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. We investigate the properties of diagram operators and give numerous examples. All our examples are formalized in the Mmt meta-logical framework.

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 at once. For example, we can form the diagram `Magmas` of the magma-hierarchy (including theories for semigroup, commutative magma, etc.), apply an operator to enrich the diagram with all unions (band, semilattice, etc.), and finally apply another operator to obtain in one step the corresponding diagram of homomorphisms (including theories for band 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 commutes with flattening.

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 diagram operators developed in [SR19] on top of the Mmt framework for building formal systems [Rab17a]. Every diagram operator is declared as an Mmt symbol and implemented by a computation rule in the underlying programming language. Thus, all our operators are directly usable in Mmt specifications, and all our examples have been formalized in that way. Because Mmt is foundation-independent, these diagram operators are immediately applicable to any logic defined in any logical framework implemented in Mmt. This includes any logic defined in the LATIN logic atlas [CHK+11], which itself is defined within the LF logical framework [HHP93]. While most interesting operators are logic-specific, we can maximize reuse by implementing all our operators relative to the weakest possible logic.

Related Work In parallel to the present work, [JCS20] 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.

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 emphasize the general framework in which these operators are defined and allows applications to structured theo-

ries. 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 Sect. 2. Then we develop our main results in Sect. 3 and present example applications in Sect. 4 and 5. We conclude in Sect. 6.

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 realized in Mmt [Rab17a], which induces a language for 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 Fig. 1. The basic **theories** are inspired by LF-style languages and are lists of typed/kinded declarations $c : A$. Correspondingly, the basic **morphisms** v, say from a theory S to a theory T, are lists of assignments $c := a$ such that, if $c : A$ is declared in S, then a must be a T-expression of type $\overline{v}(A)$, where $\overline{v}$ is the homomorphic extension of v. Further details of the type system are not essential for our purposes, and we refer to [Rab17a].

Diag	::=	$(\mathit{Thy} \mid \mathit{Morph} \mid \textbf{diagram}\ D)^*$	diagrams
Thy	::=	$T = \{\mathit{Decl}^*\}$	theory definition
Decl	::=	$c\colon A\,\underline{[= A]} \mid \underline{\textbf{include}\ T}$	declarations in a theory
Morph	::=	$v\colon S \to T = \{\mathit{Ass}^*\}$	moprhism definition
Ass	::=	$c := A \mid \underline{\textbf{include}\ v}$	assignments in a morphism
A	::=	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. Here `prop` and `tp` are the LF-types holding the `TFOL`-propositions and `TFOL`-types, respectively. For any `TFOL`-type $A\colon$ `tp`, the LF-type `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 $F :$ `prop` is captured by the type $\vdash F$, which holds the proofs of F.

`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, $f\colon \mathtt{tm}\ t_1 \to ... \to \mathtt{tm}\ t_n \to \mathtt{tm}\ t$ for a funtion symbol, $p\colon \mathtt{tm}\ t_1 \to ... \to \mathtt{tm}\ t_n \to \mathtt{prop}$ 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 $\mathit{list}\colon \mathtt{tp} \to \mathtt{tp}$ and $\mathit{cons}\colon \Pi A\colon \mathtt{tp}.\ \mathtt{tm}\ A \to \mathtt{tm}\ \mathit{list}\,A \to \mathtt{tm}\ \mathit{list}\,A$.

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

The underlined parts in the grammar are the structuring principles that LF inherits from MMT. 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 $\overline{v}(c) = \overline{a}$. **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 feature, but we omit them here due to space constraints.

The semantics of MMT structuring features is given by the flattening operation $-^\flat$ that transforms structured theories and morphisms into basic 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 an order relation on theories.

$$\begin{array}{rlrl} T^\flat &:= \Sigma^\flat \quad \text{if } T = \{\Sigma\} & v^\flat &:= \sigma^\flat \quad \text{if } v\colon S \to T = \{\sigma\} \\ \cdot^\flat &:= \emptyset & \cdot^\flat &:= \emptyset \\ (c : A, \Sigma)^\flat &:= \{c\colon A^\delta\} \cup \Sigma^\flat & (c := A, \sigma)^\flat &:= \{c := A^\delta\} \cup \sigma^\flat \\ (c\colon A = a, \Sigma)^\flat &:= \Sigma^\flat & & \\ (\textbf{include}\ S, \Sigma)^\flat &:= S^\flat \cup \Sigma^\flat & (\textbf{include}\ w, \sigma)^\flat &:= w^\flat \cup \sigma^\flat. \end{array}$$

