pres}, m)$ where m maps $c := c'$ and other constants to themselves.

Example 5. We obtain additive and multiplicative variants of `Magma` as follows:

```
AdditiveMagma := Magma RENAME {op ⇝ +}
MultiplicativeMagma := Magma RENAME {op ⇝ *}
```

Theory Expression	Input Diagram(s)	Output Diagram
`D extends Σ`		
`D rename r`		
`combine D₁ r₁ D₂ r₂` `mixin D₁ r₁ D₂ r₂`		
`D₁ ; D₂`		

In general, input diagrams may contain arbitrary other nodes/edges, which are copied over to the output diagram. Distinguished nodes and implicit edges are shown in blue.

Table 1. Theory-forming diagram operators

Combine Combine is a pushout operator for the special case where both of the inputs are embedding morphisms. Intuitively, an embedding is a composition of extensions and renames:

Definition 4 (Embedding). *An **embedding** is a morphism in which each assignment is of the form $c := c'$ for a definition-less constant c' and in which there are no two assignment with the same constant on the right-hand side.*

We say that a diagram D has an embedding $e : S \to T$ if $S \xrightarrow{\vec{e}} T$ such that each e_i is an extension or a renaming and their composition is an embedding.

The notation for combine is $\texttt{COMBINE}\, D_1\, r_1\, D_2\, r_2$ where D_i are diagrams and r_i are lists of renamings like above. It is defined as follows:

- all nodes and edges in D_1 and D_2
- a new node `pres` which contains the following theory: Let $S \xrightarrow{\vec{e_i}} D_i^{\texttt{dist}}$ be the shortest embeddings in D_i such that the distinguished node of each D_i embeds the same node S. Let R_i be the theory that arises by applying all renamings in r_i to $D_i^{\texttt{dist}}$. `pres` is the canonical pushout (if that exists) of the two morphisms that arise by composing $\vec{e_i}$ and the renaming morphisms $D_i^{\texttt{dist}} \to R_i$. While the pushout always exists, the *canonical* pushout exists only if there are no name clashes, i.e., if every name of S is renamed in neither R_i or in the same way in both R_i. In that case, the pushout can reuse the names of S, R_1, and R_2.
- three new edges, `diag`, $\texttt{extend}_1$ and $\texttt{extend}_2$ with codomain `pres` and domains S, $D_1^{\texttt{dist}}$, resp. $D_2^{\texttt{dist}}$.

– the new node is the distinguished node, and the edge `diag` is implicit.

Example 6. Continuing our running example, we define

```
Band := COMBINE Semigroup {} Idempotent {}
Semilattice := COMBINE Band {} Commutative {}
```

The two diagrams `Semigroup` and `Idempotent` have implicit edges that are extensions of `Magma`, which becomes the common source node S. As a result of the pushout, the arrow $\texttt{diag} : \texttt{Magma} \to \texttt{Band}$ is created and marked as implicit. Therefore, when defining `Semilattice` in the next step, `Magma` is again found as the common source of implicit edges into `Band` and `Commutative`.

In the case of `Band` and `Semilattice`, no name conflicts exist to begin with, so no renamings are needed. To show why renames are useful, consider the following:

```
AdditiveCommutative :=
   COMBINE Commutative {op ⇝ +} AdditiveMagma {}
```

Here `Magma` is again the common source, but the name *op* of `Magma` is renamed in two different way along the implicit edges into `Commutative` and `AdditiveMagma`. This would preclude the existence of the canonical pushout because there is no canonical choice of which name to use. Therefore, an explicit rename is needed to remove the name conflict, and the name + is used in the pushout.

Mixin Mixin is a special case of pushout related to combine. Whereas combine is symmetric, mixin takes one arbitrary morphism, one embedding, and two rename functions as arguments. Consider a morphism $m : S \to T$ and a diagram D with an embedding $e : S \to D^{\texttt{dist}}$. We write $\texttt{MIXIN}\, m\ r_1\, D\, r_2$ for the diagram consisting of:

– all nodes and edges from D,
– m and T (if not already part of D),
– a new node `pres` resulting from the pushout of m composed with $T \to R_1$ and e composed with $D^{\texttt{dist}} \to R_2$. Similar to combine, rename functions are used to avoid name clashes.
– 3 edges similar to combine.

