Structure-Preserving Diagram Operators

Navid Roux[0000−0002−8348−2441] and Florian Rabe[0000−0003−3040−3655]

Computer Science, FAU Erlangen-Nürnberg, Erlangen, Germany

Abstract. Theory operators are meta-level operators in logic that map theories to theories. Often these are functorial in that they can be extended to theory morphisms and possibly enjoy further valuable properties such as preserving inclusions. Thus, it is possible to apply them to entire diagrams at once, often in a way that the output diagram mimics any morphisms or modular structure of the input diagram. Our results are worked out using the Mmt language for structured theories instantiated with the logical framework LF for basic theories, but they can be easily transferred to other languages. We investigate how Mmt/LF diagram operators can be defined conveniently and give multiple examples.

1 Introduction and Related Work

Motivation Diagrams of theories and theory morphisms have long been used successfully to build large networks of theories both in algebraic specification languages [SW83, ST88, CoF04] and in deduction systems [FGT93, KWP99, SJ95, RK13]. In particular, they enable the use of meta-level operators on theories such as union and translation to build large structured theories in a modular way.

Recently, inspired by ideas in [DOL18, CO12], the second author generalized this idea to diagram operators [SR19], where an operator is applied not to a single theory but to an entire diagram. In this paper, we focus on the special case of diagram operators that are induced by applying operators on basic theories to every theory and morphism in a diagram at once. For example, we can form the diagram `Magmas` of the magma-hierarchy (defining theories for semigroup, commutative magma, etc.) and apply an operator to obtain in one step the corresponding diagram of homomorphisms (defining theories for semigroup homomorphisms, etc.).

Our main challenge here is to combine two conflicting goals. On the one hand, it must be **easy to define and implement** new diagram operators. This is critical for scalability as many diagram operators will be contributed by users, who may not be perfectly familiar with the overall framework and therefore need an interface as easy as possible. Moreover, it is critical for correctness: diagram operators tend to be very difficult to correctly specify and implement. On the other hand, diagram operators shine when applied to large diagrams, and **large diagrams are inevitably built with complex language features**. This applies both to the base logic, e.g., adding nodes to diagrams that require more expressive logics than originally envisioned, and to the structuring features for building larger theories.

Contribution We identify a class of diagram operators, for which these conflicting goals can be achieved. The general design uses two steps. Firstly, our diagram operators are **defined and implemented for flat theories** only. In particular, the language of structured theories remains transparent to users adding diagram operators. Secondly, every such definition is automatically **lifted to an operator on structured theories**. This lifting should preserve the structure and commute with flattening. Our work focuses on defining and implementing such diagram operators efficiently and reliably while providing only minimal meta-theoretical analysis.

We present several concrete examples. Firstly, many constructions from universal algebra fall in this class, e.g., the operator that maps a first-order theory T to the theory $\texttt{Hom}(T)$ of homomorphisms between two T-models. Secondly, we give an operator we call *polymorphify*, which maps a typed first-order theory T to a polymorphic theory $\texttt{Poly}(T)$, in which types become unary type operators. Applied to `Magmas`, this yields various theories of collection datatypes: polymorphic monoids (i.e., lists), polymorphic commutative monoids (i.e., multisets), and polymorphic bounded semilattices (i.e., sets) — all derived from the single concise expression $\texttt{Poly}(\texttt{Magmas})$. All of these operators are functorial and preserve structure, e.g., $\texttt{Poly}(\texttt{Magmas})$ contains theory morphisms and inclusions wherever `Magmas` does.

Our operators are defined within the framework for MMT diagram operators developed in [SR19]. We specialize our presentation to the LF logical framework [HHP93] as defined inside MMT. This choice is made for the sake of simplicity and concreteness, and our results can be easily transferred to other formal systems. More precisely, our results are applicable to any formal system whose syntax is described by a category of theories in which theories consist of lists of named declarations. In fact, MMT was introduced specifically as an abstract definition capturing exactly those formal systems [Rab17a]. Moreover, as LF is a logical framework designed specifically for representing the syntax and proof theory of formal systems, even the restriction to MMT/LF subsumes diagram operators for many important logics and type theories (see [CHK$^+$12] for examples). Our implementation is already applicable to any framework defined in MMT with LF as used here being just one example (see [MR19] for others).

Regarding structuring features, we limit attention to the two simplest and arguably most important features: definitions and inclusions. This choice is made for brevity, and our results generalize to the more complex structuring features supported by MMT such as renaming and translation.

Related Work In parallel to the present work, [CFS20] also expands on the development of diagram operators. That work has a similar focus and even includes some of the same examples as ours. It focuses on programmatically generating many theories derived from the algebraic hierarchy, whereas we focus on structure preservation.

[HR20] introduces syntactic theory functors in the setting where theories are pairs of a signature and a set of axioms. Because signatures are kept abstract, the setting cannot be directly compared to ours, but their treatment of axioms corresponds to ours, and several of their concrete examples fit our framework. They also consider what we will call include-preservation but do not consider morphisms.

Following [SR19], while our diagrams are formalized in MMT, our diagram operators are implemented as self-contained objects in the programming language underlying MMT. Recently, several systems for dependent type theory have introduced meta-programming capabilities [CB16, EUR+17]. That would allow defining diagrams and diagram operators in the same language and is an interesting avenue of future work. However, these systems tend to use different structuring principles and in particular do not support theory morphisms. Therefore, in practice our results cannot be directly ported to those systems even though they apply to them in theory.

Programmatic definitions of universal algebra have been done in multiple settings, e.g., in [Cap99] in Coq. Our work emphasizes the general framework in which these operators are defined and allows applications to structured theories. To the best of our knowledge, the polymorphify operator, while folklore in principle, is formalized here for the first time.

Overview We sketch the diagram operator framework of [SR19] and the theory structuring features of MMT in Section 2. Then we develop our main results in Section 3 and present example applications in Sections 4.1 and 4.2. We conclude in Section 5.

2 Preliminaries

The logical framework LF [HHP93] is a dependent type theory designed for defining a wide variety of formal systems including many variants of first- and higher-order logic and set and type theory [CHK+11]. We work with LF as realized in MMT [Rab17a], which induces a language of structured theories for any such formal system defined in LF. MMT uses structured theories and theory morphisms akin to algebraic specification languages like OBJ [GWM+93] and CASL [CoF04].

The **grammar** for MMT/LF is given in Figure 1. There and in the sequel, we use S, T for theory identifiers, v, w for morphism identifiers, and c and x for constant and variable identifiers, respectively. In the following, we first discuss flat theories and morphisms (the non-underlined parts of Figure 1) and state their semantics. Then we discuss our structuring features (underlined), and finally the parts of the grammar that deal with diagram expressions D.

The **flat theories** are inspired by LF-style languages and are anonymous lists of typed/kinded declarations $c\colon A$. Correspondingly, a **flat morphism** σ between two flat theories Σ and Σ' is an anonymous list of assignments $c := a$ such that, if $c\colon A$ is declared in Σ, then a must be a Σ'-expression of type $\overline{\sigma}(A)$, where $\overline{\sigma}$ is the homomorphic extension of σ. The flat theories and morphisms form a category $\mathbb{LF}^\flat$. Further details of the type system are not essential for our purposes, and we refer to [Rab17a].