Fig. 2: Flattening

[SR19] introduced **diagram expressions** D and added a third toplevel declaration to MMT for installing all theories andmorphisms in a diagram expression. For our purposes, it is sufficient to consider only two simple cases:

Diagram(T^*, v^*) builds a basic diagram by including some previously defined theories and morphisms. And $O(D)$ applies a diagram operator O to the diagram D. O is simply an identifier that MMT binds to a user-provided computation rule implemented in the underlying programming language.

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 $c : A$,
- the well-formedness of theories is checked declaration-wise, i.e., $\Sigma, c : A$ 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.

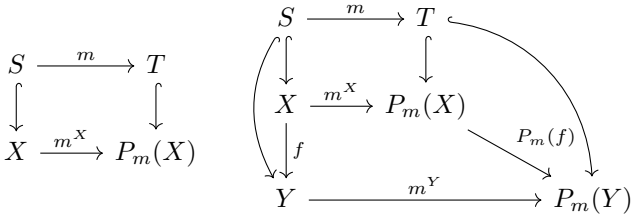
Thus, in a straight language, $\mathrm{WF}_\Sigma(A)$ can be reduced to $\mathrm{WF}_{\Sigma_0}(A)$ where Σ_0 consists of the smallest subset of declarations in Σ, whose names transitively occur in A. We call Σ_0 the **dependencies** of c. In a straight language, it is easy and harmless to **identify 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 that 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 $m\colon S \to T$. It 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 $P_m(X)$.



These are defined as follows:

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

We can observe a number of structural properties that are helpful for implementing this operator at large scales and that we often see in diagram 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, X, Y, and f) to morphisms between extensions of T that agree with id_T. This situation is very common when transporting developments relative to a base language S to a different base language T. For example, a logic translation from logic S to logic T typically involves a functor that maps S-theories (i.e., extensions of S) to T-theories (i.e., extensions of T).

Secondly, it is defined declaration-wise: every declaration in the input yields one declaration of the same name in the output. Moreover, the output only depends on the input declaration and not on what other declarations occur before or after it. This is very common for interesting theory operators, which are typically defined by induction on the list of declarations in a theory. This immediately yields that pushout preserves includes: if X is included into Y, then $P_m(X)$ is included into $P_m(Y)$. That allows extending pushout to diagrams of structured theories: if we map every theory T in a diagram to T', we can map every declaration **include** S to **include** S' and thus preserve the structure.

Thirdly, pushout extends to definitions. If the input contains the declaration $c\colon A = a$, we can generate the definition $c\colon \overline{m^{S,\Sigma}}(A) = \overline{m^{S,\Sigma}}(a)$ in the output. 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 $P_m(X)$. This is also possible for many interesting theory operators.

3.2 Linear Operators

We will induce diagram operators from functors on flat theories. More precisely, we will usually have to work with partial functors. We define a **partial functor** from C to D as a functor from a subcategory of C to D. Thus, if a partial functor is defined for an object X, it is also defined for id_X, and if it is defined for morphisms f, g, it is also defined for their domains, codomains, and their composition.

We will usually work with *partial* endofunctors on LF theories. There are two reasons for this. Firstly, many diagram operators are logic-specific, e.g., we can define the typical notion of model homomorphisms for many variants of first-order logic, but not for higher-order logics or for LF in general. Therefore, we usually need to restrict an operator to certain LF-theories, e.g., to those extension of `TFOL` that represent theories of typed first-order logic.

Secondly, defining diagram 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 simply be to leave the operator undefined for some inputs.

Taking partiality into account and abstracting from the properties of the pushout operator, we arrive at the following definition:

Definition 1 (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:*

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

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

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

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

We see easily that every linear operator **preserves inclusions**, i.e., if Σ is included into Σ' then so is $O(\Sigma)$ into $O(\Sigma')$. The converse does not hold. For example, we can define $O(\Sigma)$ to contain some declaration for every pair of declarations in Σ. This operator still preserves inclusions but is not linear.