Example 7. Consider a theory `LeftNeutral` that extends `Magma` with a left-neutral element. The theory `RightNeutral` can be obtained by applying `mixin` to the view `Flip`, from example 2, with the diagram `LeftNeutral`, as follows

```
RightNeutral := MIXIN Flip {} LeftNeutral {}
```

Composition We write $D_1; D_2$ for the composition of two morphisms. Technically, our definition is more general because it allows arbitrary diagrams D_i as the arguments. The precise definition is that $D_1; D_2$ is defined if each D_i has an implicit edge e_i into $D_i^{\texttt{dist}}$ such that the source S of e_2 is $D_1^{\texttt{dist}}$. In that case, $D_1; D_2$ contains the union of D_1 and D_2 with the distinguished node $(D_1; D_2)^{\texttt{dist}} = D_2^{\texttt{dist}}$ and a new implicit edge for the composition of e_1 and e_2.

Example 8. On our way to defining rings, we want to combine `AdditiveMagma` and `MultiplicativeMagma`. We cannot do that directly because the closest common source theory of the implicit edges in those two diagrams is still `Magma`. Thus, the pushout would not duplicate the magma operation. Instead we can use composition:

```
BiMagma :=
 COMBINE (Magma ; AdditiveMagma) {}
         (Magma ; MultiplicativeMagma) {}
```

Because the composition diagrams have only a single implicit edge `Carrier` $\to$ `AdditiveMagma` resp. `Carrier` $\to$ `MultiplicativeMagma`, now `Carrier` is the closest common source so that we obtain the desired pushout over `Carrier`. The distinguished node of `BiMagma` now has two binary operations $+$ and $*$.

3.2 General Diagram Formation

Diagrams are essentially sets (of theories and morphisms). In particular, if all diagram components have already been defined (and thus named) previously, we can simply form diagrams by listing their names.

Diagram from named elements Let $\vec{N}$ be a list of names of existing theories, morphisms, and diagrams. We write $\mathbf{diag}(N_1, \dots, N_r)$ for the diagram consisting of

1. the listed theories and morphisms,
2. for any listed diagram, the distinguished node and all implicit edges into it
3. the domain and codomain of every morphism (if not anyway among the above)

We reuse the existing names as labels, i.e., each node/edge has label N_i.

Set-Theoretic Operations We write $D \cup D'$ for the union of the diagrams D and D'. Here nodes/edges in D and D' are identified iff they agree on all components including their label. We define $D \cap D'$ accordingly. Note that $D \cup D'$ and $D \cap D'$ have no distinguished nodes.

Moreover, if the same label is used for different theories in the two diagrams D and D', we use qualified names to make sure the labels in the diagram $D \cup D'$ are unique.

Given a list of labels l_i of D, we write $D \setminus \vec{l}$ for the diagram that arises from D by removing

- all nodes/edges with label l_i
- all edges with domain or codomain l_i

The distinguished node of $D \setminus \vec{l}$ is the one of D if still present, and none otherwise.

Example 9. We create a diagram of the extensions of `Magma` defined in example 4.

```
MagmaExtensions := diag(Commutative, Semigroup, Idempotent)
```

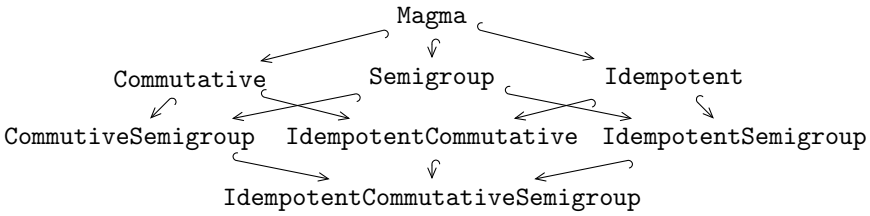
This diagram contains `Magma` and 3 implicit extension edges out of it.

3.3 Batch-Formation of Theories

Combining the operators from Sect. 3.1 and 3.2, we can introduce novel, very powerful operators that create many theories at once. We give two examples: batch combine and batch mixin.

Batch Combine Consider a diagram D consisting of a set of extension arrows that all have the same source. We define **BCOMBINE** D as the diagram that applies the combine operation exhaustively to every pair of extensions.