Diag	::=	$(\mathit{Thy} \mid \mathit{Morph} \mid \textbf{install}\ D)^*$	diagrams
Thy	::=	$T = \{\mathit{Decl}^*\}$	theory definition
Decl	::=	$c\colon A\,\underline{[= A]} \mid \underline{\textbf{include}\ T}$	declarations in a theory
Morph	::=	$v\colon S \to T = \{\mathit{Ass}^*\}$	morphism definition
Ass	::=	$c := A \mid \underline{\textbf{include}\ v}$	assignments in a morphism
A	::=	$\mathtt{type} \mid c \mid x \mid A\ A \mid$ $\lambda x\!:\!A.\,A \mid \Pi x\!:\!A.\,A \mid A \to A$	terms
D	::=	$\mathrm{Diagram}(T^*, v^*) \mid O(D)$	diagram expressions

Fig. 1: MMT/LF Grammar

For example, the theory TFOL for typed first-order logic can be defined as below. There, prop and tp are the LF-types holding the TFOL-propositions and TFOL-types, respectively. For any TFOL-type $A\colon \mathtt{tp}$, the LF-type $\mathtt{tm}\,A$ holds the TFOL-terms of TFOL-type A. We only give a few connectives as examples. Following the judgments-as-types paradigm, the validity of a proposition $F\colon \mathtt{prop}$ is captured by the non-emptiness of the type $\vdash F$, which holds the proofs of F.

$$\mathtt{TFOL} = \left\{\begin{array}{ll} \mathtt{prop} & : \mathtt{type} \\ \mathtt{tp} & : \mathtt{type} \\ \mathtt{tm} & : \mathtt{tp} \to \mathtt{type} \\ \neg & : \mathtt{prop} \to \mathtt{prop} \\ \Rightarrow & : \mathtt{prop} \to \mathtt{prop} \to \mathtt{prop} \\ \doteq & : \Pi A\colon \mathtt{tp}.\ \mathtt{tm}\ A \to \mathtt{tm}\ A \to \mathtt{prop} \\ \forall & : \Pi A\colon \mathtt{tp}.\ (\mathtt{tm}\ A \to \mathtt{prop}) \to \mathtt{prop} \\ \vdash & : \mathtt{prop} \to \mathtt{type} \end{array}\right\}$$

The underlined parts in the grammar in Figure 1 are the structuring principles that LF inherits from MMT and that give rise to **structured theories and morphisms**. For simplicity, we restrict attention to the two most important structuring principles: defined constants and includes. In theories, a **definition** $c\colon A = a$ is valid if a has type A. In morphisms, defined constants are subject to the implicit assignment $c := \overline{v}(a)$. **Include** declarations allow combining theories into theories and morphisms into morphisms. MMT provides further structuring features, in particular for translating theories and renaming constants during an include. We expect our results to carry over to those features, but we omit them here due to space constraints.

TFOL-theories can now be represented as LF-theories that include TFOL, and similarly TFOL-theory morphisms as LF-morphisms that include the identity morphism of TFOL. More precisely, the image of this representation contains theories that include TFOL and then only add declarations of certain shapes, namely $t\colon \mathtt{tp}$ for a type symbol, $f\colon \mathtt{tm}\ t_1 \to \ldots \to \mathtt{tm}\ t_n \to \mathtt{tm}\ t$ for a function symbol, $p\colon \mathtt{tm}\ t_1 \to \ldots \to \mathtt{tm}\ t_n \to \mathtt{prop}$ for a predicate symbol, and $a\colon \vdash F$ for an axiom. More generally, we can choose to allow polymorphic declarations which are represented, e.g., as $\texttt{list}\colon \texttt{tp} \to \texttt{tp}$ and $\texttt{cons}\colon \Pi A\colon \texttt{tp}.\, \texttt{tm}\, A \to \texttt{tm}\, \texttt{list}\, A \to \texttt{tm}\, \texttt{list}\, A$.

The semantics of Mmt structuring features is given by the flattening operation $-^\flat$ that transforms structured theories and morphisms into flat ones as sketched in Figure 2. There, A^δ is the expression arising from A by recursively replacing any reference to a defined constant with its definiens.

We say a theory S is **included** into a theory T if $S^\flat \subseteq T^\flat$. This is the case, in particular, if **include** S occurs in the body of T. Thus, inclusion is a preorder relation on theories.

$$\begin{aligned}
S^\flat &= \Sigma^\flat \quad \text{if } S = \{\Sigma\} \\
\cdot^\flat &= \varnothing \\
(c\colon A, \Sigma)^\flat &= \{c\colon A^\delta\} \cup \Sigma^\flat \\
(c\colon A = a, \Sigma)^\flat &= \Sigma^\flat \\
(\textbf{include } T, \Sigma)^\flat &= T^\flat \cup \Sigma^\flat
\end{aligned}
\qquad
\begin{aligned}
v^\flat &= \sigma^\flat \quad \text{if } v\colon S \to T = \{\sigma\} \\
\cdot^\flat &= \varnothing \\
(c := A, \sigma)^\flat &= \begin{cases} \{c := A^\delta\} \cup \sigma^\flat & c\colon A' \text{ in } S \\ \sigma^\flat & c\colon A' = a' \text{ in } S \end{cases} \\
(\textbf{include } w, \sigma)^\flat &= w^\flat \cup \sigma^\flat .
\end{aligned}$$

Fig. 2: Flattening

Diagram expressions D were introduced in [SR19]: they are used in a third toplevel declaration **install** D, whose semantics is to declare all theories and morphisms in D at once. For our purposes, it is sufficient to consider only two simple cases for D: Diagram(T^*, v^*) builds an anonymous diagram by aggregating some previously defined theories and morphisms. And $O(D)$ applies a diagram operator O to the diagram D. Here, O is simply an identifier that Mmt binds to a user-provided computation rule implemented in the underlying programming language.

Remark 1. The exact nature of diagram expressions and the install declaration in Mmt is still somewhat of an open question. Here, we follow [SR19] and add them to the formal syntax of the language. Alternatively, we could relegate all diagram expressions to the meta-level. For example, a preprocessor could be used to compute $O(D)$ and generate the theory and morphism declarations in it.

One indication against the latter is that we have already identified some operators that take more arguments than just a diagram. For example, pushout takes a morphism m and returns the diagram operator $O = P_m$. Such operators can benefit from a tight integration with the type checker.

Further work with larger case studies is necessary to identify the most convenient syntax and work flow for utilizing diagram operators. However, large case studies are best done with structured diagrams, which is why we have prioritized the present work. In any case, the work presented here is independent of how that question is answered, and the current implementation can be easily adapted to other work flows.

For simplicity, we will restrict attention to `TFOL` in our examples below. But our results apply to any formal system defined in LF. In fact, Mmt allows implementing many other logical frameworks [MR19], and our results apply to most of them as well. In order to pin down the **limitations** of our work more precisely, we introduce the following definition: A formal system is called **straight** if

- its theories consist of lists of declarations $c\colon A$,
- the well-formedness of theories is checked declaration-wise, i.e., $\Sigma, c\colon A$ is a well-formed theory if c is a fresh name and A satisfies some well-formedness condition $\mathrm{WF}_\Sigma(A)$ relative to Σ, and
- $\mathrm{WF}_\Sigma(A)$ depends only on those declarations in Σ that can be reached by following the occurs-in relation between declarations.

Thus, in a straight language, $\mathrm{WF}_\Sigma(A)$ can be reduced to $\mathrm{WF}_{\Sigma_0}(A)$ where Σ_0 consists of the set of declarations in Σ whose names transitively occur in A. We call Σ_0 the **dependencies** of c. In a straight language, it is easy and harmless to **identify theories up to reordering of declarations**, and we will do so in the sequel.

Straightness is a very natural condition and is satisfied by LF and thus any formal system defined in it. However, there are practically relevant counterexamples. Most of those use proof obligations as part of defining $\mathrm{WF}_\Sigma(A)$ and discharging them may have to make use of all declarations in Σ. Typical examples are languages with partial functions, subtyping, or soft typing. (Many of these can be defined in LF as well, but only by enforcing straightness at the cost of simplicity.) Moreover, any kind of backward references such as in mutually recursive declarations violates straightness. We expect that our results can be generalized to such languages, but we have not investigated that question yet.