Note that Δ and δ are uniquely determined by O. Vice versa, 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 A. 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.

$\Delta_\Sigma(-)$ need not be a single declaration — it could be a list of declarations. Pushout also satisfies the following stronger property:

Definition 2 (Strongly Linear Operator). *A linear operator is called **strongly linear** if $\Delta_\Sigma(c\colon A)$ always contains exactly one declaration that 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 via $\Delta_\Sigma(c\colon A) = c\colon E_\Sigma(A)$ and $\delta_\sigma(c := a) = c := \varepsilon_\sigma(a)$. That makes them even simpler to implement as users only need to worry about the inductive expression translation function and the framework can take over all bureaucracy for names and declarations. Moreover, it is even simpler to check that two arbitrary function (E, ε) induce a strongly linear operator. We omit the details, but essentially we only have to check that $a\colon A$ implies $\varepsilon(a)\colon E(A)$.

Note that for pushout, 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 where ε inserts a λ.

3.3 Structure-Preserving Lifting of Linear Operators

Our goal now is to extend linear operators O from L to M — that 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 L and every morphism by including id_L. Correspondingly, theories and morphisms in $O(D)$ will include M and id_M, 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, L 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 D and M. In particular, this means that no theory in D may have an include declaration for M or any named theory included into M. This is usually true in practice anyway because L and M usually represent formal systems and D represents domain knowledge written in L.

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 random but obtained from the names in D in a systematic way that is predictable for the user. That is important so that users can 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 $T = \{\Sigma\}$, we could obtain the theory definition $\texttt{pushout}.m.T = \{P_m(\Sigma)\}$. Therefore, we define:

Definition 3. *An operator O is **liftable** if it additionally provides a function that maps names to names. We denote this function also by $O(-)$.*

Thus, we can map the theory definition $T = \{\Sigma\}$ to the theory definition $O(T) = \{O(\Sigma)\}$. To avoid confusion, keep in mind that S, T, v, w are always names, which 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 4 (Lifting). *Given a liftable linear operator O, we define its lifting $O(-)$ to diagrams as follows:*

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

$$O(\cdot) = \cdot \qquad O(\Sigma,\ c\colon A) = O(\Sigma),\ \Delta_\Sigma(c\colon A)$$
$$O(\Sigma,\ c\colon A = a) = \mathrm{AsDef}(\delta_{id_\Sigma}(c := a))$$
$$O(\Sigma,\ \textbf{include}\ S) = O(\Sigma),\ \textbf{include}\ O(S)$$

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

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

Definition 4 looks complex but is mostly straightforward. It uses Δ 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 $\Sigma,\ c : A$ to Σ. This is easy to see — in both cases, 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 the resulting assignments back into definitions. Using the morphism above, O yields the morphism $O(id_\Sigma),\ \delta_{id_\Sigma}(c := a)$ from $O(\Sigma),\ \Delta_\Sigma(c\colon A)$ to $O(\Sigma)$. The function $\mathrm{AsDef}(-)$ then turns this morphism back into a sequence of definitions: it replaces every assignment $c' := a'$ for a constant $c'\colon A'$ in $\Delta_\Sigma(c\colon A)$ with the definition $c'\colon A' = a'$.

$\mathrm{AsDef}(-)$ 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 anyway.

Remark 1 (Strongly Linear Lifting). Every strongly linear operator O with expression translation functions (E, ε) is linear and liftable (via the identity name map for constants and an arbitrary name map for theories and morphisms). Thus its lifting 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 Semantics Preservation of the Lifting

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

Theorem 1 (Semantics Preservation). *Consider a well-formed diagram D and an operator O as in Definition 4. Then, $O(D)$ is a well-formed diagram and*
- *for every theory named T defined in D, we have $O(T^\flat) = O(T)^\flat$*
- *for every morphism named v defined in D, we have $O(v^\flat) = O(v)^\flat$.*

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 4.