Example 10. Continuing our running example, **BCOMBINE** `MagmaExtensions` adds the other 4 combinations of associativity, commutativity, idempotence that are not yet part of `MagmaExtensions`: The resulting diagram is



It is difficult to make sure that **BCOMBINE** generates nodes with the intended names. `Commutative`, `Semigroup`, and `Idempotent` (defined by the `extend` operator) all have a distinguished node with the same label `pres`. When applying the union operation to define `MagmaExtensions`, the resulting diagram has no distinguished node, but each node is labeled with qualified names like `Commutative_pres` etc. When performing the **BCOMBINE** operation, these labels can be used to generate reasonable names like `CommutativeSemigroup`. It is still tricky to define convenient syntax that allows the user to communicate that the name `IdempotentCommutativeSemigroup` should be preferred over `IdempotentSemigroupCommutative`, let alone that the name `Semilattice` should be used instead. We get back to that in Sect. 5.

Batch Mixin Mixing in the morphism $m : S \to T$ is (barring some subtleties, see e.g. [3]) a functor from extensions of S to extensions T. Using the universal property of the pushout, any morphism $x : X \to Y$ between two S-extensions induces a universal morphism $m(x) : m(X) \to m(Y)$. Here we have written $m(X)$ for the theory introduced by the pushout. Consequently, we can apply mixin to an entire diagram at once.

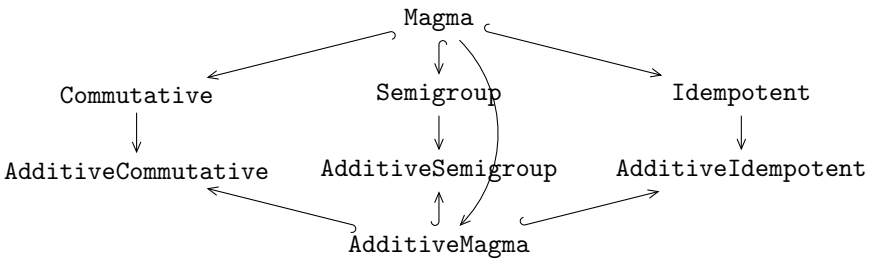
Let D be a diagram where every node has an embedding of a fixed node S. We write **BMIXIN** $m\, D$ for the diagram that extends D with

- for every $\mathrm{Node}(l, X)$ with $S \xrightarrow{\vec{e}} l$,
  - a node $\mathrm{Node}(\texttt{mixin}_l, m(e))$ where $e : S \to X$ is the morphism induced by $\vec{e}$
  - egdes that complete the pushout rectangle like for a single mixin
- for every edge $\mathrm{Edge}(l, d \to c, x)$ an edge $\mathrm{Edge}(\texttt{mixin}_1, \texttt{mixin}_d \to \texttt{mixin}_c, m(x))$.

Example 11. Continuing our running examples

BMIXIN `Additive MagmaExtensions`

creates the additive of every extension of `Magma`. The resulting diagram is:



Even more powerfully, we could use

BMIXIN `Additive` (**BCOMBINE** `MagmaExtensions`)

3.4 Advanced Operators

Universal algebra defines a range of theory operators that apply to an arbitrary theory with meta-theory `SFOL`. Examples include homomorphisms, products, submodels, quotients models or term algebras. Like the operators from Sect. 3.1 these usually introduce some morphisms. Moreover, many of them also admit batch version. Thus, they should be seen as diagram operators.

The implementation of all these operators in Mmt is still ongoing. Here we only sketch the non-batch case of the homomorphism operator as an example. Given a diagram D We write `HOM` D for the diagram that extends D with

- A new node `hom` whose theory is the theory of homomorphisms between two models m_1 and m_2 of $D^{\texttt{dist}}$. This theory contains two copies $D^{\texttt{dist}}$ for m_1 and m_2.
- Two edges from $D^{\texttt{dist}}$ to `hom` for the two morphisms m_1 and m_2.

Example 12. The node `hom` in the diagram `HOM Magma` contains the theory

```
theory MagmaHom : SFOL
  U1 : type
  op1: U1 → U1 → U1
  U2 : type
  op2: U2 → U2 → U2

  U  : U1 → U2
  op : ∀ [x,y:U1] (op2 (U x) (U y)) = U (op1 x y)
```

The morphism m_1 contains the assignments `U:=U1, op:=op1` and accordingly for m_2.

It is easy to see the power of, e.g., chaining batch operators for combine, mixin, and homomorphism to build large diagrams with a few commands.

Note that `HOM` D must return an `SFOL`-theory even if it is applied to a theory of unsorted first-order logic `FOL`. Therefore, we used `SFOL` as the fixed logic in order to simplify our running example. But we could just as well write all `Magma`-extension in `FOL` and translate them to `SFOL` before applying `HOM`. Because `FOL` and `SFOL` are also MMT theories, this translation is just a special case of a pushout along a morphism `FOL` $\to$ `SFOL` and thus can be expressed using the combine operation.