3 Structure-Preserving Diagram Operators

3.1 Motivation: The Pushout Operator

As a motivating example, we consider the pushout operator P_m for a fixed morphism $m\colon S \to T$ in the category $\mathbb{LF}^\flat$. P_m maps extensions X of S to the pair of $P_m(X)$ and m^X such that the square below on the left is a pushout. Moreover, as indicated below on the right, it extends to a functor by mapping morphisms f to the universal morphism out of $P_m(X)$.

$$\begin{tikzcd}
S \arrow[r, "m"] \arrow[d, hook] & T \arrow[d, hook] \\
X \arrow[r, "m^X"] & P_m(X)
\end{tikzcd}
\qquad
\begin{tikzcd}
S \arrow[r, "m"] \arrow[d, hook] \arrow[dd, hook, bend right] & T \arrow[d, hook] \arrow[ddr, hook, bend left] & \\
X \arrow[r, "m^X"] \arrow[d, "f"] & P_m(X) \arrow[dr, "P_m(f)"] & \\
Y \arrow[rr, "m^Y"] & & P_m(Y)
\end{tikzcd}$$

These are defined as

$$\begin{aligned}
P_m(S) &= T & P_m(S, \Sigma, c\colon A) &= P_m(S, \Sigma), c\colon \overline{m^{S,\Sigma}}(A) \\
P_m(id_S) &= id_T & P_m(id_S, \sigma, c := a) &= P_m(id_S, \sigma), c := \overline{m^Y}(a) \\
m^S &= m & m^{S,\Sigma,c:A} &= m^{S,\Sigma}, c := c
\end{aligned}$$

We can observe a number of abstract structural properties that are helpful for implementing this operator at large scales and that we often see in other operators. Firstly, it maps extensions of S to extensions of T. And it can be extended to a functor mapping morphisms $f\colon X \to Y$ that agree with id_S (i.e., commutative triangles of S over $X \xrightarrow{f} Y$) to morphisms between extensions of T that agree with id_T. The following definition captures this property:

Definition 1 (Functorial). *A* ***functorial*** *theory operator is a functor from a subcategory of $\mathbb{LF}^\flat$ to $\mathbb{LF}^\flat$. Thus, O is a partial map of objects and morphisms such that: if O is defined for an object X, it is also defined for id_X; if it is defined for morphisms f, g, it is also defined for their domains, codomains, and (if composable) their composition.*

We say that O is ***from*** *S* ***to*** *T if it (i) is only defined for theories extending S and then returns theories extending T, and (ii) if defined for theories X, Y, is only defined for morphisms $f\colon X \to Y$ that agree with id_S and then returns morphisms that agree with id_T.*

Thus, P_m is functorial from S to T for $m\colon S \to T$.

This situation is very common. For example, it is typical for transporting developments relative to a base language S to a different base language T. Concretely, to represent a language L (e.g., a type theory or logic) in LF, we usually employ an LF-theory S to represent the syntax and possibly proof calculus of L. L-theories are then represented as LF-theories extending S, and L-morphisms between L-theories as LF-morphisms that agree with id_S. Now given two such representations S and T for two languages, many language translations can be represented as functorial operators from S to T.

Remark 2 (Partiality). In Definition 1, we do *not* require that O is defined for all extensions of S. There are two reasons for this. Firstly, the representation of L-theories as LF-theories extending S is usually not surjective. Theory operators specific to L can usually only be defined on the image of that representation. Secondly, defining operators may run into name clashes: it may be straightforward to define a functor up to renaming of declarations (a special case of isomorphism) but awkward or impossible to do so on the nose. For example, we discussed this problem in depth for the pushout functor in [CMR17]. In those cases, the best trade-off may sometimes be to leave the operator undefined for some inputs.

Remark 3 (Functoriality). In Definition 1, we do require that O is functorial where defined. In our experience, there are few interesting operators where functoriality fails. Essentially, it means that operators (i) are defined for morphisms, not just for theories, and (ii) preserve identity and composition of morphisms. An operator failing the first requirement usually cannot be extended to theories with definitions – a critical requirement in practice. And an operator failing the second requirement would be highly unusual. It is, however, common to find transformations in the literature where only the definition on theories is worked out and the extension to a functorial operator requires additional work.

A second property of the pushout operator that often occurs in general is that it is defined declaration-wise: the declarations in the input are processed in a row, each resulting in a declaration that occurs in the output theory. Moreover, the output only depends on the input declaration (and its dependencies) and not on any other preceding or succeeding declaration. An abstract way to capture the essence of this property is the following:

Definition 2 (Include-Preservation). *An operator O is* ***include-preserving*** *if whenever O is defined for flat LF-theories X, Y, we have that $X \subseteq Y$ implies $O(X) \subseteq O(Y)$ and accordingly for morphisms.*

This property is frequently fulfilled for theory operators because they are typically defined by induction on the list of declarations in a theory, as is the case with the pushout operator.

As a third property, include-preservation in particular allows us to extend the pushout to diagrams of structured theories: if we map every theory T in a diagram to T', we can map every declaration **include** T to **include** T' and thus preserve the structure. Similarly, the pushout can be extended to theories with definitions. If the input theory contains the declaration $c\colon A = a$, we can put the definition $c\colon \overline{m^{S,\Sigma}}(A) = \overline{m^{S,\Sigma}}(a)$ into the output theory. Then in any subsequent occurrence of c in the input, we do not need to expand the definition before applying m^X. Instead, we can map any occurrence of the defined constant c in X to the correspondingly defined constant c in $P_m(X)$.

Definition 3 (Structure-Preservation). *A partial map O on LF-diagrams is a* ***structure-preserving*** *diagram operator if the following holds: if $O(D)$ is defined for a diagram D, then for every theory/morphism named n in D there is a corresponding theory/morphism named $O(n)$ in $O(D)$, and we have $O(n)^\flat = O(n^\flat)$.*

If we extend P_m to diagrams using includes and definitions as sketched above, we obtain a structure-preserving diagram operator.

Structure-preservation is satisfied by many interesting diagram operators. However, it is often awkward to define them — it is much more convenient to define an operator only for flat theories/morphisms, where category theory provides an elegant and expressive meta-theory. Indeed, many formal definitions in the literature do only that.

3.2 Linear Operators

Even though the preceding properties make intuitively sense to demand upon operators, they are also abstract in nature and do not help much users in easily specifying and implementing operators. We tackle this problem in the sections to come by developing a more accessible class of operators which are given more concretely and yet fulfill the abstract properties.

In particular, we start by identifying general patterns of operators on flat theories that allow lifting them to structure-preserving diagram operators. The following definition identifies a large class of such cases:

Definition 4 (Linear Operator). *Given two theories S and T, a **linear** diagram operator O from S to T is a partial endofunctor on flat theories mapping extensions of S to extensions of T such that*

- *O is defined for extensions of S declaration-wise:*

$$O(S) = T \qquad O(S, \Sigma, c\colon A) = O(S, \Sigma), \Delta_\Sigma(c\colon A)$$

  for some $\Delta_-(-)$,
- *O is defined similarly for morphisms $S, \Sigma \to S, \Sigma'$ that are of the form id_S, σ:*

$$O(id_S) = id_T \qquad O(id_S, \sigma, c := a) = O(id_S, \sigma), \delta_\sigma(c := a)$$

  for some $\delta_-(-)$,
- *the definedness and result of $\Delta_\Sigma(c\colon A)$ are determined by $\Delta_{\Sigma_0}(c\colon A)$ where Σ_0 are the dependencies of c, and*
- *the definedness and result of $\delta_\sigma(c := a)$ are determined by $\delta_{\sigma_0}(c := a)$ where σ_0 is the fragment of σ acting on the dependencies of c.*

It is straightforward to see that P_m is linear for every m. Moreover, we have the following basic property:

Theorem 1. *Every linear operator is functorial from S to T and preserves includes.*

Proof. Straightforward.

In contrast, functorial operators that preserve includes are not necessarily linear. For example, by setting $O(\Sigma) = \{c_1\_c_2\colon \texttt{type} \mid c_1, c_2 \in \Sigma\}$ we can define an operator that produces a single declaration for every ordered pair of declarations in Σ. This operator still preserves includes but fails to fulfill the above third requirement on linear operators.