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

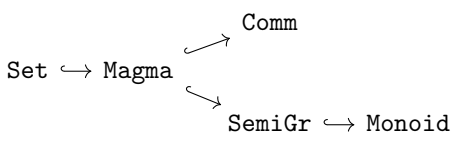
To account for definitions, we observe that a constant c' is defined in $O(T)$ iff. the lifting generates it when translating a defined constant c in T. Moreover, linearity guarantees that the lifting handles undefined constants in the same way regardless of whether defined constants in the same theory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

4 Applications to Universal Algebra

Universal algebra provides many constructions that apply to arbitrary TFOL-theories most of which can be extended in a way that preserves structuring. Moreover, 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 of these operators as examples: the ones that map a TFOL-theory T (whose models are the T-models) to the TFOL-theories of T-homomorphisms (whose models are the homomorphisms between T-models) and T-submodels (whose models are the pairs of a T-model and a submodel of it), respectively. Other constructions such as product and quotient models can be handled accordingly.

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



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

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

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

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

Homomorphisms We define the operator `Hom` such that it is linear by construction and define only Δ and δ. $\Delta_\Sigma(-)$ maps each type, function, or predicate symbol declaration to three declarations: one each for the domain and codomain of the homomorphism, which just duplicate $c\colon A$ under two different renamings, and one for the actual homomorphism. Axioms are mapped to only two declarations for domain and codomain:

- In a type symbol declaration, we have $A = \texttt{tp}$, and $\Delta(t\colon A)$ contains

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

  Here the superscripts indicate different systematic renamings of the constant t.
- In a function symbol declaration, A is of the form $\texttt{tm}\ t_1 \to \ldots \to \texttt{tm}\ t_n \to \texttt{tm}\ t$, and $\Delta_\Sigma(f\colon A)$ contains

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

  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 $\texttt{tm}\ t_1 \to \ldots \to \texttt{tm}\ t_n \to \texttt{prop}$, and $\Delta_\Sigma(p\colon A)$ contains

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

- 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 $T = \{\Sigma\}$, we define a theory morphism $T^d\colon T \to \texttt{Hom}(T) = \{\sigma^d\}$ where σ^d contains **include** S^d for every **include** S in Σ and $s := s^d$ for every constant s in Σ. We define a morphism $T^c\colon T \to \texttt{Hom}(T) = \{\Sigma^c\}$ accordingly. Thus, `Hom` is more than a functor: it maps a single theory to the diagram

$$T \xrightarrow[T^c]{T^d} \texttt{Hom}(T).$$

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, $\delta_\sigma(f := E) = t^d := E^d,\ t^c := E^c,\ t^h := P$. Here P is a proof that

$$\forall\, x_1\colon t_1^d, \dots, x_n\colon t_n^d.\ \ t^h(E^d(x_1, \dots, x_n)) \doteq E^c(t_1^h(x_1), \dots, t_n^h(x_n)),$$

  a property provable in `TFOL` for every well-typed expression E.
- For a predicate symbol $p\colon A$ as above, $\delta_\sigma(p := E) = t^d := E^d,\ t^c := E^c,\ t^h := P$. Here P is a proof that

$$\forall\, x_1\colon t_1^d, \dots, x_n\colon t_n^d.\ \ E^d(x_1, \dots, x_n) \Rightarrow E^c(t_1^h(x_1), \dots, t_n^h(x_n)),$$

  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 2. `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 **diagram** `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`:

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

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

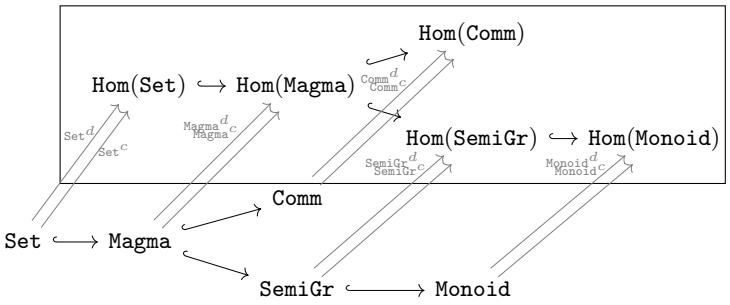


Fig. 3: Diagrams `Magmas` and (boxed) `Hom(Magmas)`

Submodels We proceed in essentially the same way. 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 declaration building the submodel. We define Δ by:

- Every type symbol declaration $t \colon \texttt{tp}$ is mapped to two declarations: $t^p \colon \texttt{tp}$ for the parent model and $t^s \colon \texttt{tm}\ t^p \to \texttt{prop}$ 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

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

  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 by relativizing every quantifier on type t using the predicate t^s.