Finally, we sketch another operator to indicate how our diagram operators are independent of the meta-theory and can even include changes to the meta-theory:

Example 13 (Collection Types). The operator `LiftToTypeFamily` () maps a diagram of `SFOL`-theories to the corresponding diagram of polymorphic `SFOL`-theories: every `SFOL`-type declaration U : `type` is turned into a type operator U : `type` $\to$ `type`, and every constant is turned into a polymorphic constant that operates pointwise, e.g., `op` : $U \to U \to U$ becomes `op` : $\{a\}\ U\,a \to U\,a \to U\,a$ where $\{a\}$ binds a type variable a.

`Singletons` := `LiftToTypeFamily` (`Carrier`) `EXTEND` {`singleton` : $\{a\}\ a \to U\,a$}
`LiftedMagmas` := `LiftToTypeFamily` (`BCOMBINE MagmaExtensions`)

Now combining some of the nodes in `LiftedMagmas` with `Singletons` yields many of the usual collection types. In particular, `Monoid`, `IdempotentMonoid`, and `CommutativeIdempotentMonoid` yield theories describing polymorphic lists, multisets, and sets, respectively. Here the lifting of the magma operation is the concatenation/union of collections and the lifting of the neutral element is the empty collection.

4 Realization in MMT

In this section we describe the realization of diagram operators in MMT.

4.1 Diagram Definitions

Previously, MMT supported only named theories and morphisms. These were introduced by toplevel declarations that supported the examples from Sect. 2.1.

We extended this syntax in two ways. First, we added a new theory `Diagrams` that declares three MMT constants `anoynmous-theory`, `anonymous-morphism`, and `anonymous-diagram` along with notations and typing rules for building anonymous theories, morphism, and diagrams. This allows writing MMT expressions that represent the normal forms of theories, morphisms, resp. diagrams.

Second, we added a toplevel declaration for diagram definitions

$$\textbf{diagram}\ d : M := E$$

E is an expression that normalizes to a diagram and M is the meta-theory in which E is defined. In the simplest case, $M = \texttt{Diagrams}$, but M is typically a user-declared extension of `Diagrams` that declares additional diagram operators (see Sect. 4.2).

The semantics of diagram definitions is that E is evaluated into a normal diagram D and then a new theory/morphism of name d_l is defined for each new node/edge with label l in D.

Example 14. For example, the MMT expression

$$\begin{array}{l}\texttt{anoynmous-diagram}(\texttt{pres}, \texttt{Carrier} : \texttt{SFOL} = \{U : \texttt{type}\},\\ \quad \texttt{pres} : \texttt{SFOL} = \{U : \texttt{type}, \texttt{op} : U \to U \to U\},\\ \quad \texttt{extend} : \texttt{Carrier} \to \texttt{pres} = \{\ldots\})\end{array}$$

is the normal form of the diagram expression $E = \texttt{Carrier}\ \texttt{EXTEND}\,\{\texttt{op} : U \to U \to U\}$. Here the first argument `pres` indicates the label of the distinguished node.

The diagram definition

$$\textbf{diagram}\ \texttt{Magma} : \texttt{Diagrams} := E$$

normalizes E and finds a new node `pres` and a new edge `extend` in it. Thus, it creates a new theory named `Magma_pres` and a new morphism named `Magma_extend` of type `Carrier` $\to$ `Magma_pres`. The node `Carrier` is recognized as resulting from a reference to a previously defined theory and does not result in a new theory.

Because `pres` is the distinguished node, future references to `Magma` are coerced into `Magma_pres` if a theory is expected. Thus, users can use `Magma` as a theroy in the usual way. Similarly, if `Magma` is used where a morphism is expected, the implicit edge of the diagram (if exists) is used.

4.2 Diagram Combinators

With diagram definitions in place once and for all, we can start adding diagram combinators. Each combinator is defined by

- an MMT constant c that introduce the name and notation of the combinator,
- a rule that defines its semantics.

Here rules are user-supplied objects in MMT's underlying programming language that are injected into the MMT kernel as described in [12]. MMT automatically loads these rules at run time, and the MMT kernel always uses those rules that are visible to the current theory. Thus, users can add new combinators without rebuilding MMT, and different sets of combinators can be available in different scopes.