The functions Δ and δ serve to output a *list* of declarations for every input declaration. They are uniquely determined by O, and vice versa, (fixing S and T) O is uniquely determined by Δ and δ. Therefore, to implement linear operators, we can fix $O(S)$ and implement the functions Δ and δ. Moreover, the argument Σ in $\Delta_\Sigma(c\colon A)$ is only needed to look up the types of constants occurring in Σ. Thus, implementing a linear operator essentially means to map declarations $c\colon A$ individually. That makes them much simpler to implement than arbitrary operators on theories.

Concerning the declarations output by Δ and δ, the pushout operator also satisfies the following stricter property:

Definition 5 (Strongly Linear Operator). *A linear operator is called **strongly linear** if $\Delta_\Sigma(c\colon A)$ always contains exactly one declaration, which is also named c.*

Strongly linear operators are even simpler to implement because they are already determined by a pair (E, ε) of expression translation functions by means of $\Delta_\Sigma(c\colon A) = c\colon E_\Sigma(A)$ and $\delta_\sigma(c := a) = c := \varepsilon_\sigma(a)$. Namely, developers of operators only need to worry about the inductive expression translation functions, and the framework can take over all bureaucracy for names and declarations. Moreover, it is even simpler to check that two arbitrary functions (E, ε) induce a strongly linear operator. We omit the details, but essentially we only have to check that the typing judgement $a\colon A$ implies $\varepsilon(a)\colon E(A)$.

For the pushout operator we even have $E_\Sigma = \varepsilon_\sigma$, but that is not common enough to deserve its own definition. However, it is often the case that E_Σ and ε_σ are very similar, e.g., they might be the same except that E inserts a Π-binder where ε inserts a λ one.

3.3 Structure-Preserving Lifting of Linear Operators

Our goal now is to extend linear operators O from S to T — which by definition act on flat theories and morphisms only — to operators on diagrams D of structured theories. We make some simplifying assumptions to state our rigorous definitions.

Firstly, we assume that every theory in D starts by including S and every morphism by including id_S. Correspondingly, theories and morphisms in $O(D)$ will include T and id_T, respectively. However, we will not mention these includes in the syntax and instead assume that they are fixed globally for the entire diagram. This mimics how diagrams are actually implemented in MMT. For example, S could be `TFOL` and D a diagram of `TFOL`-theories. It would then be very awkward to manually include `TFOL` in every theory in D. Instead, MMT provides a mechanism for declaring D to be a `TFOL`-diagram, in which case `TFOL` is available in all theories and fixed by all morphisms in D.

Secondly, we assume that D is self-contained, i.e., every theory mentioned in D, be it in an include declaration or as the (co)domain of a morphism, is also defined in D.

Thirdly, we assume that there are no name clashes between theories defined in D and the theory T: other than the names already declared in S, no theory in D should declare names also used in T. In particular, this means that no theory in D may include T or any named theory that is transitively included into T (unless S itself includes that theory). This requirement can be dropped in practice because MMT anyway distinguishes two constants of the same name, one declared in a theory in D and one declared in T. But it is necessary here to (i) avoid explaining MMT's name disambiguation mechanism, and (ii) alert to this subtlety any reader who plans to implement the lifting in other systems.

Our output will be the diagram $O(D)$ that contains the results of applying O to every theory and morphism in D. Our goal is that $O(D)$ mirrors as much of the structure of D as possible.

We need to generate fresh names for the new theories and morphisms in $O(D)$. These new names should not be arbitrary but obtained from the names in D in a systematic way that is predictable for the user. That is necessary to enable users to actually make use of the new theories and morphisms after having applied O to obtain them. For example, when applying the operator

P_m to a theory defined as $X = \{\Sigma\}$, we could obtain the theory definition $\texttt{pushout}.m.X = \{P_m(\Sigma)\}$. Therefore, we define:

Definition 6. *A **liftable operator** O is an operator together with a function that maps names to names. We denote this function also by $O(-)$. The composition of liftable operators is defined by composing both the operators and their name maps.*

Thus, we can map the theory definition $X = \{\Sigma\}$ to the theory definition $O(X) = \{O(\Sigma)\}$. To avoid confusion, keep in mind that X, Y, v, w are always names that O maps to other names, and that Σ and σ are lists of declarations/assignments for which O is defined inductively.

Then we can finally define the lifting:

Definition 7 (Lifting). *Given a linear liftable operator O, we define its lifting $O(-)$ to diagrams as follows:*

- *For every theory $X = \{\Sigma\}$ in D, $O(D)$ contains the theory $O(X) = \{O(\Sigma)\}$ where*

$$O(\cdot) \;=\; \cdot \qquad O(\Sigma,\; c\colon A) \;=\; O(\Sigma),\; \mathit{\Delta}_\Sigma(c\colon A)$$
$$O(\Sigma,\; c\colon A = a) \;=\; O(\Sigma),\; \mathrm{AsDef}(\delta_{id_\Sigma}(c := a))$$
$$O(\Sigma,\; \textbf{include}\; Y) \;=\; O(\Sigma),\; \textbf{include}\; O(Y)$$

  and AsDef$(-)$ *as defined below.*
- *For every morphism $v\colon X \to Y = \{\sigma\}$ in D, $O(D)$ contains the morphism $O(v)\colon O(X) \to O(Y) = \{O(\sigma)\}$ where*

$$O(\cdot) \;=\; \cdot \qquad O(\sigma,\; c := a) \;=\; O(\sigma),\; \delta_\sigma(c := a)$$
$$O(\sigma,\; \textbf{include}\; w) \;=\; O(\sigma),\; \textbf{include}\; O(w).$$

Here, the subscripts of $\mathit{\Delta}$ and δ technically must be flattened first. But we omit that from the notation.

Definition 7 looks complex but is mostly straightforward. It uses $\mathit{\Delta}$ and δ for the flat cases and maps includes to includes. The only subtlety is the case for defined constants. Here, we exploit the general property of MMT that a constant definition $\Sigma,\; c\colon A = a$ is well-formed iff $id_\Sigma,\; c := a$ is a well-formed morphism from $\Sigma,\; c\colon A$ to Σ. This is easy to see — in both directions the key property is the typing judgment $\vdash_\Sigma a\colon A$. This equivalence lets us turn a definition into a morphism assignment, apply O, and then turn the resulting assignments back into definitions. Using the morphism above, O yields the morphism $O(id_\Sigma),\; \delta_{id_\Sigma}(c := a)$ from $O(\Sigma),\; \mathit{\Delta}_\Sigma(c\colon A)$ to $O(\Sigma)$. We define the function AsDef$(-)$ to turn this morphism back into a sequence of definitions: it replaces every assignment $c' := a'$ for a constant $c'\colon A'$ in $\mathit{\Delta}_\Sigma(c\colon A)$ with the definition $c'\colon A' = a'$.

AsDef$(-)$ is only needed in our definition on paper. When implementing the lifting in MMT, it is not even needed because declarations in a theory and assignments in a morphism are internally represented in the same way.

Remark 4 (Strongly Linear Lifting). By definition, every strongly linear operator O with expression translation functions (E, ε) is linear, and hence its lifting (via an arbitrary name map) is a special case of the above.

The case of defined constants is particularly interesting: Applying the definition, we see that it maps defined constants by

$$O(\Sigma,\ c\colon A = a) = O(\Sigma),\ c\colon E_{\Sigma}(A) = \varepsilon_{id_{\Sigma}}(a)$$

Thus, E and ε can be seen as using one translation function for types and one for typed terms.

3.4 Structure-Preservation of the Lifting

It remains to show that the lifted operators behave as expected. This is captured formally in our main theorem:

Theorem 2 (Structure Preservation). *Consider a well-formed diagram D and an operator O as in Definition 7. Then, $O(D)$ is a well-formed diagram, and*

- *for every theory named X defined in D, we have $O(X^\flat) = O(X)^\flat$*
- *for every morphism named v defined in D, we have $O(v^\flat) = O(v)^\flat$.*

In particular, O is structure-preserving.