In the same way as for `Hom`, we obtain a morphism $T^p \colon T \to \texttt{Sub}(T)$ that assigns $c := c^p$. Hypothetically, if we worked in a stronger logic that allows taking subtypes, we could define a morphism $T^s \colon T \to \texttt{Sub}(T)$ 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 $f^p := E^p$ and $f^s := P$ where P is the 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 by adapting all occurrences of quantifier rules to the relativized variants.

Theorem 3. *The operator* $\texttt{Sub}(-)$ *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 **diagram** `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 $\texttt{Sub}(T)$ are in fact provable. For example, all axioms using only $\forall$ and $\doteq$ (i.e., the axioms primary 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 we 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)`. Thus, it is desirable that `Hom` adds a definition to them. But that makes it trickier to preserve the structure of the diagram: 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)`.

5 An Operator for Polymorphic Generalization

We give a linear operator $\texttt{Poly}(-)$ 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 $f\colon \Pi\, u_1\colon \mathtt{tp}, \ldots, \Pi\, u_m\colon \mathtt{tp}.\ \mathtt{tm}\ t_1 \to \ldots \to \mathtt{tm}\ t_m \to \mathtt{tm}\ t$, 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 $E_\Sigma(A)$ for types in declarations $c\colon A$ and $\varepsilon_\sigma(a)$ for expressions in assignments $c := a$. These are defined as follows

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

where the function $-^{\Sigma,u}$ replaces every occurrence of a constant c declared in Σ with $c\,u$.

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

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

Theorem 4. $\mathtt{Poly}(-)$ *is a strongly linear operator.*

Proof. We only need to show 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 $id_{\mathtt{TFOL}}, \sigma, c := a$ from $\mathtt{TFOL}, \Sigma, c\colon A$ to $\mathtt{TFOL}, \Sigma'$. Applying the definition of Poly, we see that the well-typedness of the resulting morphism hinges on $\varepsilon_\sigma(a)\colon E_\Sigma(A)$ to hold over $\mathtt{Poly}(\mathtt{TFOL}, \Sigma')$. That follows immediately from the definition of E and ε.

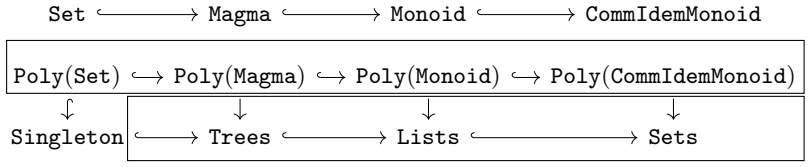


Fig. 4: Diagrams Magmas, Singleton and (boxed) results of applying Poly

The first two rows in Figure 4 show a fragment of the diagram **diagram** Poly(Magmas). For example, $\mathtt{Poly}(\mathtt{Monoid})^\flat$ is given by

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

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 lists, sets, etc. 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 $\texttt{Poly(Magmas)}$ where $P_{\texttt{Singleton}}$ takes the pushout along the inclusion $\texttt{Poly(Set)} \hookrightarrow \texttt{Singleton}$. These are shown in the third row in Figure 4.

6 Conclusion and Future Work

We introduced a class of functorial diagram operators with particularly nice properties, 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 translations 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 $O(T)$ 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. These applications are just simple examples, and linear operators occur much more widely than our examples may indicate. In fact, we chose the examples because their simplicity and partial overlap allowed for a compact presentation.

More generally, our results apply to any logic defined in LF, and in fact almost any logical framework developed on top of Mmt. In particular, in the area of language translations, linear operators occur frequently. For example, the LATIN atlas [CHK+11] did not include many interesting logic translations

because it used the pushout operator as the main form of representing logic translations. Our work generalizes that approach and allows the representation of most practically relevant translations, while (and this is the novelty) retaining many of the valuable properties of pushout-based definitions.

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.

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.

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.

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.

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.

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.

JCS20. W. Farmer J. Carette and Y. Sharoda. Leveraging information contained in theory presentations. In C. Benzmüller and B. Miller, editors, *Intelligent Computer Mathematics*. Springer, 2020. to appear.

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.

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.