Example 15. We give a theory that introduces two of the combinators described in section 3.1.

```
theory Combinators =
    include Diagrams
    extends # 1 EXTEND {2,...} prec -1000000
    combine # COMBINE 1 {2,...} 3 {4,...} prec -2000000
    rule rules?ComputeExtends
    rule rules?ComputeCombine
```

The constant declarations define the concrete and abstract syntax for the diagram operators. And the rules refer to the Scala objects that define their semantics.[4]

Thus, users can flexibly add new diagram operators with arbitrarily complex semantics.

5 Conclusion and Future Work

Building large digital libraries of mathematics is a challenging task that requires special kind of tool support. Diagram-level operations achieve high levels of modularity and reuse, while avoiding a lot of boilerplate.

It remains future work to assess the behavior of the combinators as the size of the library gets much larger We are currently migrating the MathScheme library to MMT, which contains over a thousand theories built modularly using the combinators in [2].

A particular problem that future work will have to address is name generation. Batch operations introduce many theories/views at once, often with unwieldy names. It is non-trivial task to provide user-friendly syntax and we expect that satisfactory options can only be determined after conducting more experiments with large case studies. For now, we have simply added a top-level declarations **alias** $n := N$ that allows users to add a nice alias for an automatically generated name.

Acknowledgment

We would like to thank Jacques Carette, William Farmer, and Michael Kohlhase for fruitful discussions.

References

1. Autexier, S., Hutter, D., Mantel, H., Schairer, A.: Towards an Evolutionary Formal Software-Development Using CASL. In: Bert, D., Choppy, C., Mosses, P. (eds.) WADT. Lecture Notes in Computer Science, vol. 1827, pp. 73–88. Springer (1999)
2. Carette, J., O'Connor, R.: Theory Presentation Combinators. In: Jeuring, J., Campbell, J., Carette, J., Reis, G.D., Sojka, P., Wenzel, M., Sorge, V. (eds.) Intelligent Computer Mathematics. vol. 7362, pp. 202–215. Springer (2012)

[4] The Scala code is available at https://github.com/UniFormal/MMT/blob/devel/src/mmt-lf/src/info/kwarc/mmt/moduleexpressions/Combinators.scala.

3. Codescu, M., Mossakowski, T., Rabe, F.: Canonical Selection of Colimits. In: James, P., Roggenbach, M. (eds.) Recent Trends in Algebraic Development Techniques. pp. 170–188. Springer (2017)
4. CoFI (The Common Framework Initiative): CASL Reference Manual, LNCS, vol. 2960. Springer (2004)
5. The distributed ontology, modeling, and specification language. Tech. rep., Object Management Group (OMG) (10 2018), version 1.0
6. Farmer, W., Guttman, J., Thayer, F.: Little Theories. In: Kapur, D. (ed.) Conference on Automated Deduction. pp. 467–581 (1992)
7. Goguen, J., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J.: Introducing OBJ. In: Goguen, J., Coleman, D., Gallimore, R. (eds.) Applications of Algebraic Specification using OBJ. Cambridge (1993)
8. Kammüller, F., Wenzel, M., Paulson, L.: Locales – a Sectioning Concept for Isabelle. In: Bertot, Y., Dowek, G., Hirschowitz, A., Paulin, C., Thery, L. (eds.) Theorem Proving in Higher Order Logics. pp. 149–166. Springer (1999)
9. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: O. Grumberg and M. Huth (ed.) Tools and Algorithms for the Construction and Analysis of Systems 2007. Lecture Notes in Computer Science, vol. 4424, pp. 519–522 (2007)
10. Müller, D., Rabe, F.: Structuring Theories with Implicit Morphisms. In: Fiadeiro, J., Tutu, I. (eds.) Recent Trends in Algebraic Development Techniques. Springer (2019), to appear
11. Rabe, F.: How to Identify, Translate, and Combine Logics? Journal of Logic and Computation **27**(6), 1753–1798 (2017)
12. Rabe, F.: A Modular Type Reconstruction Algorithm. ACM Transactions on Computational Logic **19**(4), 1–43 (2018)
13. Rabe, F., Kohlhase, M.: A Scalable Module System. Information and Computation **230**(1), 1–54 (2013)
14. Sannella, D., Wirsing, M.: A Kernel Language for Algebraic Specification and Implementation. In: Karpinski, M. (ed.) Fundamentals of Computation Theory. pp. 413–427. Springer (1983)
15. Srinivas, Y., Jüllig, R.: Specware: Formal Support for Composing Software. In: Möller, B. (ed.) Mathematics of Program Construction. Springer (1995)