Proof. The proof of well-formedness of $O(D)$ proceeds by induction on the well-formedness derivation for D in the MMT/LF calculus. Only the case for defined constants is non-obvious. The argument for that case is already sketched in the remarks explaining the use of AsDef$(-)$ after Definition 7.

Because linear operators preserve includes, it is easy to see that the preservation statements hold for diagrams that do not contain definitions.

To account for definitions, we observe that a constant c' in $O(X)$ is defined iff the lifting generates it when translating a defined constant c in X. Moreover, linearity guarantees that the lifting handles undefined constants in the same way regardless of whether defined constants in the same theory are eliminated or not. Thus, $O(X^\flat)$ and $O(X)^\flat$ contain the same declarations.

3.5 Composing Operators

While we do not focus on the meta-theoretical analysis of our operators, it is critical to study the composition of operators. This is because our language allows repeated application as in $O'(O(D))$. All properties defined above are preserved under composition in the following sense:

Theorem 3 (Composition of Theory Operators). *Consider two operators O, O' on $\mathbb{LF}^\flat$ such that O' is defined on the image of O. Then:*

- *If O is functorial (from R to S) and O' is functorial (from S to T), then $O' \circ O$ is functorial (from R to T).*
- *If O and O' preserve includes, so does $O' \circ O$.*

Proof. All proofs are straightforward.

Theorem 4 (Composition of Diagram Operators). *Consider two liftable operators O, O' on LF-diagrams such that O' is defined on the image of O. Then:*

- *If O and O' are structure-preserving, then so is $O' \circ O$.*
- *If O and O' are (strongly) linear, then so is $O' \circ O$. Moreover, its lifting is the composition of the liftings of O' and O.*

Proof. All proofs are straightforward.

4 Applications

Linear operators occur in all areas of formal languages. Before we describe some operators in detail below, we give a list of examples that we are aware of:

- Universal algebra provides many constructions that correspond to linear operators from `TFOL` to itself. Examples include the operators that map any `TFOL`-theory X to the theory of homomorphisms between two X-models, the submodels of an X-model, and congruence relations of an X-model, respectively. [CFS20] reports on a systematic survey identifying dozens more, which are applied uniformly to a diagram of over 200 algebraic theories.
- As described in [CHK$^+$11], many translations are already special cases of the pushout P_m for an appropriate m. Examples include the relativization translations from modal or description logics to unsorted first-order logic and the interpretation of type theory in set theory.
  But many more complex translations are functorial operators from S to T that cannot be represented as pushouts, and that was one of the original motivations of the present work. Typical examples are translations that use encodings to eliminate features such as:
  - The negative translation from classical to intuitionistic logic is strongly linear but cannot be given by a pushout: being strongly linear, the output theory declares a propositional variable p whenever the input theory does; but all references to p are translated to $\neg\neg p$ (whereas the pushout would translate them to p).
  - The type-erasure translation from typed to untyped first-order logic is linear but not strongly linear: it maps every typed declaration to an untyped declaration *and* an axiom capturing the typing properties.
- Logical frameworks often allow multiple different encodings for the same formal system, in which case there are often systematic syntactic transformations that mediate between them. For example, many type theoretical features (function types, product types, etc.) can be formalized both using intrinsic (terms carry their type) and using extrinsic (terms are assigned types by the environment) encodings of typing. Because both have their merit, they are usually both done in practice. Using a linear operator, we cannot only generate one from the other, but we can also generate the transformations between them.

– Many structural operations needed to manage large libraries can be seen as linear operators. Examples include the conversion between different packaging conventions in the sense of [GGMR09] and promotion in Z in the sense of [CAPM15]. Often these are written ad hoc for a subset of the theories of the library, sometimes multiple times for the same theory. They are especially valuable to automate in order to reduce workload and ensure the uniform application of boilerplate constructions across the library.

4.1 Universal Algebra

The algebraic hierarchy in which theories are developed is a prime example of a diagram of structured theories due to the high degree of reuse in that domain. Therefore, the combination of the two is an ideal match. We develop two operators as examples: one that maps a `TFOL`-theory X (whose models are the X-models) to the `TFOL`-theory of X-homomorphisms (whose models are the homomorphisms between X-models). And another that maps a `TFOL`-theory X to the `TFOL`-theory of X-submodels (whose models are the pairs of a X-model and a submodel of it). Other constructions such as product and quotient models can be handled accordingly.

In both cases, we use the following structured diagram of magma-based theories as an example:

$$\begin{tikzcd}
 & & \texttt{Comm} & \\
\texttt{Set} \arrow[r, hook] & \texttt{Magma} \arrow[ur, hook] \arrow[dr, hook] & & \\
 & & \texttt{SemiGr} \arrow[r, hook] & \texttt{Monoid}
\end{tikzcd}$$

We start with $\texttt{Set} = \{U : \texttt{tp}\}$ and build the hierarchy in a straightforward way:

$$\texttt{Magma} = \left\{\begin{array}{l}\textbf{include } \texttt{Set} \\ \circ : \texttt{tm}\ U \to \texttt{tm}\ U \to \texttt{tm}\ U\end{array}\right\} \qquad \texttt{Comm} = \left\{\begin{array}{l}\textbf{include } \texttt{Magma} \\ \texttt{ax\_comm} : \vdash \ldots\end{array}\right\}$$

$$\texttt{SemiGr} = \left\{\begin{array}{l}\textbf{include } \texttt{Magma} \\ \texttt{ax\_assoc} : \vdash \ldots\end{array}\right\} \qquad \texttt{Monoid} = \left\{\begin{array}{ll}\textbf{include } \texttt{SemiGr} & \\ \texttt{e} & : \texttt{tm}\ U \\ \texttt{ax\_neut} : \vdash \ldots & \end{array}\right\}$$

For later application of our operators, let us by `Magmas` abbreviate the Mmt diagram expression Diagram(`Set`, `Magma`, `Comm`, `SemiGr`, `Monoid`).

Homomorphisms We define the operator `Hom` such that it is linear by construction and only define Δ and δ. We map every type, function, or predicate symbol declaration $c : A$ via Δ to three declarations. Two of them serve as renamed copies to build up the domain and codomain of the homomorphism, and the third declaration accounts for the actual homomorphism. Axioms are mapped to two declarations only, one each for domain and codomain:

– In a type symbol declaration, we have $A = \mathtt{tp}$, and $\Delta_\Sigma(t\colon A)$ contains

$$t^d\colon \mathtt{tp},\ t^c\colon \mathtt{tp},\ t^h\colon \mathtt{tm}\ t^d \to \mathtt{tm}\ t^c$$

Here, the superscripts indicate different systematic renamings of the constant t.

– In a function symbol declaration, A is of the form $\mathtt{tm}\ t_1 \to \ldots \to \mathtt{tm}\ t_n \to \mathtt{tm}\ t$, and $\Delta_\Sigma(f\colon A)$ contains

$$\begin{array}{l} f^d\colon A^d, \qquad f^c\colon A^c, \\ f^h\colon \vdash \forall x_1\colon \mathtt{tm}\ t_1^d, \ldots, x_n\colon \mathtt{tm}\ t_n^d.\ \ t^h(f^d(x_1,\ldots,x_n)) \doteq f^c(t_1^h(x_1),\ldots,t_n^h(x_n)) \end{array}$$

Here and below, the superscripts as in A^d indicate the translation of the expression A by renaming all constants occurring in it.

– In a predicate symbol declaration, A is of the form $\mathtt{tm}\ t_1 \to \ldots \to \mathtt{tm}\ t_n \to \mathtt{prop}$, and $\Delta_\Sigma(p\colon A)$ contains

$$\begin{array}{l} p^d\colon A^d, \qquad p^c\colon A^c, \\ p^h\colon \vdash \forall x_1\colon \mathtt{tm}\ t_1^d, \ldots, x_n\colon \mathtt{tm}\ t_n^d.\ \ p^d(x_1,\ldots,x_n) \Rightarrow p^c(t_1^h(x_1),\ldots,t_n^h(x_n)) \end{array}$$

– In an axiom declaration, A is of the form $\vdash F$, and $\Delta_\Sigma(a\colon A)$ contains $a^d\colon A^d,\ a^c\colon A^c$.

Moreover, for any TFOL-theory $X = \{\Sigma\}$, we define the theory morphism $X^d\colon X \to \mathtt{Hom}(X) = \{\sigma^d\}$ where σ^d contains **include** Y^d for every **include** Y in Σ and $s := s^d$ for every constant s in Σ. Analogously, we define the morphism $X^c\colon X \to \mathtt{Hom}(X) = \{\sigma^c\}$. Thus, Hom is more than a functor: it maps a single theory X to the diagram

$$\begin{tikzcd} X \arrow[r, shift left, "X^d"] \arrow[r, shift right, "X^c"'] & \mathtt{Hom}(X). \end{tikzcd}$$

It is this diagram-valued behavior that inspired MMT diagram operators in the first place.

The definition of δ is the same in principle but there is a subtle issue: Hom is not actually functorial on all morphisms between TFOL-theories. We refer to [Rab17b] for a detailed analysis of the problem. Because our framework uses partial functors anyway, we can remedy this with the following definition: A formula is called *monotone* if it only uses the connectives $\wedge$, $\vee$, $\doteq$, and $\exists$ (but not $\neg$, $\Rightarrow$, or $\forall$). The intuition behind monotone formulas is that they are preserved along homomorphisms. Then we restrict the applicability of Hom as follows: If a TFOL-theory contains a defined predicate symbol, the definiens must be monotone. And in a morphism between TFOL-theories, all predicate symbols must be assigned a monotone expression. This partiality is a bit awkward but desirable in practice: users trying to apply Hom to a theory or morphism violating the restriction are usually unaware of this issue and thus have inconsistent expectations of the behavior of Hom.

After making this restriction, we can define δ as follows:

- For a type symbol assignment, $\delta_\sigma(t := E) = t^d := E^d,\ t^c := E^c,\ t^h := E^h$. Here, the superscripts on E again denote the translations that replace every constant accordingly. Note that in TFOL, up to β-normalization in LF, E can only be a type *symbol*, never a complex expression.
- For a function symbol $f\colon A$ as above, $\delta_\sigma(f := E) = f^d := E^d,\ f^c := E^c,\ f^h := P$. Here, P is a proof that

$$\forall x_1\colon \mathtt{tm}\ \widehat{t_1^d}, \dots, x_n\colon \mathtt{tm}\ \widehat{t_n^d}.\quad \widehat{t^h}\left(E^d(x_1, \dots, x_n)\right) \doteq E^c\left(\widehat{t_1^h}(x_1), \dots, \widehat{t_n^h}(x_n)\right),$$

  where the circumflex $\widehat{\ }$ abbreviates the application of $\overline{\mathtt{Hom}(\sigma)}$; a property provable in TFOL for every well-typed expression E.
- For a predicate symbol $p\colon A$ as above, $\delta_\sigma(p := E) = p^d := E^d,\ p^c := E^c,\ p^h := P$. Here, P is a proof that

$$\forall x_1\colon \mathtt{tm}\ \widehat{t_1^d}, \dots, x_n\colon \mathtt{tm}\ \widehat{t_n^d}.\ E^d\left(x_1, \dots, x_n\right) \Rightarrow E^c\left(\widehat{t_1^h}(x_1), \dots, \widehat{t_n^h}(x_n)\right),$$

  a property provable in TFOL for every monotone well-typed expression E.
- For an axiom symbol assignment, $\delta_\sigma(a := E) = a^d := E^d,\ a^c := E^c$.

Theorem 5. Hom(−) *is linear from* TFOL *(restricted to monotone theories and morphisms) to* TFOL*.*

Proof. It is well-known that Hom as defined above is indeed a functor, see e.g., [Rab17b]. Then linearity is obvious.

Consequently, we can immediately apply Hom to our diagram of structured theories: The MMT declaration **install** Hom(Magmas) adds the encircled diagram shown in Figure 3 to the previous diagram Magmas also shown there. As an example, we show the generated theories for Set and Magma:

$$\mathtt{Hom(Set)} = \left\{\begin{array}{ll} U^d & : \mathtt{tp} \\ U^c & : \mathtt{tp} \\ U^h & : \mathtt{tm}\ U^d \to \mathtt{tm}\ U^c \end{array}\right\}$$

$$\mathtt{Hom(Magma)} = \left\{\begin{array}{ll} \multicolumn{2}{l}{\textbf{include}\ \mathtt{Hom(Set)}} \\ \circ^d & : \mathtt{tm}\ U^d \to \mathtt{tm}\ U^d \to \mathtt{tm}\ U^d \\ \circ^c & : \mathtt{tm}\ U^c \to \mathtt{tm}\ U^c \to \mathtt{tm}\ U^c \\ \circ^h & : \vdash \forall x_1\colon \mathtt{tm}\ U^d, x_2\colon \mathtt{tm}\ U^d. \\ & \quad U^h(x_1 \circ^d x_2) \doteq U^h(x_1) \circ^c U^h(x_2) \end{array}\right\}$$

Submodels We proceed in essentially the same way to define a theory that pairs a model with one of its submodels. Again, we translate and rename all declarations of the input, but this time only once using the superscript ^{p} for the parent of the submodel. We use the superscript ^{s} for the declarations building the submodel. We define Δ by:

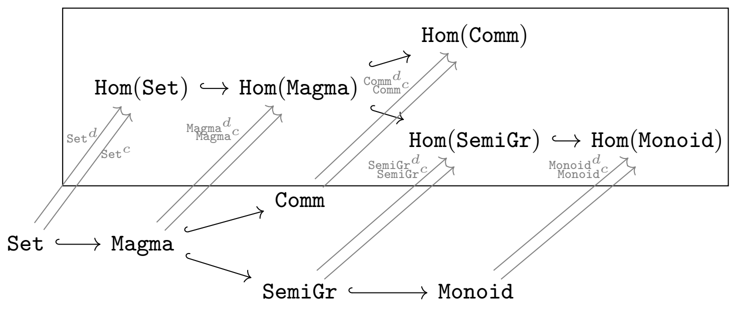

Fig. 3: Diagrams Magmas and (boxed) Hom(Magmas)

- Every type symbol declaration $t\colon \mathtt{tp}$ is mapped to two declarations: $t^p\colon \mathtt{tp}$ for the parent model and $t^s\colon \mathtt{tm}\ t^p \to \mathtt{prop}$ for the predicate singling out the values of the submodel.
- Every function symbol declaration $f\colon A$ (of the form as above) is mapped to two declarations, $f^p\colon A^p$ for the parent model and

$$f^s\colon \vdash \forall x_1\colon \mathtt{tm}\ t_1^p, \ldots, x_n\colon \mathtt{tm}\ t_n^p.\ \ t_1^s(x_1) \wedge \ldots \wedge t_n^s(x_n) \Rightarrow t^s(f^p(x_1, \ldots, x_n))$$

  for the closure property of the submodel.
- Predicate symbols are not relevant for submodels, and we simply translate $c\colon A$ to $c^p\colon A^p$.
- Axioms $a\colon \vdash F$ are translated to $a^p\colon \vdash F^p$ for the parent model and $a^s\colon \vdash F'$ for the submodel. Here, F' arises from F^p by relativizing every quantifier on a type t^p using the predicate t^s.

In the same way as for Hom, we obtain a morphism $X^p\colon X \to \mathtt{Sub}(T)$ that assigns $c := c^p$. Hypothetically, if we worked in a stronger logic that allowed taking subtypes, we could define a morphism $X^s\colon X \to \mathtt{Sub}(X)$ that maps, e.g., a type t to the subtype of t^p given by t^s.

Finally, we define δ by:

- Every type symbol assignment $t := E$ is mapped to $t^p := E^p$, $t^s := E^s$.
- Every function symbol assignment $f := E$ is mapped to $f^p := E^p$, $f^s := P$ where P is a proof that the result of E is in the submodel whenever all its arguments are.
- Predicate symbol assignments $c := E$ are mapped to $c^p := E^p$.
- Axiom assignments $a := E$ are translated to $a^p := E^p$, $a^s := E'$ where E' arises from E^p by adapting all occurrences of quantifier rules to the relativized variants.

Theorem 6. *The operator* Sub$(-)$ *is linear from* TFOL *to* TFOL.

Proof. The proof proceeds as for Hom.

These examples give a first glimpse of the power of composing diagram operators. We can, for example, use **install** `Hom(Sub(Magmas))` to obtain the theories of homomorphisms between submodels. By further combining that with a diagram operator for coequalizers, we can then obtain the theory of homomorphisms between submodels of the *same* parent model.

Future Work: Provable Axioms The submodel example indicates an important avenue of future work: many of the axioms a^s we copy to $\mathtt{Sub}(X)$ are in fact provable. For example, all axioms using only $\forall$ and $\doteq$ (i.e., the axioms primarily used in universal algebra) are automatically true in each submodel. Here, `Sub` could do much better and translate such axioms to theorems by synthesizing an appropriate proof and adding it as the definiens of a^s. More generally, we could run a theorem prover on every axiom we generate (independent of its shape) and generate a definiens whenever a proof can be found.

This issue can be more subtle as a similar example with `Hom` shows: sometimes the generated axioms become provable in the context of stronger theories. For example, every magma homomorphism between groups is automatically a group homomorphism. Thus, the preservation axioms e^h for the neutral element and i^h (where i is the unary function symbol for the inverse element) are provable in the theory `Hom(Group)`, at which place it would therefore be desirable for `Hom` to add definitions. However, this makes it trickier for the operator to be structure-preserving: the theory `Hom(Monoid)` must still contain e^h without a definition, and the definition should only be added when `Hom(Monoid)` is included into `Hom(Group)`.

4.2 Polymorphic Generalization

We give a linear operator $\mathtt{Poly}(-)$ from `TFOL` to polymorphic first-order logic `PFOL`. `PFOL` uses the same LF-theory as `TFOL`. But the well-formed symbol declarations in a `PFOL`-theory are more general: they are like the ones in `TFOL`-theories except that they may be polymorphic in type variables. Concretely, `PFOL` allows type *operator* declarations $t\colon \mathtt{tp} \to \ldots \to \mathtt{tp} \to \mathtt{tp}$ and *polymorphic* function symbol declarations $f\colon \Pi\, u_1 : \mathtt{tp}.\ \ldots\ \Pi\, u_m : \mathtt{tp}.\ \mathtt{tm}\ t_1 \to \ldots \to \mathtt{tm}\ t_m \to \mathtt{tm}\ t$, where all t_i and t may now contain the type variables u_i. Accordingly, `PFOL` allows polymorphic predicate symbols and polymorphic axioms.

`Poly` is strongly linear by construction, i.e., we only need to give expression translation functions E_Σ for types in declarations $c\colon A$ and ε_σ for expressions in assignments $c := a$. These are defined as

- $E_\Sigma(A) = \Pi u\colon \mathtt{tp}.\ A^{\Sigma,u}$
- $\varepsilon_\sigma(a) = \lambda u\colon \mathtt{tp}.\ a^{\Sigma,u}$ where Σ is the codomain of σ

where the function $-^{\Sigma,u}$ replaces every occurrence of a constant c declared in Σ with $c\,u$. Here, u is any variable name that does not occur bound anywhere in the argument.

Applying the definition, we see that $\mathtt{Poly}(X)$ maps declarations in X as follows:

– Every type $t\colon \texttt{tp}$ yields a unary type operator $t\colon \Pi u\colon \texttt{tp}.\ \texttt{tp}$ (i.e., $t\colon \texttt{tp} \to \texttt{tp}$).
– Every function symbol $f\colon \texttt{tm}\ t_1 \to \ldots \to \texttt{tm}\ t_n \to \texttt{tm}\ t$ yields a polymorphic function symbol $f\colon \Pi u\colon \texttt{tp}.\ \texttt{tm}\ t_1 u \to \ldots \to \texttt{tm}\ t_n u \to \texttt{tm}\ t u$. Thus f abstracts over an arbitrary type u and replaces every X-type symbol t with the type $t\,u$, which arises by applying the $\texttt{Poly}(X)$-unary type operator t to u.
– Every predicate symbol is mapped in essentially the same way as function symbols.
– Every axiom $c\colon \vdash F$ yields an axiom $c\colon \Pi u\colon \texttt{tp}.\vdash F^{\Sigma,u}$ where the operation $-^{\Sigma,u}$ now affects all type, function, and predicate symbols occurring in F.

Theorem 7. $\texttt{Poly}(-)$ *is a strongly linear operator.*

Proof. We only need to show that `Poly` indeed yields well-typed theories and morphisms. For theories, that is obvious. For morphisms, the critical step of the inductive proof considers a `TFOL`-morphism $id_{\texttt{TFOL}}, \sigma, c := a$ from $\texttt{TFOL}, \Sigma, c\colon A$ to $\texttt{TFOL}, \Sigma'$. Applying the definition of `Poly`, we see that the well-typedness of the resulting morphism hinges on $\varepsilon_\sigma(a)\colon \overline{\texttt{Poly}(id_{\texttt{TFOL}}, \sigma)}(E_\Sigma(A))$ to hold over $\texttt{Poly}(\texttt{TFOL}, \Sigma')$. That follows from the definitions of E and ε and the fact that $a\colon \overline{id_{\texttt{TFOL}}, \sigma}(A)$ over Σ'.

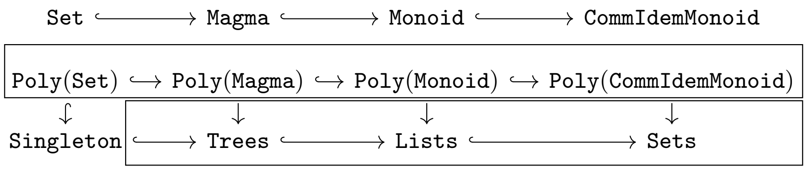
$$\begin{tikzcd}
\texttt{Set} \arrow[r, hook] & \texttt{Magma} \arrow[r, hook] & \texttt{Monoid} \arrow[r, hook] & \texttt{CommIdemMonoid} \\
\texttt{Poly(Set)} \arrow[r, hook] \arrow[d, hook] & \texttt{Poly(Magma)} \arrow[r, hook] \arrow[d] & \texttt{Poly(Monoid)} \arrow[r, hook] \arrow[d] & \texttt{Poly(CommIdemMonoid)} \arrow[d] \\
\texttt{Singleton} \arrow[r, hook] & \texttt{Trees} \arrow[r, hook] & \texttt{Lists} \arrow[r, hook] & \texttt{Sets}
\end{tikzcd}$$

Fig. 4: Diagrams `Magmas`, `Singleton` and (boxed) results of applying `Poly`

Consider Figure 4 for an example application: the first row draws and extends on a fragment of `Magmas`, while the second row shows the corresponding theories and morphisms emerging from **install** `Poly(Magmas)`. For instace, $\texttt{Poly(Monoid)}^\flat$ is given by

$$U\colon \texttt{tp} \to \texttt{tp},\ \circ : \Pi u\colon \texttt{tp}.\ \texttt{tm}\ U u \to \texttt{tm}\ U u \to \texttt{tm}\ U u,\ e\colon \Pi u\colon \texttt{tp}.\ \texttt{tm}\ U u,\ \ldots$$

where we omit the axioms. That already looks very close to the theory of lists with $U\,u$ for the type of lists over u and $\circ$ and e for concatenation and empty list. Similarly, magmas, commutative monoids, and commutative-idempotent monoids yield theories close to the collection data types for trees, multisets, and sets over a type u.

In all of these cases, the only thing missing to complete the specification of the collection data type is an operator for creating singleton trees, lists, and sets. We can uniformly inject it into all theories in the diagram by defining the theory

$$\texttt{Singleton} = \{\textbf{include}\ \texttt{Poly(Set)},\ \texttt{singleton}\colon \Pi u\colon \texttt{tp}.\ \texttt{tm}\ u \to \texttt{tm}\ U u\}$$

and then applying $P_{\texttt{Singleton}}$ to $\texttt{Poly}(\texttt{Magmas})$ where $P_{\texttt{Singleton}}$ takes the pushout along the inclusion $\texttt{Poly}(\texttt{Set}) \hookrightarrow \texttt{Singleton}$. The result is shown in the third row in Figure 4.

5 Conclusion and Future Work

We introduced a class of structure-preserving functorial diagram operators, and showed how to define them easily and lift them to large diagrams. They allow building small diagram expressions that evaluate to large diagrams whose structure remains intuitive and predictable to users.

They can be seen as different degrees of compositionality-breaking: pushout-based translations are induced by the homomorphic extension of a morphism and thus entirely compositional; strongly linear operators use arbitrary expression translation functions and extend them compositionally to declarations and theories; linear operators use arbitrary declaration translation functions and extend them compositionally to theories; finally, the most general class translates theories without any constraint. Thus, our work can be seen as identifying good trade-offs between the rather restrictive pushout-based and the unpredictable arbitrary operators.

A drawback of our approach is the lack of a static type system at the diagram expression level: all our diagram operators are partial, and the only way to check that $O(D)$ is defined is to successfully evaluate it. That is frustrating, but our experiments in this direction have indicated that any type system that could predict definedness would be too complicated to be practical. Moreover, in practice, the immediate evaluation of diagram expressions is needed anyway for two reasons: Firstly, many operators can only be type-checked if their arguments are fully evaluated, thus obviating the main advantage of static type-checking. Secondly, because diagram expressions introduce theories that are to be used as interfaces later, their evaluation is usually triggered soon after type-checking.

We presented two applications centered around the algebraic hierarchy. We chose these examples because their simplicity and partial overlap allowed for a compact presentation. But linear operators occur widely in many formal systems. In particular, compared to the pushout-based definitions of logic translations used in the LATIN atlas [CHK+11], linear operators significantly increase the expressivity while retaining some desirable structural properties.

Future work will focus on defining and implementing more operators such as the ones from universal algebra or the gaps in the LATIN atlas. This can now be done extremely efficiently. We will also present the generalization of our results to all structuring features of Mmt, in particular renaming and translation, in a longer report. Additionally, it is interesting to extend our work to the framework of institutions [GB92]. This would replace Mmt with an abstract definition of declaration-based theories such as an institution with symbols [Mos99]. Then structure-preservation could be restated in terms of models and satisfaction, e.g., by relating include-preservation to forgetful model reduction.

References

Cap99. V. Capretta. Universal algebra in type theory. In Yves Bertot, Gilles Dowek, André Hirschowits, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99*, pages 131–148. Springer, 1999.

CAPM15. P. Castro, N. Aguirre, C. López Pombo, and T. Maibaum. Categorical foundations for structured specification in Z. *Formal Asp. Comput*, 27(5-6):831–865, 2015.

CB16. D. Christiansen and E. Brady. Elaborator reflection: extending idris in idris. In J. Garrigue, G. Keller, and E. Sumii, editors, *International Conference on Functional Programming*, pages 284–297. ACM, 2016.

CFS20. J. Carette, W. Farmer, and Y. Sharoda. Leveraging information contained in theory presentations. In C. Benzmüller and B. Miller, editors, *Intelligent Computer Mathematics*. Springer, 2020. to appear.

CHK$^+$11. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, and F. Rabe. Project Abstract: Logic Atlas and Integrator (LATIN). In J. Davenport, W. Farmer, F. Rabe, and J. Urban, editors, *Intelligent Computer Mathematics*, pages 289–291. Springer, 2011.

CHK$^+$12. M. Codescu, F. Horozal, M. Kohlhase, T. Mossakowski, F. Rabe, and K. Sojakova. Towards Logical Frameworks in the Heterogeneous Tool Set Hets. In T. Mossakowski and H. Kreowski, editors, *Recent Trends in Algebraic Development Techniques 2010*, pages 139–159. Springer, 2012.

CMR17. M. Codescu, T. Mossakowski, and F. Rabe. Canonical Selection of Colimits. In P. James and M. Roggenbach, editors, *Recent Trends in Algebraic Development Techniques*, pages 170–188. Springer, 2017.

CO12. J. Carette and R. O'Connor. Theory Presentation Combinators. In J. Jeuring, J. Campbell, J. Carette, G. Dos Reis, P. Sojka, M. Wenzel, and V. Sorge, editors, *Intelligent Computer Mathematics*, volume 7362, pages 202–215. Springer, 2012.

CoF04. CoFI (The Common Framework Initiative). *CASL Reference Manual*, volume 2960 of *LNCS*. Springer, 2004.

DOL18. DOL editors. The distributed ontology, modeling, and specification language (dol). Technical report, Object Management Group, 2018.

EUR$^+$17. G. Ebner, S. Ullrich, J. Roesch, J. Avigad, and L. de Moura. A Metaprogramming Framework for Formal Verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):34:1–34:29, 2017.

FGT93. W. Farmer, J. Guttman, and F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11(2):213–248, 1993.

GB92. J. Goguen and R. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.

GGMR09. F. Garillot, G. Gonthier, A. Mahboubi, and L. Rideau. Packaging mathematical structures. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 327–342. Springer, 2009.

GWM$^+$93. J. Goguen, Timothy Winkler, J. Meseguer, K. Futatsugi, and J. Jouannaud. Introducing OBJ. In J. Goguen, D. Coleman, and R. Gallimore, editors, *Applications of Algebraic Specification using OBJ*. Cambridge, 1993.

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.

HR20. M. Haveraaen and M. Roggenbach. Specifying with syntactic theory functors. *J. Log. Algebraic Methods Program*, 113:100543, 2020.

KWP99. F. Kammüller, M. Wenzel, and L. Paulson. Locales – a Sectioning Concept for Isabelle. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Thery, editors, *Theorem Proving in Higher Order Logics*, pages 149–166. Springer, 1999.

Mos99. T. Mossakowski. Specifications in an Arbitrary Institution with Symbols. In D. Bert, C. Choppy, and P. Mosses, editors, *Workshop on Recent Trends in Algebraic Development Techniques*, pages 252–270. Springer, 1999.

MR19. D. Müller and F. Rabe. Rapid Prototyping Formal Systems in MMT: Case Studies. In D. Miller and I. Scagnetto, editors, *Logical Frameworks and Meta-languages: Theory and Practice*, pages 40–54, 2019.

Rab17a. F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

Rab17b. F. Rabe. Morphism Axioms. *Theoretical Computer Science*, 691:55–80, 2017.

RK13. F. Rabe and M. Kohlhase. A Scalable Module System. *Information and Computation*, 230(1):1–54, 2013.

SJ95. Y. Srinivas and R. Jüllig. Specware: Formal Support for Composing Software. In B. Möller, editor, *Mathematics of Program Construction*. Springer, 1995.

SR19. Y. Sharoda and F. Rabe. Diagram Operators in MMT. In C. Kaliszyk, E. Brady, A. Kohlhase, and C. Sacerdoti Coen, editors, *Intelligent Computer Mathematics*, pages 211–226. Springer, 2019.

ST88. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Control*, 76:165–210, 1988.

SW83. D. Sannella and M. Wirsing. A Kernel Language for Algebraic Specification and Implementation. In M. Karpinski, editor, *Fundamentals of Computation Theory*, pages 413–427. Springer, 1983.